Refining, Implementing, and Evaluating the Extended Continuous Variable-Specific Resolutions of Feature Interactions

by

Chi Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Systems that involve feature-oriented software development suffer from **feature interactions**, in which features affect one another's behaviour in surprising ways. As the number of features increases, the complexity of examining feature combinations and fixing undesired interactions increases exponentially, such that the workload of resolving interactions comes to dominate feature development. The **Feature Interaction Problem** results from aiming resolve feature interaction by providing optimal resolutions. Resolution strategies combat the Feature Interaction Problem by offering default strategies that resolve entire classes of interactions, thereby reducing the work of the developer who is charged with the task of resolving interactions. However, most such approaches employ coarse-grained resolution strategies (e.g., feature priority) or a centralized arbitrator. This thesis focuses on evaluating and refining a proposed architecture that resolves features' conflicting actions on system's outputs. In this thesis, we extend a proposed architecture based on variable-specific resolution to enable co-resolution of related outputs and to promote smooth continuous resolutions over execution sequences. We implemented our approach within the PreScan simulator for advanced driver assistance systems, and performed a case study involving 15 automotive features that we implemented. We also devised and implemented three resolution strategies for the features' outputs. The results of the case study show that the approach produces smooth and continuous resolutions of interactions throughout interesting scenarios.

Acknowledgements

I would like to thank my supervisor, Jo Atlee. She is kind and enthusiastic. She spent a lot of time to help me not only on supervision and research, but also assisting me get used to study and life in a new country. I also want to thank my readers, Nancy Day and Krzysztof Czarnecki, for their precious advice and time.

Thank you all the members in our group. They all enlightened me and supported me a lot. Especially thank to M. H. Zibaeenejad. I had a great time working with him.

Finally, I would like to thank my parents and fiancee. Altough they are far away in China, they are always my biggest spiritual support, helping me out whenever I'm in plight.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, along with the development of a software system's size and complexity, the variability of a software system grows rapidly as well. More and more requirements arise from increments to a system's functionality and variability (sometimes across multiple platforms), which makes decomposing the system into units of functionality an option to addressing this complexity. **Feature-oriented software development** (FOSD) realizes a decomposition by functionality or by feature. A **feature** is a unit of functionality of a software system that satisfies a requirement, represents a design decision, or provides a potential configuration option. The basic idea of FOSD is to decompose a software system in terms of the features it provides. The goal of the decomposition is to construct well-structured software that can be tailored to the needs of the user and the application scenario; through the selection of optional features. Typically, from a set of features, many different software systems can be generated that share common features and differ in their selection of optional features [2].

Using feature orientation and FOSD in a large-scale system has multiple advantages. First, it can facilitate the structure, reuse, and variation of software in a systematic and uniform way. Since features encapsulate functionalities, they provide a much more convenient way to release system updates via additions, deletions or updates to features. Second, feature decomposition also enables parallel development by different teams. For example, in automotive software development, different teams can work independently on different features, such as Cruise Control and Speed Limit Control features that both aim to enhance driver safety. Additionally, another benefit of feature orientation is that features can serve as a shared vocabulary among diverse stakeholders (e.g., marketers, customers, other engineers) in way that other types of software fragments - such as modules, objects, or components - cannot [5].

However, despite the advantages of feature orientation and FOSD, there are some complications that arise from treating features as truly separate concerns. Problems occur when developers try to integrate several features into a coherent system. For example, consider telephony, which typically offers a set of features including Call Forwarding and Call Waiting. These two features both help a user who is involved in one call to not miss a second call. When the customer is already using the phone and a new call comes in, the Call Waiting feature will enable the user to receive notification of the new incoming call and have the option of putting either the incoming call or the current call on hold. Whereas the Call Forwarding feature will forward the new call to a pre-specified secondary phone number. Consider the case where a customer subscribes to both Call Forwarding and Call Waiting. When a first call arrives, there is no problem because no feature is activated. However, when a second call arrives before the first call has ended, the system has to determine whether the new call should be forwarded to the secondary number (Call Forwarding) or whether the user should be notified that another call has arrived (Call Waiting). This decision has no obvious correct answer; the optimal solution depends on the customer's requirements. This is a typical **feature interaction**: the feature combination exhibits unexpected behaviour that does not occur when either feature executes in isolation.

There are multiple ways in which features can interact with one another. Some feature interactions are expected and desirable, such as Call Waiting and Block Call Waiting, where Block Call Waiting is designed to interact with and override Call Waiting. Other interactions are potentially undesirable, such as the interaction between Call Forwarding and Call Waiting as described above. In this thesis, we are primarily focused on conflict interactions. A **conflict interaction** is a type of feature interaction that occurs when multiple features attempt to assign different values to the same instance of an output variable in the same execution step [21].

Conflict interactions need to be resolved. A **resolution** is a kind of strategy that can define an appropriate system behaviour in the circumstance in which a feature interaction occurs. An appropriate resolution strategy depends on the structure of the system, the user's requirements, the developer's intentions in specifying feature behaviour, the nature of the interaction, and so on. It can also depend on the circumstances and context in which the interaction occurs, and can even depend on which other features are active in the system. Thus, the complexity of what it means to resolve a feature interaction increases as system complexity and the number of features grows - especially if an appropriate resolution is conditional on the **combination** of active features. In such a case, the work to resolve feature interactions can grows exponentially. This is the essence of the **Feature Interaction Problem**: the need to analyze all feature combinations, resolve all undesired

interactions, and verify all resolutions.

Rather than trying to resolve individually every feature interaction due to conflict, one feasible approach to solving the Feature Interaction Problem is to provide a default resolution strategy to resolve interactions, thereby reducing the number of interactions that developers must address individually. The goal of using default resolution strategy is not necessarily to resolve all conflict interactions, but to resolve a majority of interactions, thusly leaving many fewer that need special attention from the engineers. Default resolution strategies include resolution by feature priority [16, 19, 25], feature precedence [4, 9, 18], rolling back conflicting actions [25], disabling feature activation [30], terminating features [25], and terminating the application. Most of these strategies are coarse grained (e.g., based on the priority or precedence of the features themselves rather than on the features' interacting actions). They provide suboptimal win/lose resolutions; that is, some features' actions are sacrificed in favor of other features' actions. Moreover, such approaches often require pre-determined total or partial ordering on features [31].

To address these weaknesses, Bocovich and Atlee [5] proposed in 2014 a new architecture model that promotes feature-interaction resolution that is specific to the phenomena being acted on. Rather than devising one resolution strategy to handle all feature interactions, their approach allows developers to program appropriate default strategies for resolving conflict interactions on each system output, thereby specializing the default resolution strategies to these outputs. The results are win/win resolutions that do not require a total or partial ordering on features. Moreover, the amount of default-resolution work done by the developer is linear in the number of types of output variables that are modified by multiple features, rather than exponential in the number of features. It defines a resolution module for each system output, such as a system actuator that is controlled by a feature in order to satisfy its requirements. Each resolution module takes as input all of the features' actions on its respective system output and, through a variable-specific resolution strategy, assigns a conflict-free value to the variable. Thus, the resolution module is variable specific and feature agnostic.

The variable-specific resolution architecture was subsequently adapted for dynamic systems. In this work, Zibaeenejad proposed that it is necessary to separate feature logic from control logic within features, and to distinguish functional features from actuator features. He also proposed an extension to accommodate coupled variables. However, his proposed architecture was theoretical and had never been used or evaluated in an executable setting. The goal of this thesis has been to assess this proposed architecture in an executable environment and provide extensions to solve problems that arose during experiments.

Before we continue, in order to clearly distinguish which architecture variant we are

referring to at any point in the thesis, we define names for each of the different phases of the architecture of development. The original architecture that was proposed in [5] and that promotes variable-specific resolution is called the "preliminary architecture". The starting point of our evaluation, which is proposed by my colleague Zibaeenejad, promotes continuous variable-specific resolution, and is called the "extended architecture". This thesis further extends the architecture to promote smooth continuous resolutions, and is called the "current architecture".

Thesis Statement

The current architecture shows that it is possible to provide smooth and continuous resolutions of interactions and accommodate coupled variables' resolution modules. This not only preserves all of the advantages of the original variable-specific resolution architecture, but also satisfies by construction global system properties. The current architecture can produce resolutions that result in acceptable outputs over a sequence of continuous steps when applied to semi-autonomous system, thereby providing a comfortable and safe driving experience.

1.1 Contributions

The contributions of this thesis are as follows:

1. We evaluate and refine a dynamic system's run-time approach to resolving feature interactions, using programmed variable-specific resolution modules. The refinement is:
   (a) Allow different types of inputs to resolution modules to improve the stability of system outputs over a sequence of actions.
2. We present the implementation of the extended variable-specific resolution architecture within a simulator for semi-autonomous vehicle. We provide resolution strategies for attribute variables in the case study.
3. We performed a case study to assess the quality of resolutions produced by the current architecture and by the resolution modules. The case study involves 15 automotive features and 3 resolution modules. The results of the case study demonstrate that the quality of resolutions is acceptable.

1.2 Organization

This thesis is organized as follows. In the next chapter, we give an overview of feature interactions and the Feature Interaction Problems using a motivating scenario. We also present the preliminary architecture and prior extensions devised to adapt the approach to dynamic systems. In Chapter 3, we introduce the simulator tool we used and the simulations that we built. In Chapter 4, we introduce our own extensions to the extended architecture and approach that aim to provide smoother sequences of outputs. In Chapter 5, we present the results of our case study. Resolution strategies for resolution modules are also introduced in this chapter. Chapter 6 reviews the related work and Chapter 7 concludes the thesis.

Chapter 2

Preliminaries

Throughout this thesis, we use examples from the automotive domain. Every feature extends a Basic Driving Service (BDS), which serves as a base system whose functionality is entirely self-contained [8].

2.1 Motivation

Consider the following scenario depicted in Figure 2.1. A semi-autonomous yellow car, whose features are listed in Table 5.1, drives on a highway. The features listed in Table 5.1 are implemented as distinct modules, to be plugged into a software architecture that supports plugin-able features.

The scenario is divided into three phases:

1. The yellow car drives along a straight road, with an initial speed of 75km/h. Meanwhile, the driver sets Cruise Control (CC) to the cruising speed of 100km/h. The speed limit is 90km/h, which means that the Speed Limit Control (SLC) feature should guarantee that the car speed does not exceed 90km/h.

2. A green car with speed of 80km/h cuts in front of the yellow car. At the same time, a blue car driving 80km/h is in the right hand blind spot of the yellow car. Because of the Headway Control (HC) feature, the yellow car will slow down to 80km/h, and then keep within a safe distance (30 meters) behind the green car. The yellow car then invokes the Lane Changing Control (LCC) feature to change into the right lane.

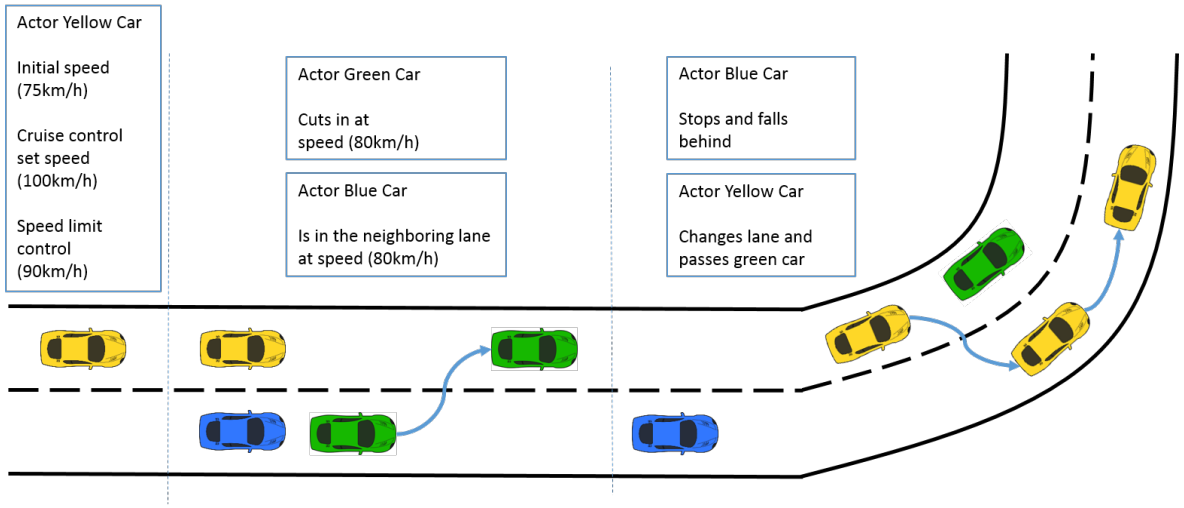

Figure 2.1: Motivating scenario

However, because the blue car is in its right-hand blind spot, LCC will not initiate the lane-change action.

3. The blue car falls behind as the yellow and green cars both go into a left curve in the road. The Lane Keeping Assist (LKA) feature will help the semi-autonomous yellow car to keep in the center of its lane. The driver of the yellow car again signals to change to the right lane in order to pass the green car. This time, LCC will successfully move the car to the right lane, and the car's speed will increase to 90km/h (the speed limit) because there is no car ahead of it after changing lanes. The yellow car passes the green car on the right.

2.1.1 Conflict Interactions

Before we start, we first define a few basics. The **system** is the software controlling a vehicle. A system's behaviour is expressed in terms of its reactions to changes and conditions in its environment - specially, as actions on environmental variables. **Monitored variables** such as the vehicle's speed represent phenomena that are sensed by or act as inputs to the system. Changes to monitored variables prompt reactions in the system behaviour. **Controlled variables** represent phenomena that are controlled or affected by system outputs [5]. The system has input devices (**sensors**) by which the system senses variables. The sensed values are recorded in system **input variables**. The system performs actions on

output variables, and these actions are realized on controlled variable by output devices (**actuators**). A feature's view of the system's input and output variables are called the **input** and **output variables** of that feature.

For example, consider the features Headway Control (HC) and Speed Limit Control (SLC). Suppose that a car with both features is traveling faster than the road's speed limit and there is a slower vehicle in front of it. Each feature will issue a command to slow the vehicle down. However, the two features' commands can be different, leading to a conflict interaction on the actuator that decelerates the vehicle.

In the example above, we used two features to explain conflict interactions. However, in a large-scale system, feature interactions can involve multiple features issuing commands and values to the same output variables, which may cause the situation to be a lot more complicated. A **conflict interaction** occurs when the actions of a set of features simultaneously modify the same controlled variable. According to Juarez et al.'s [21] two features are in conflict if they attempt to modify the same controlled variable by assigning different values to the same actuator or to distinct actuators that affect the same controlled variable.

2.1.2 Feature Interaction Problem

In the motivating scenario described in Section 2.1, there are several conflict interactions that need to be solved. We describe these feature interactions in Table 2.1. Feature interactions can occur in an automobile system at any time and in different circumstances. Additionally, feature interactions can involve different numbers of features and different types of features. As such, feature interactions can be considerably complex and can require a great amount of work for developers to resolve.

During system integration, software developers are expected to detect all interactions among combinations of features and provide an acceptable resolution for each conflict (possibly multiple resolutions for a conflict, if different resolutions are appropriate in different contexts, such as the presence or absence of other features). Then each resolution must be implemented and tested. Unfortunately, each addition of a new feature or each modification of an existing one requires the whole process be repeated for all combinations that involve the new/modified feature. As the number of features increases, the complexity of examining feature combinations and fixing undesired interactions increases exponentially, such that the workload of resolving interactions comes to dominate feature development.

The goal of our work is to devise default resolutions for feature interactions, which can greatly reduce the number of resolutions that developers need to design, implement, and test.

Phase	Involved features in feature interaction	Description
Phase 1	CC and SLC	When the vehicle's speed is 90 km/h, CC will try to accelerate the vehicle to 100 km/h, whereas SLC will try to keep the vehicle's speed at or below 90 km/h. These two features try to assign different values to the vehicle's speed.
Phase 2	HC, SLC and CC	HC will try to keep the vehicle's speed at or below 80 km/h (the speed of the car in front of it) and to keep a safe distance; whereas CC will try to accelerate the vehicle to 100 km/h; and SLC will try to keep the vehicle's speed at or below 90 km/h. All three features try to modify the vehicle's speed.
Phase 2	LCC and LCA	When the driver issues a lane-change signal, Lane Changing Control (LCC) will try to move the vehicle to the next lane, but Lane Changing Assist (LCA) will detect the blue car in the vehicle's blind spot and will keep the vehicle from moving to that lane. The two features try to assign different values to the vehicle's steering.
Phase 3	LKA, LCC, and LDW	When the driver issues a lane-change signal, LCC performs the lane-change action. As the vehicle is changing lanes, Lane Departure Warning (LDW) issues a warning that the vehicle is leaving the lane. Meanwhile, LKA will try to modify the steering angle to keep the vehicle in the centre of the lane. LKA and LCC features interact directly with respect to the vehicle's steering angle. Indirect feature interactions also exist between LDW and LCC because LCC responds to the driver's command to change lanes and LDW assumes that the driver wants to be warned when the vehicle leaves its current lane.

Table 2.1: Feature Interaction Description

Note that one cannot provide a priority among features in the resolution that would work in all scenarios. For example, when SLC's target speed is 90 km/h and HC's target speed is 80 km/h, we may choose HC's target speed. However, when SLC's target speed is 80 km/h and HC's target speed is 90 km/h, we may choose SLC's target speed. Therefore, it is not possible to predefine a priority.

2.2 Preliminary Architectural Model: Variable-specific Resolution

This thesis addresses feature interactions that are due to conflicts between features' actions on the system's outputs. We assume that the actions of each feature are assignments to output variables. Different features may assign different values to the same variable which causes conflicts. "Variable-specific" resolution was recently introduced by Bocovich and Atlee [5] for feature interactions due to conflict. In their work depicted in Figure 2.2, all features' actions that apply to a particular output variable are input to a resolution module designed for that variable. The resolution module is programmed to produce a conflict-free action that is based on the features' actions and on an appropriate resolution strategy for that variable. For example, for speed, it is reasonable to choose the minimum of the outputs from features since smaller speed is usually "safer"; and as for warning, issuing all the features' warnings is a rational resolution strategy since all the warnings are related to safety in different aspects.

Variable-specific resolution promises to have the following properties. First, it maintains feature modularity. The advantage of feature modularity is reduction of complexity - each feature can be considered and developed in isolation. Second, variable-specific resolution strategies can be implemented for each variable for which there can be conflicting actions, such that the set of action sequences output by variable-specific resolution are proven to be conflict-free [5]. Third, the resolutions are agnostic to the number and to the specific features involved in the interaction. Therefore new features can simply be "plugged in" to the preliminary architecture without extensive modification to the resolution modules. This enables rapid integration of new features and features developed by third-party manufacturers.

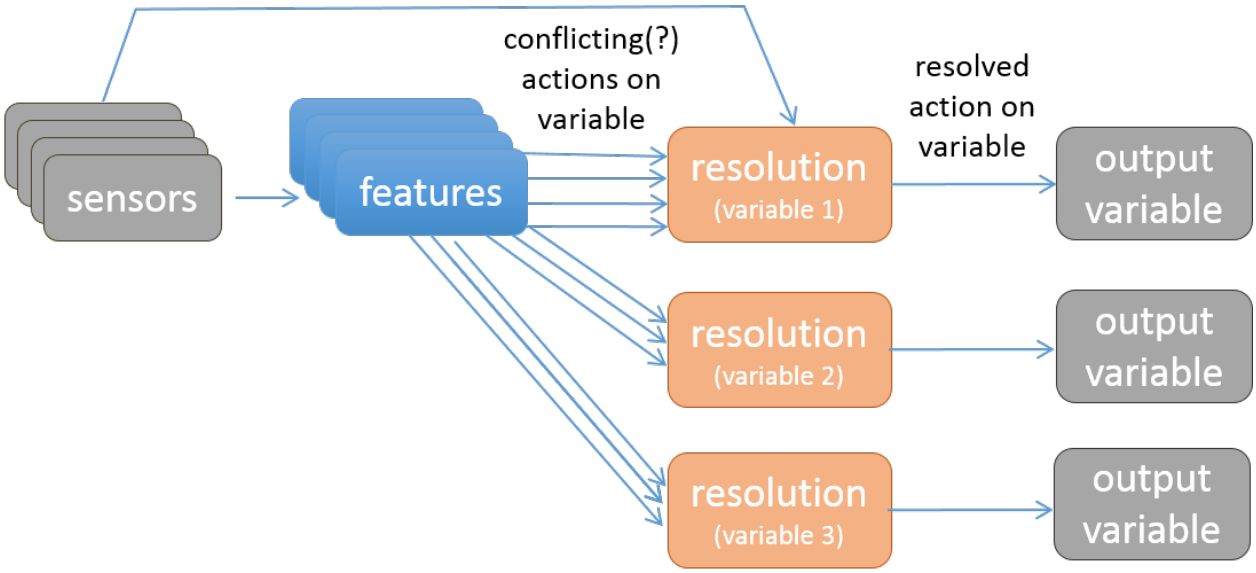

Figure 2.2: Architectural model for variable-specific resolution

2.3 Extended Architectural Model

The original work focused on resolution of conflicting actions in a single execution step. Zibaeenejad investigated extensions to the preliminary architecture to resolve conflicting interactions over a sequence of execution steps. His investigation led to three proposed extensions to the preliminary architecture:

1. Separation of feature logic vs. control logic.
2. Attribute features vs. actuator features.
3. Coupled variables.

These extensions to the preliminary architecture are discussed in the following subsections.

2.3.1 Architecture Extension 1: Separate Feature Logic from Control Logic

One of the biggest issues with the application of variable-specific resolution, especially to features in a dynamic system, is that many of the features' outputs are subject to real-time

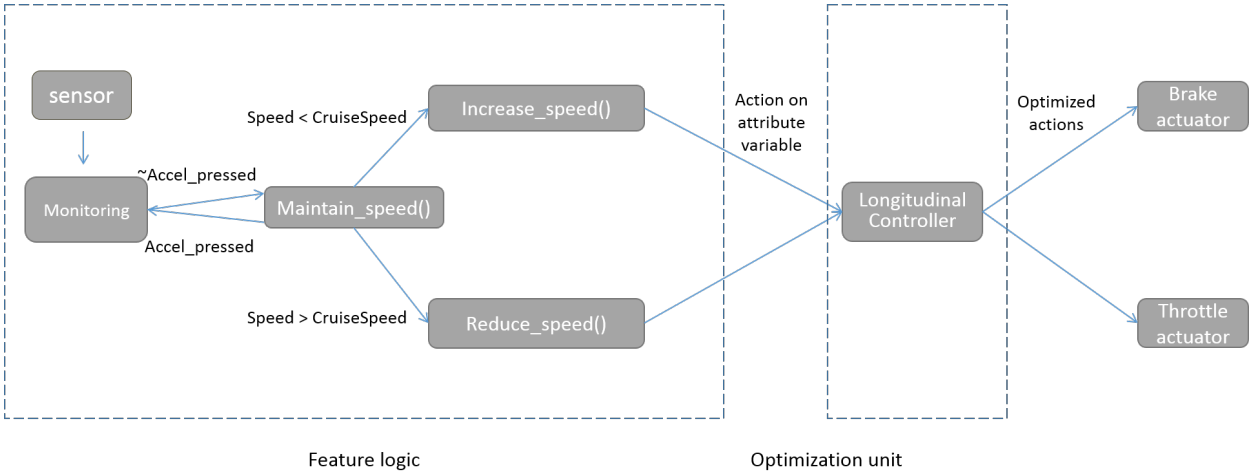

Figure 2.3: A high-level view of Cruise Control feature decomposed into feature logic and control logic

optimizations. Consider the Cruise Control feature as modelled in Figure 2.3. The feature logic calculates a target vehicle speed and aims to accelerate or decelerate vehicle so that the vehicle's speed matches the feature's cruising speed. The longitudinal controller takes the target speed and optimizes it with respect to the vehicle's state and the gradient of the road. The optimized value is sent to the appropriate actuators for realization: the throttle for positive acceleration and the braking system for negative acceleration. It makes no sense for each feature to individually optimize its outputs, and then try to resolve conflicts among multiple optimized actions on the same variable. Instead, the first proposed extension is to extract the optimization of outputs out of features. **Attribute variables** are introduced to represent abstractions of features' output variables, and feature actions are instead expressed in terms of these variables. Thus, features are decomposed into feature logic (which is responsible for calculating the desired value of an attribute variable in a given time period) and control logic (which is responsible for optimizing actions on actuators, taking into account system dynamics).

Once the feature logic is separated from the control logic there is an opportunity to resolve feature interactions at the feature-logic level, and then perform optimizations on the resolved actions. Therefore, in the proposed extended architecture (see Figure 2.4), feature modules represent the feature logic of features, outputting actions on attribute values; there is a unique resolution module for actions on each attribute variable; and there is a unique control-logic module for each attribute variable that realizes the (resolved) actions on that variable as optimized control signals to the appropriate actuators. Thus, a feature

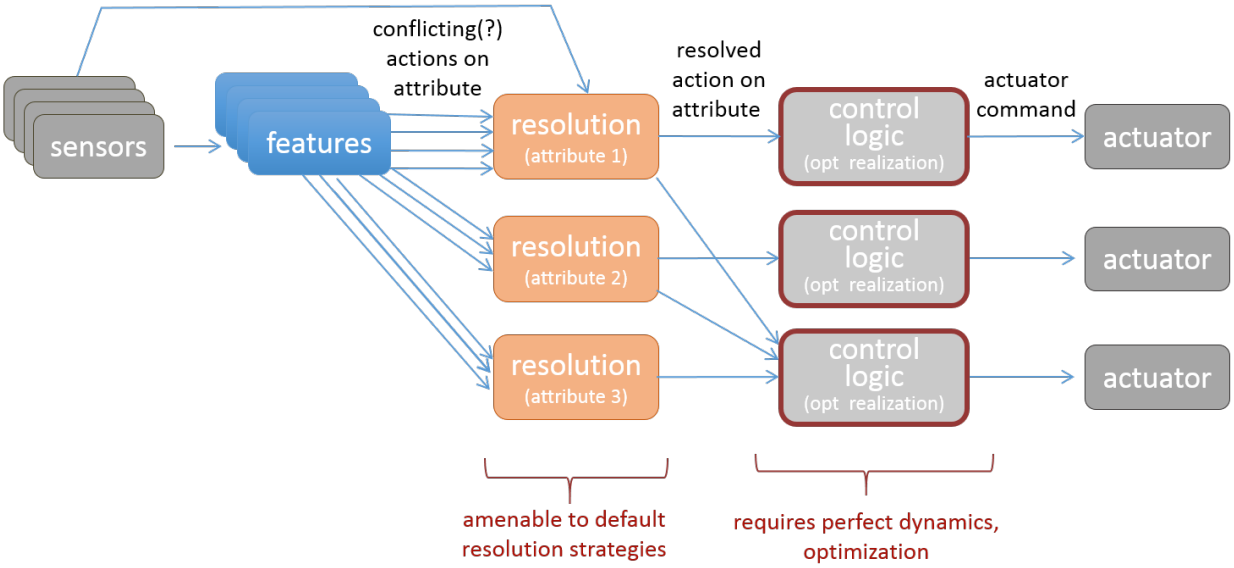

Figure 2.4: Architectural model extension # 1 - separate feature logic from control logic

developer is only asked to develop the 'feature logic' part of a new feature, which is then plugged into the extended architecture.

2.3.2 Architecture Extension 2: Actuator Features

Once control logic is separated out of features, the feature modules are actuator agnostic. That is, feature modules act on attribute variables, resolution modules implement variable-specific resolutions on conflicting actions, and the control logic modules optimize and realize the actions using specific actuators.

This extended architecture is applicable if all features are **function features** whose purpose is to add value or functionality to the overall system (in this case, the vehicle). But some features are created for the purpose of adding value and functionality to specific actuators. Examples of such features are Anti-lock Braking System (ABS) and Traction Control System (TCS). ABS allows the wheels on a motor vehicle to maintain tractive contact with the road surface according to driver inputs while braking, preventing the wheels from locking up (ceasing rotation) and avoiding uncontrolled skidding. On the other hand, when TCS detects one or more driven wheels spinning significantly faster than another, it invokes the ABS electronic control unit to apply brake friction to the wheels that are spinning with lessened traction. Because these features enhance the functionality

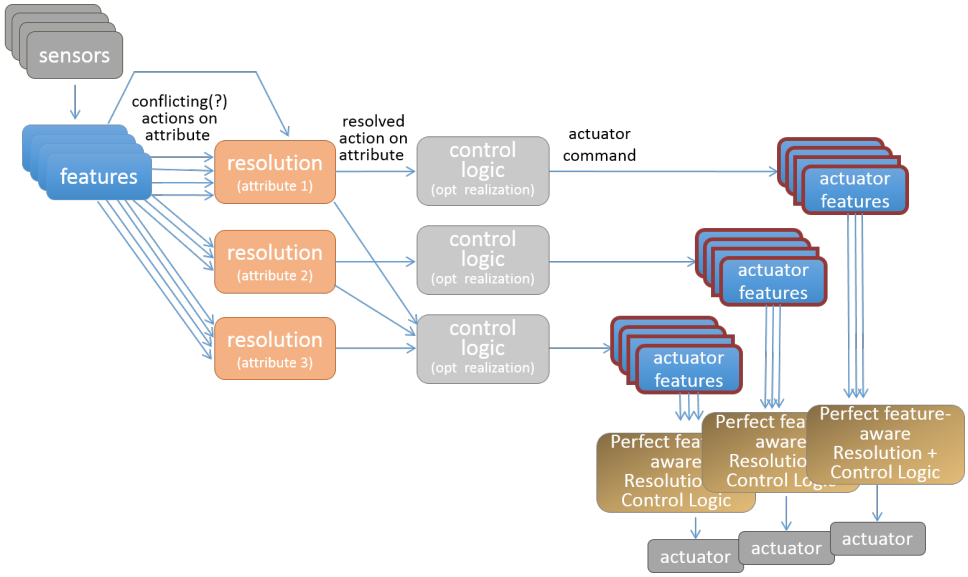

Figure 2.5: Architectural model extension # 2 - actuator features

of actuators, they are called **actuator features**. Actuator features enhance the resolved and pre-optimized actions that originate from the features and RMs. Actuator features modify these actions and then optimize them according to their own goals. One of the problems is that if there are multiple actuator features that operate on the same actuator, their actions may conflict. Moreover, because the actuator features operate on optimized actions and may be further optimized according to different goals, it seems unlikely that actuator features will be amenable to default resolution strategies. They require engineer-crafted resolution that takes into account the current vehicle state, its environment, and the set of actuator features. But actions on attributes are amenable to default resolution and they remain our focus.

The result of this second extension to the extended architecture is shown in Figure 2.5. Function features take data from sensors and specify actions on attribute variables and output these actions to the variables' respective resolution modules. The resolution module for each attribute variable then applies a variable-specific default resolution strategy on all features' actions on that variable, to produce a resolved action on the attribute variable. The control logic will realize such action as optimized signals to the appropriate actuators. These optimized signals will be sent to actuator features. An engineer-crafted resolution module that is knowledgeable of the actuator features integrates the features' actions into an ideal, feature-specific resolved action and output to actuators.

2.3.3 Architecture Extension 3: Coupled Attribute Variables

The preliminary architecture addresses features' conflicts over the values of individual attribute variables through variable-specific resolution modules. However, some attribute values are related to or depend on other attributes. Consider two features that act on different attributes. Therefore is no direct conflict between these features but there may be on implied conflict; such as if changes to one attribute variable affects the value of the other variable, or if the default resolution of one variable benefits from knowing the planned value of the other variable. We call these attribute variables **coupled**. For example, the attribute speed is related to the attribute steering, with respect to keeping the vehicle stable if the steering angle is significant, then the speed should not exceed a certain threshold, otherwise the lateral acceleration could be so large that the vehicle may turnover. There are two possible approaches to resolving conflicts over coupled attribute variables: resolve them together in a single resolution module that produces resolved outputs for both variables, or if possible, keep the resolution modules separate but allow for some information flow between them. This thesis explores the latter approach.

The result of this third extension to the extended architecture is shown in Figure 2.6. Directional communication paths among the resolution modules are established, such that the outputs of one resolution module (i.e., resolved actions on one attribute variable) to be communicated as inputs to the resolution modules of coupled attribute variables. For instance, after the resolution module for steering decides on the value of steering angle, it sends a next step value to speed resolution module, and the speed resolution module will use this steering angle to calculate an upper bound of the speed.

Note that the communication flows among resolution modules must be acyclic so that there are no circular dependencies.

2.3.4 Example of Extended Architecture

An example of the extended architecture for variable-specific resolutions, applied to features that manipulate speed, is shown in Figure 2.7.

The example uses speed as a representative attribute variable to demonstrate the extended architecture. Speed-related function features take data from sensors and calculate (possibly conflicting) actions on speed, and then send them to the speed resolution module. The speed resolution module takes the actions and data from sensors as input, and it outputs a resolved action on acceleration that is forwarded to the control-logic modules for the throttle and the brake. The throttle's control-logic module sends a control signal

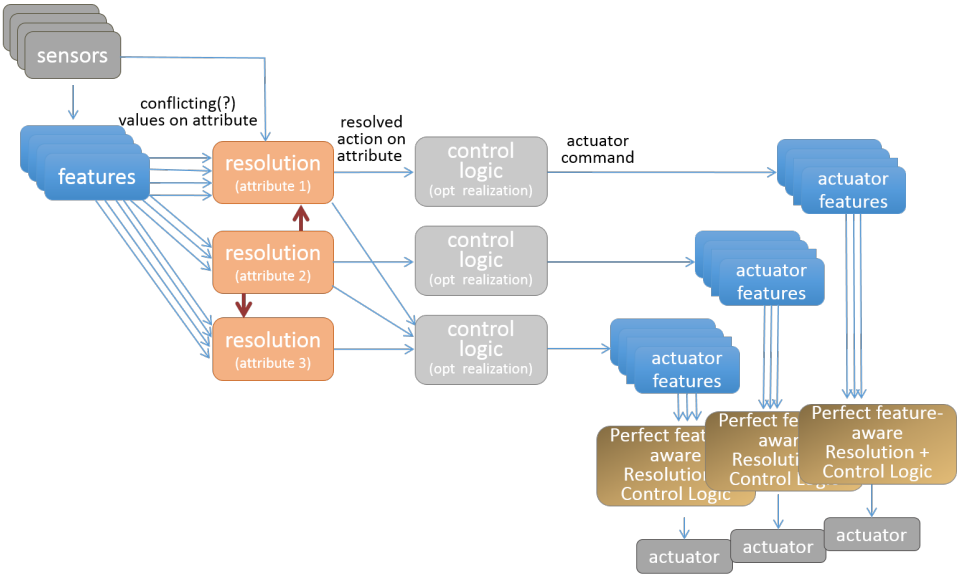

Figure 2.6: Architectural model extension # 3 - coupled attribute variables

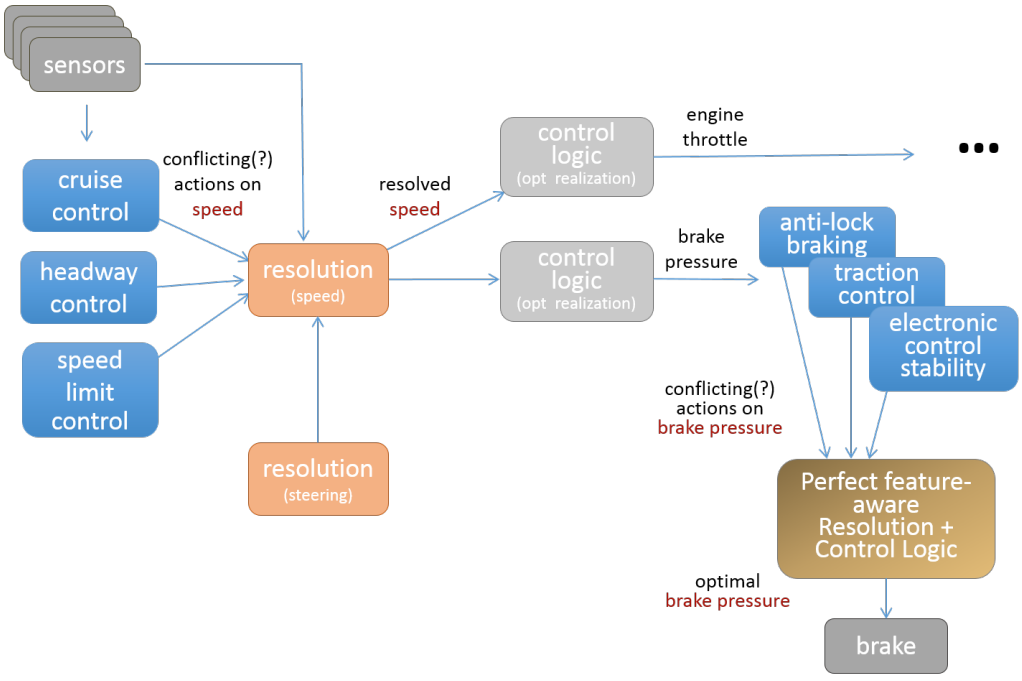

Figure 2.7: Example of extended architecture for variable-specific resolution

to the throttle (assuming no actuator feature for the throttle). However, the process of sending a signal to the brake is more complex than for throttle because there are several actuator features that enhance the functionality of the brake in our example. These actuator features will operate on the resolved action and output (possibly conflicting) actions on brake pressure to another control logic. An engineer-crafted resolution module inputs these actions and determines ideal resolutions for all interactions in all circumstances, and the control logic will compute on optimal brake pressure signal and send it to the brake.

Chapter 3

Simulation of Variable-Specific Resolution of Feature Interactions

Given the proposed extended architecture that aims to resolve interactions at runtime, one of the goals of this thesis is to evaluate how well it performs in practice (where a simulation environment is used to demonstrate practice). Throughout this thesis, we focus on runtime resolutions of feature interactions in the automotive domain. Thus, we embedded the extended architecture into an automotive simulator named PreScan. We also implemented features as distinct modules within the simulator, and we implemented resolution modules that take the features' actions as input and use our resolution strategies to resolve conflict interactions and give outputs to actuators, especially brake, throttle, steering. With this setup, we could evaluate the features, resolution modules, and the extended architecture with respect to how well it resolves interactions.

3.1 PreScan Simulator

Our work is embedded in a simulation tool named PreScan.[1] PreScan is a physics-based simulation platform that is used in the automotive industry for the development of Advanced Driver Assistance Systems (ADAS). ADAS is based on sensor technologies such as radar, laser/lidar, camera and GPS. PreScan is also used for designing and evaluating vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communications as well as autonomous-driving applications. We can also build and modify traffic scenarios using a

[1]From PreScan Official Website https://www.tassinternational.com/prescan

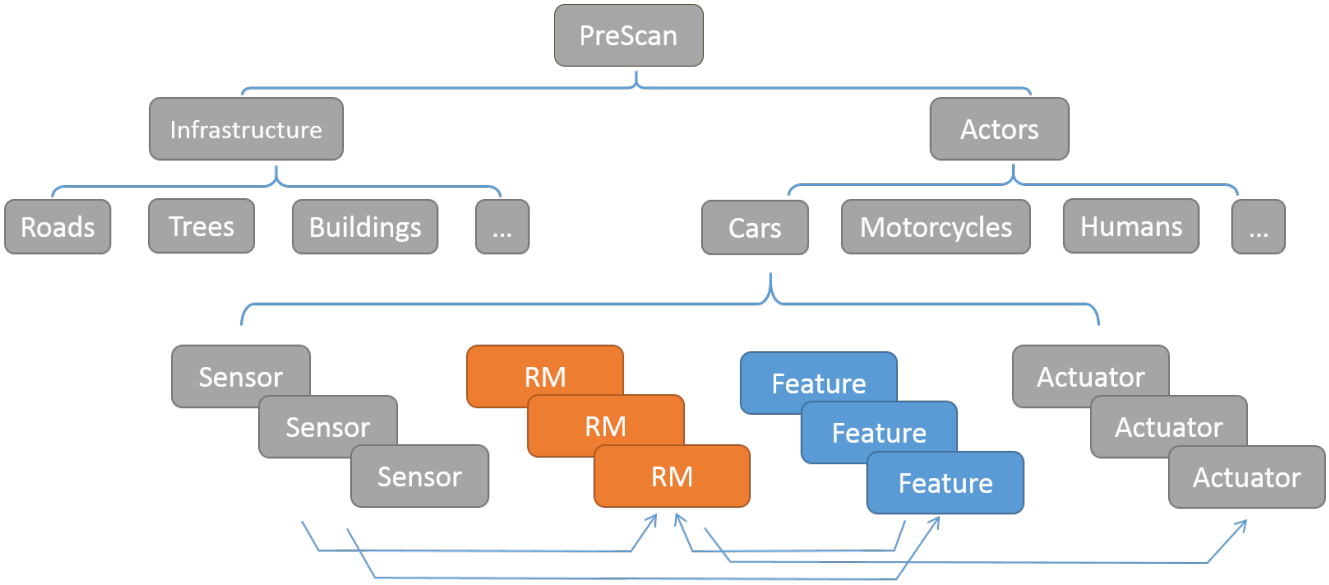

Figure 3.1: Implementation Structure in PreScan

database of road sections, infrastructure components (trees, buildings, and traffic signs), actors (cars, trucks, bikes, and pedestrians), weather conditions (rain, snow, and fog) and light sources (sun, headlights, and lampposts). PreScan can simulate different types of vehicle models, which are called actors. Actors are programmable automobiles that can be equipped with different sensor types.

PreScan is built to work with MATLAB Simulink, therefore the implementations of architectures, feature modules and resolution modules are realized in the MATLAB language as Simulink blocks. To program the actor (i.e., the feature-laden vehicle), we use the MATLAB/Simulink interface to add a **control system**. A control system is the software system running in the vehicle, which controls the vehicle's behaviour. This is where we program our architecture, features, and resolutions. The overall structure of the PreScan modules is shown in Figure 3.1.

Figure 3.2 shows a typical flow of implementing a simulation and running an experiment in PreScan. Details about each implementation step are introduced in the following sections.

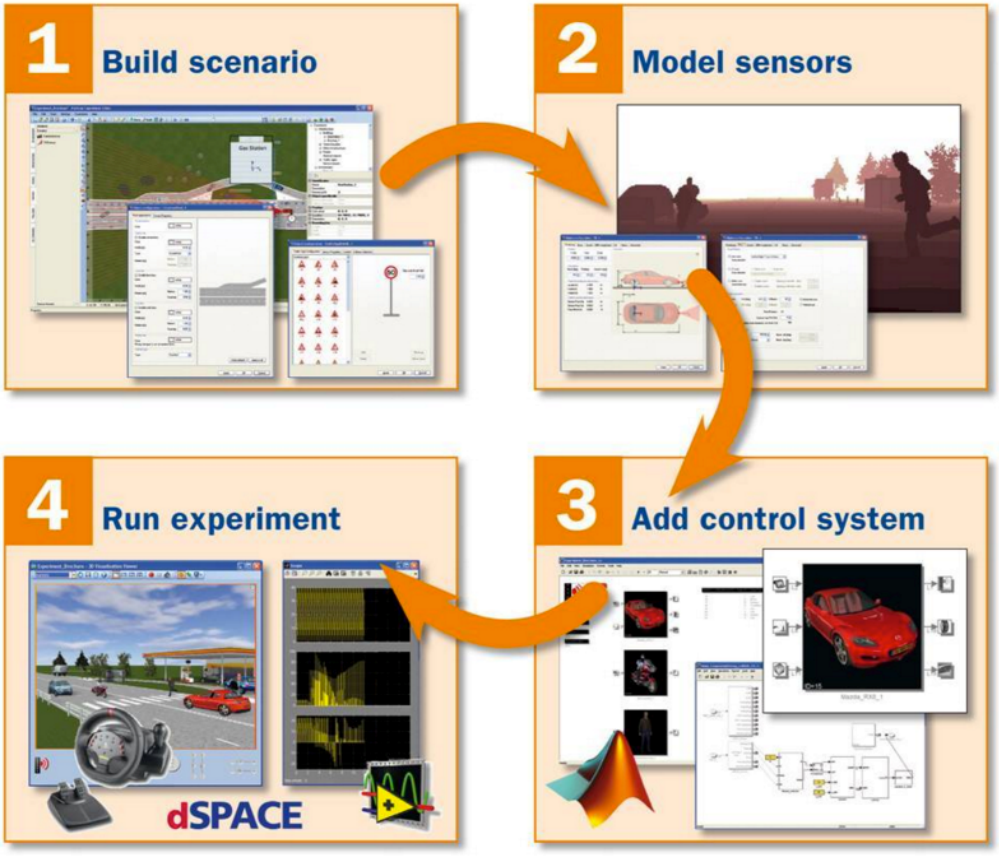

Figure 3.2: Flow of Running Experiment on Test Bed

Infrastructure	Road Segments, Road Markings, Buildings, Abstract Objects, Traffic Signs, Animated Elements (Highway Overhang and so on), Reflectors & General Content(Walls and so on).
Actors	Cars, Motors, Trucks, Busses, Trailers, Humans, Calibration Elements (Box, Sphere, and so on, used to represent obstacles).

Table 3.1: Environment Components

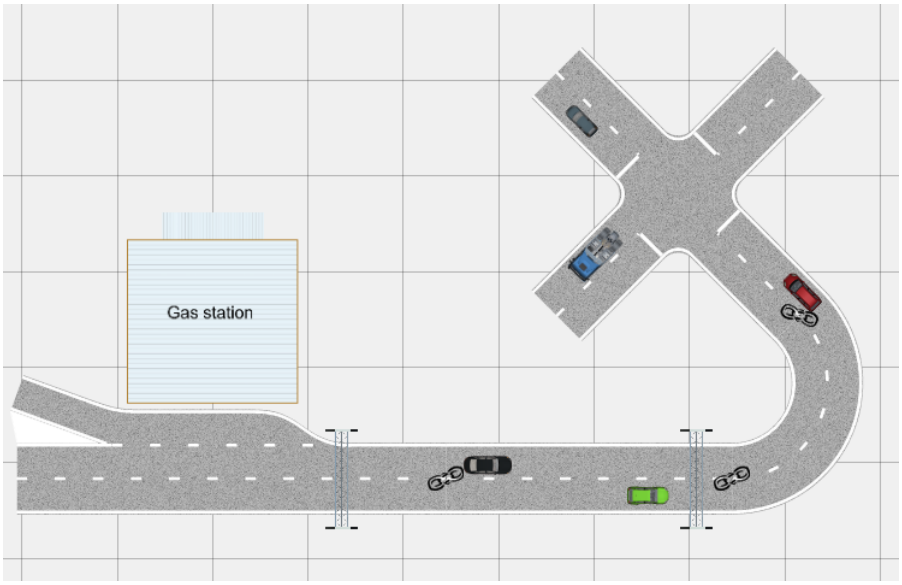

Figure 3.3: Example of Building Scenario

3.2 Build Scenario

First, we build a virtual world as our test foundation. The virtual world includes two basic components: the infrastructure and the actor. Test scenarios for the actor can be changed by modifying infrastructure parameters (e.g., changing the length or the curvature of a road). All the components we used are listed in Table 3.1.

Using the PreScan GUI, we could add and modify the infrastructure as one could design a city blue print. An example of a scenario that is under construction is shown in Figure 3.3. Note that although most parameters of infrastructures can be changed, there are some of them that cannot be modified (such as we cannot change the gradient of the road), which result in some limitations in modelling environmental conditions.

3.3 Build Actor

An actor comprises 5 types of modules: Sensors/Settings, Feature Modules, Resolution Modules, Control Logic, and Actuators.

3.3.1 Model Sensors

Each actor can be fitted with multiple kinds of sensors (such as Camera, Lidar, Radar, TIS, and so on) to obtain information needed from the environment. Different sensors have different uses. For example, cameras and lane-marker sensors are used to detect lane markings. Information about lane markings are used to calculate the distance between the vehicle and the edges of the current lane, which in turn are used to implement steering-related features such as Lane Keeping Assistant. Radar and TIS are used to detect the distance to objects. Sensors can also provide the speed of detected objects, and the information can be used to implement speed-related features such as Headway Control.

Sensors are modifiable. We can change sensors' parameters (heading, range, frequency, and so on) in order to accommodate different requirements from various features. For example, the LCA needs to know whether there is a car in the vehicle's blind spot. To detect this, there must be a sensor that is directed perpendicular to a vehicle's front. In contrast, HC needs to know the preceding car's speed and its distance in front of the vehicle; therefore, there must be a sensor that is headed in the same direction that the vehicle is travelling. Additionally, considering that the vehicle travels on different shapes of roads and that the preceding car may be in different positions in front of the vehicle, the sensor must be able to adjust its beam range and degree.

An example of an actor and its sensors is shown in Figure 3.4. The Ultrasonic sensor is used to detect whether there is an object in the vehicle's left-hand blind spot, and the CameraSensor is used to detect the lane mark and to get data used to compute the distance between the vehicle and the lane marks. We can see that different sensors have different parameters according to the strength and direction of their respective beams.

3.3.2 Control System

The control system is the part of simulation that is programmable rather than just configurable. Within the control system, each feature and resolution module are represented as a distinct module. Our feature modules use information data from the sensors as input. After feature logic calculations, features will give outputs as commands or actions on

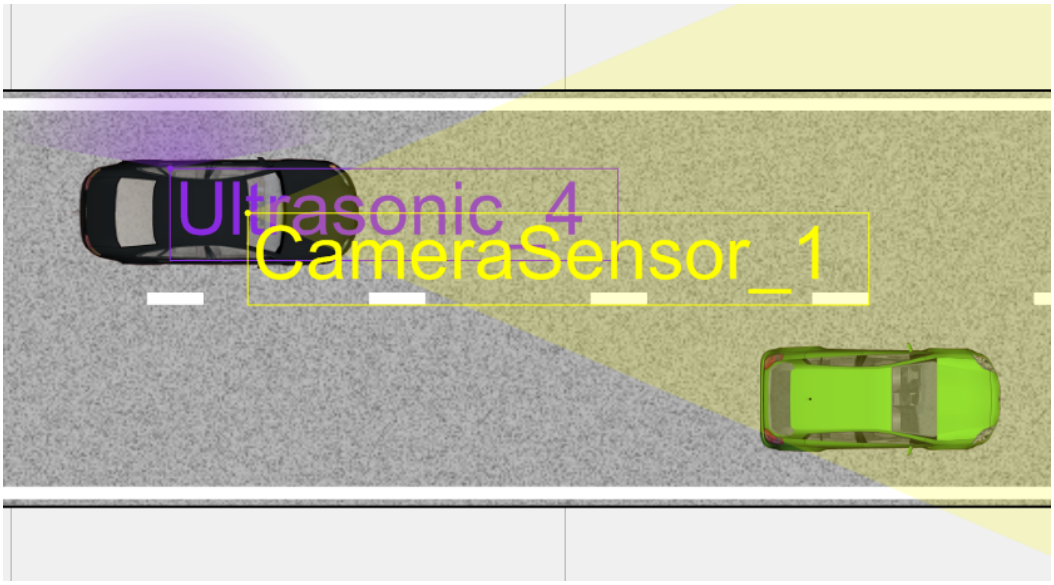

Figure 3.4: Example of the Actor's Sensors

output variables. Each of the features' actions is input to the resolution module associated with the action's target variable. Then the resolution module will resolve conflicting assignments to the same variable and output a resolved value to the target variable.

An example of a control system for steering features is shown in Figure 3.5. We can follow the arrows to trace the data flows. Sensors provide data to the features. The features output their actions to the appropriate resolution modules. The actions are also output to the simulator's viewer to facilitate our evaluation of the vehicle's behaviour. The viewer is introduced in Section 3.4. Then the resolution module outputs a resolved action to the appropriate control logic to translate the action into an optimized actuator signal, to send to the respective actuators.

3.3.3 Actuators and Control Logics

The control logics are generated by PreScan as separate modules, one for each actuator. For example, the control logic for speed will translate the desired acceleration into signals to be sent to the brake and throttle. Actuators are pre-built as a module in the control system, and are configured according to the desired dynamics. Different types of dynamics include different actuators. The dynamics type we chose is called "simple dynamics" because this dynamics type includes brake, throttle and steering actuators, which satisfies our research

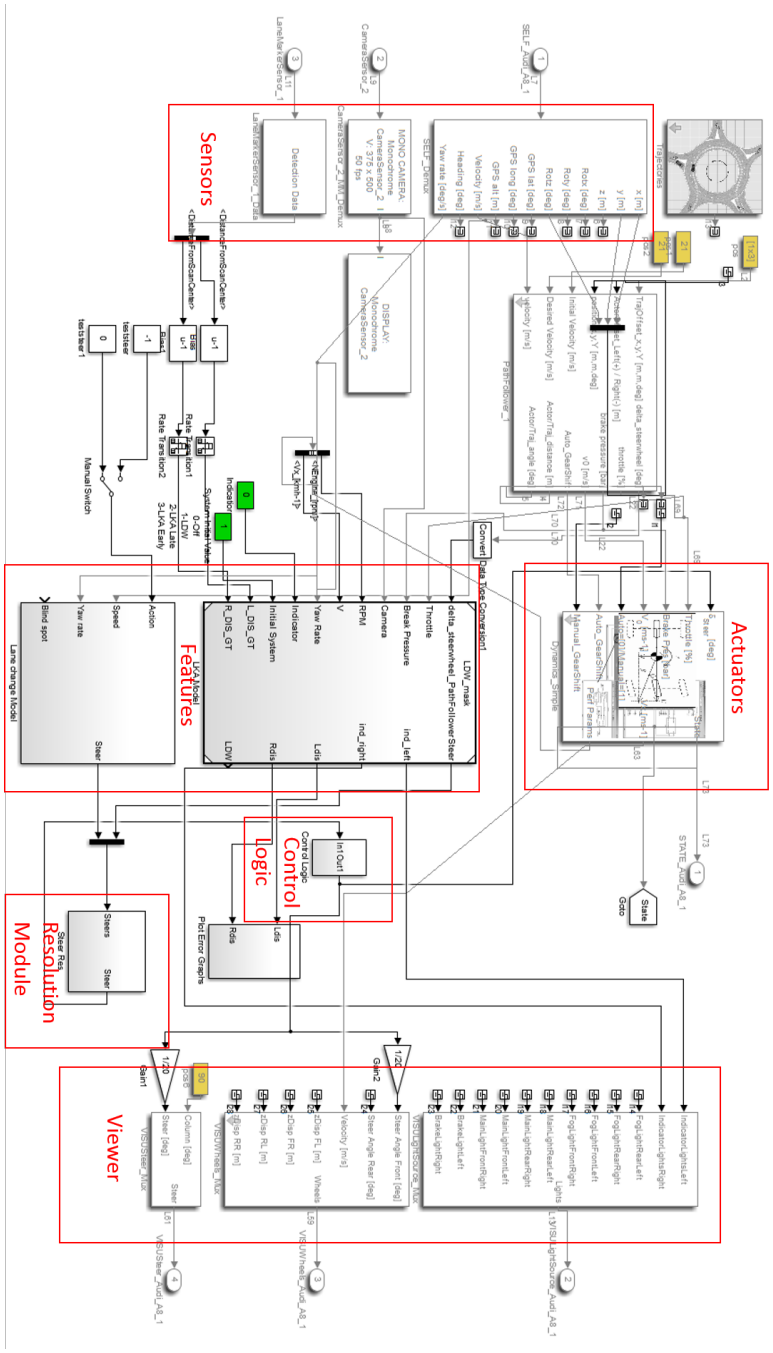

Figure 3.5: Example of Control System Implementation

needs. Because the actuators are pre-built into the actors, we could not directly manipulate an actuator's inner behaviour. This limitation means that we cannot simulate actuator features such as ABS and TCS.

3.4 Run Experiment

Once we finished all of the previous steps, we can run experiments. During simulations, we can monitor the vehicle's behaviour, and the values of controlled variables and output variables through the graphs of variable values and the viewer, and so on. Figure 3.6 shows several ways in which we could monitor the state of the system, and therefore perform our evaluations. Figure 3.6 (a) and (c) are viewers that provide 3D visualizations of the vehicle's behaviour. Figure 3.6 (b) is a graph showing the values of a specific variable over time. Any variables can be plotted over time, including attribute variables, sensor outputs, and actuator values. Figure 3.6 (d) shows a collection of data of the vehicle's status, which gives us accurate information for evaluating the system's behaviour during a simulation run.

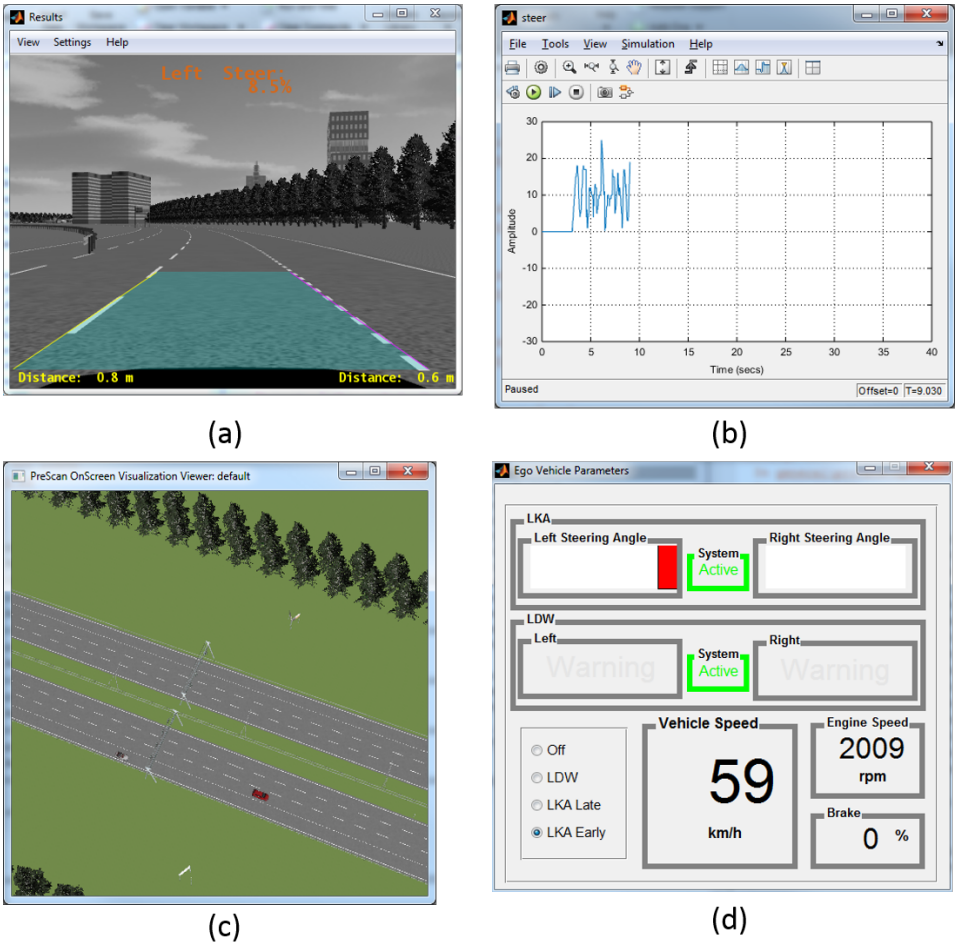

Figure 3.6: Example of Different Ways to Monitor the System during an Experiment

Chapter 4

Further Extensions to the Architectural Model

Once we understood how to embed the extended architecture in the PreScan simulator, we started to evaluate its effectiveness by running small experiments. Specifically, my research goal is to evaluate the effectiveness of the extended architecture in:

1. devising acceptable resolutions at runtime
2. devising acceptable resolutions over variables that are coupled

In particular, when evaluating "acceptable resolutions at runtime", we determined to solve a key research question:

- Can variable-specific resolution modules produce sequences of resolved outputs that are smooth (i.e. not jerky)?

This question is discussed in section 4.2.

4.1 Methodology

The methodology we used to evaluate the extended architecture is based on **iterative and incremental development**. The basic idea of this method is to develop a system

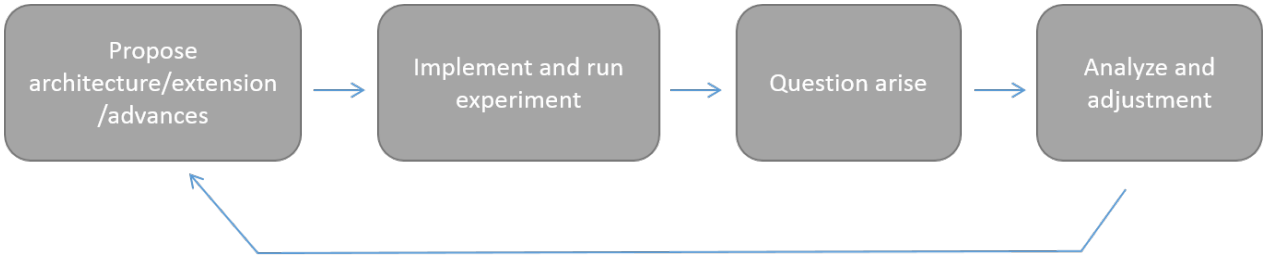

Figure 4.1: Iterative and incremental development for extended architecture

through repeated cycles (iterative) and in small portions at a time (incremental), thereby allowing us to take advantage of what is learned from the development of earlier parts or versions of the system. The procedure is shown in Figure 4.1. Specifically, we started with the extended architecture in Chapter 2.3, embedded it within PreScan, built test scenarios, and ran experiments. Questions arose from the performance of the features and the resolution modules in the experiments. We then analyzed the results and ran additional test scenarios with small variations to understand why the simulation performed the way it did. We determined what adjustments were needed to the features, resolution modules, or the architecture. Then we went back to the first step, and proposed extensions or advances to our previous architecture.

The concrete preliminary control system that we embedded in PreScan is shown in Figure 4.2. We used this initial control system as our **Functional Prototype** for the beginning of the iterative and incremental development. In the functional prototype, Sensors, Control Logic, and Actuators were configured and generated in PreScan (represented in the color gray). Features and Resolution Modules were implemented by us (represent in blue and orange).

For example, from the original prototype **(Propose architecture)**, when LCC is doing lane-change action, the LDW would always warn the driver the vehicle is leaving the lane **(Implement and run experiment)**. However, we actually did not want the warning since we are leaving the lane deliberately **(Question arise)**. So we added a communication flow from LCC to LDW that LCC will inform LDW when it is working so that LDW will not give out the warning **(Analyze and adjustment)**. We implemented the change and testified the increment that solved the question **(Increment)**. We continued in this manner until we got to the current architecture **(Iterative)**.

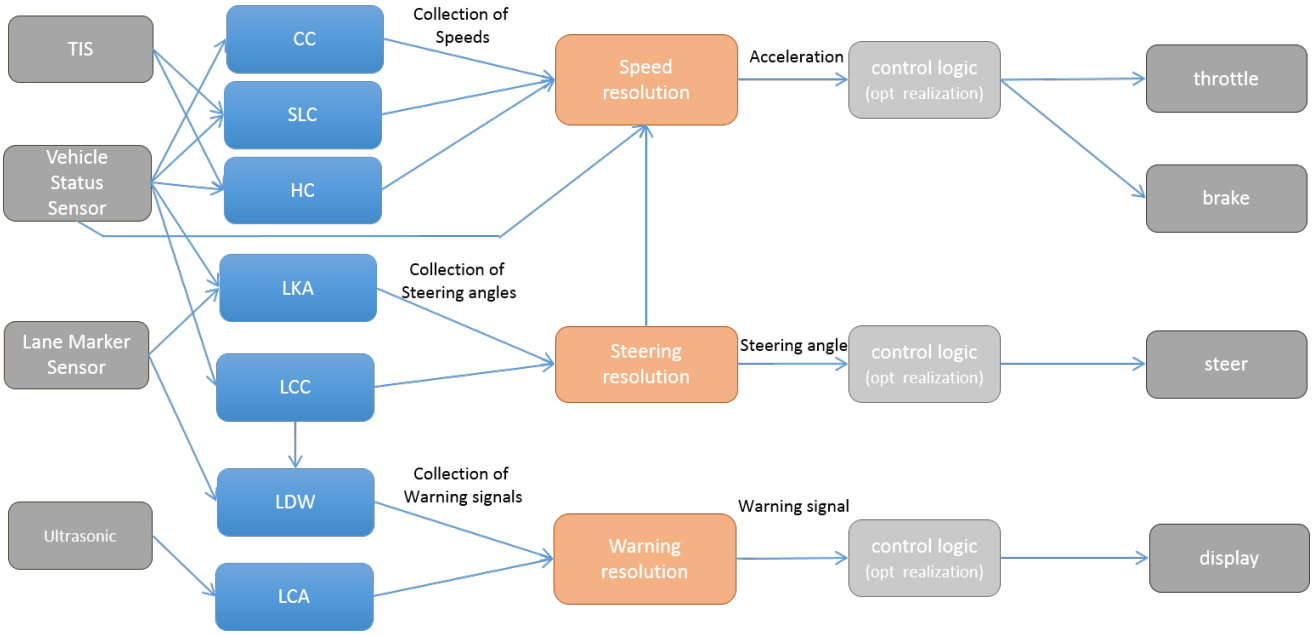

Figure 4.2: Example of overall extended architecture for resolution of feature interaction

4.2 Architecture Extension 4: Smooth Resolutions

One issue that we encountered in our experiments required an extension to the architecture to achieve smooth sequences of resolved actions from the resolution modules. Consider that features CC and SLC are competing around a certain threshold, when the vehicle's speed is slower than the threshold, CC will require the vehicle to speed up. When the vehicle's speed is faster than the threshold, SLC will require the vehicle to slow down. These two features assert different target values, and a resolution strategy that picks the minimum value makes the selection of a target value vary extremely from execution step to execution step. Individual decisions about features' actions make sense and are desired, but the resulting behaviour is not desired as it is not smooth.

Originally in the preliminary architecture [5], the feature action language was limited to assignments of values to attribute variables. Thus, the input to a resolution module was a possibly-conflicting set of target values for an attribute variable. Unfortunately, this single type of input does not provide enough information about features' intentions to formulate correct and smooth resolution decisions. In our research, we found that two more types of actions on attributes would not only provide a richer description of the features'

requirements on an attribute's next value, but also give a RM more information about past and expected future actions, so that the RM can produce resolved actions that result in smoother sequences of outputs. Those two input types, **derivatives** of attributes and **bounds/constraints** on attribute values, are discussed in the following subsections.

4.2.1 Derivative of Attributes

The reason that we include derivatives of attributes in the features' action language is that in some circumstances two features may aim to reach the same target value of an attribute variable, but within different time frames. For instance, assume that the vehicle is cruising at a speed of 60 km/h and enters a zone where the speed limit is 50 km/h. At the same time, the vehicle is travelling too close to a car that is driving 50 km/h in front of it. In this scenario, both SLC and HC try to set the target speed as 50 km/h. However, HC requires that this target be reached quicker to avoid a collision, which means they have different requirements on the vehicle's rate of acceleration. Therefore, the resolution strategy should take into account the features' desired rates of change of the attribute instead of only the features' desired target value.

To remedy this issue, we extend the feature-logic action language to allow feature-logic modules to send the rate of change of variables to the resolution module, and the resolution decisions are based on rates of change of variables as well as target value. For example, instead of sending only a target speed to speed resolution module, a speed feature can send a target acceleration along with target speed.

4.2.2 Attribute Bounds/Constraints

Before we refined the extended architecture to allow attribute bounds as inputs to the resolution module, we observed that it was possible for our simulations to experience turbulence around a threshold. We discovered that the problem is that a feature issues an action only when it wants to change the vehicle's behaviour. After the feature achieves some desired behaviour, it would not issue any action it needs to re-achieve the desired behaviour again.

In regard the CC and SLC features: assume that the CC feature has a cruising speed that is above the speed limit of the road. When the vehicle exceeds the speed limit, SLC activates and the resolution module continuously selects the output of SLC to reduce the vehicle's speed to below the speed limit. After reducing the speed to below the speed limit, CC will try to increase the speed and the resolution module will issue this action until the

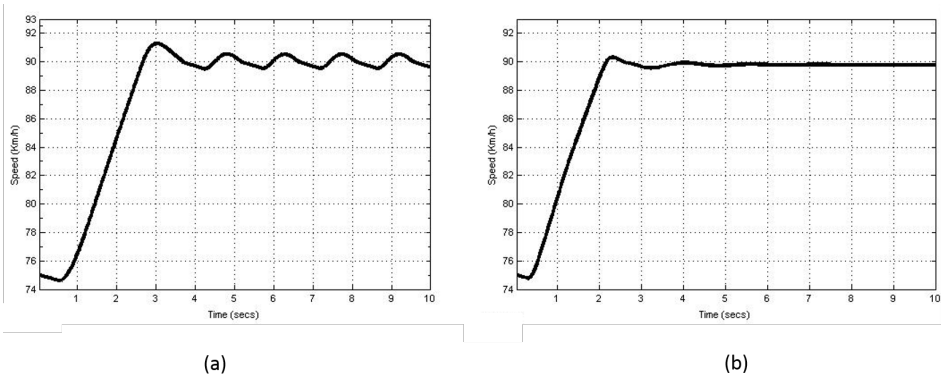

Figure 4.3: Improvement in the quality of resolutions due to the inclusion of constraints on attribute variable values in the resolution strategy (Transition Scenario 1)

vehicle exceeds the speed limit again. This pattern can be repeated, causing a periodic increase and decrease in the vehicle speed around the speed-limit threshold. We simulated this condition, and the fluctuations in vehicle speed are shown in Figure 4.3 (a).

To remedy this situation, we extend the feature-logic action language to allow a feature-logic module to specify that the value of an attribute variable remain within certain bounds/constraints. Then the resolution module guarantees that the resolved output does not violate those bounds/constraints. This calculation ensures that the future values of a variable, up to a certain prediction horizon (e.g., restrict the acceleration to ensure the speed will not violate bounds/constraints in the next 5 seconds), considers the bounds/constraints of features that will be triggered in future. In the case of the example presented, at each time instance, the resolution module will calculate the acceleration upper bound that the vehicle can have when accelerating to reach desired cruising speed; but once the speed limit is reached, the resolution module will assign target acceleration values that do not violate the speed-limit constraint.

We allow features (such as SLC) to continuously issue constraint goals (even if the goals are currently met). This way, a resolution module can consider those goals in their resolutions. CC, HC, SLC all have goals that they should issue in conjunction with the target values that they issue when their goals are violated. However,in this work we consider only constraint goals (attribute bounds). Examples of constraint goals are speed limit in SLC and the preceding car's speed in HC. Another kind of goal is a target attribute value, such as CC's desired cruising speed. Target-value goals are not considered in this thesis, we leave it to future research.

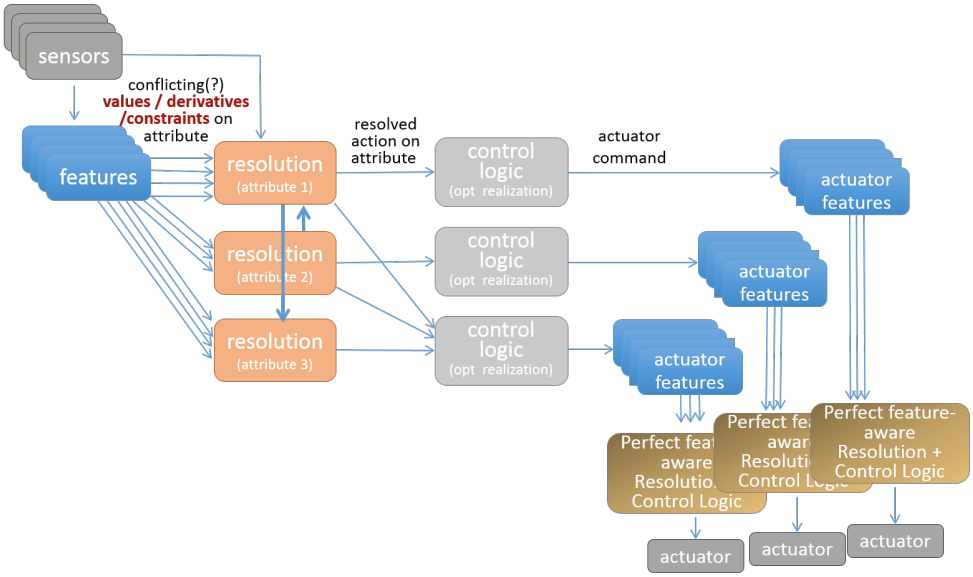

Figure 4.4: Architectural model extension # 4 - smooth resolution

This extension helps to void thrashing at the thresholds' values of attribute variables. We can see the improvement in the simulation of vehicle speed under the same scenario setting in Figure 4.3 (b).

The result of these extensions is our current architecture for programmable variable-specific feature interactions (see Figure 4.4.). Our next step is to evaluate the current architecture with a case study of features.

4.3 Architecture Extension 5: Task Features

We introduce the concept of a **task feature** that requires special attention in this section. The feature logic of a task feature is comprised of a series of actions. When a task feature starts its task, it has priority over other features. Note that these features need to have a roll back mechanism if they cannot complete the task. For example, Lane Change Control (LCC) is a task feature. The task of changing a lane starts upon driver's request, by checking the traffic in the target lane. When the transition to another lane starts, the output Lane Keeping Assistant (LKA) and Lane Departure Warning (LDW) features are ignored. A sudden change in the environment such as appearance of an obstacle in the target lane occurs, then feature stops the transition and return to the original lane.

Because of the feature logic's particularity of task features, these kinds of features need special treatment within the resolution strategy (introduced in 5.2.2).

Chapter 5

Case Study and Evaluation

In order to evaluate the current architecture presented in this paper, we conducted a case study involving automotive features by Bosch[1]. There were a total of 22 automotive features provided by Bosch. Because of the simulator limitation and constraints on implementation, only 15 features could be implemented and used in the case study. The features are listed in Table 5.1. The features labelled with a star were not implemented owing to the following reasons. For features Driver Drowsiness Detection (DDD) and Road Sign Assist (RSA) there are no appropriate sensors supported in the simulation tool. The feature Automatic Park Assist (APA) is too complicated to be implemented and has no conflict interactions with other features. Lastly, actuator features are not implemented, because of the simulation toolbox does not allow altering actuator behaviour (which is the goal of actuator features).

There are three attribute variables in the test bed: speed, steering, and warning signal. We have one resolution module for each of them respectively. These three attribute variables are the only ones acted on by the implemented features in the case study.

We implemented the features ourselves as Simulink modules, and tested each feature individually to ensure that it behaves as expected in isolation. After that, we plugged features into the simulation test bed and executed them in combinations.

Then we evaluated the effectiveness of the current architecture from two aspects:

1. The minimum evaluation criterion are met.

[1]From "Bosch Mobility Solutions" http://products.bosch-mobility-solutions.com/en/de/homepage/homepage_1.html

Feature Name	**Function**	**Acronym**	**Feature type**	**Feature logic output**
Speed Limit Control	Automatically keep the vehicle's speed at or below the road's speed limit.	SLC	Attribute (speed)	Speed bound
Cruise Control	Maintain a desired speed of vehicle as set by the driver.	CC	Attribute (speed)	Acceleration
Headway Control	Keep the vehicle within a safe distance behind the preceding car.	HC	Attribute (speed)	Speed bound & Acceleration
Traffic Jam Assist	When activated by the driver, the car assumes the same speed as the car in front of it, following at a safe distance.	TJA	Attribute (speed)	Acceleration
Predictive Pedestrian Protection Braking	Automatically initiate emergency braking in case of impending collision with a pedestrian.	PPPB	Attribute (speed)	Acceleration
Predictive Braking System	When warned of a collision, initiate partial braking to give more time to the driver to react.	PBS	Attribute (speed)	Acceleration
Emergency Braking System	When warned of a collision, prepare the braking system for full braking, to be activated with any braking signal from the driver.	EBS	Attribute (speed)	Acceleration
Lane Keeping Assist	Issue appropriate steering commands to keep the vehicle in the centre of the lane.	LKA	Attribute (steering)	Steering angle
Lane Changing Control	Automatically finish a lane-change action.	LCC	Attribute (steering)	Steering angle
Lane Departure Warning	Detect if the vehicle is about to move out of its lane, and warn the driver.	LDW	Attribute (warning signal)	Warning feedback
Collision Warning	Detect and warn the driver when the distance to the preceding car is critically short.	CW	Attribute (warning signal)	Warning feedback
Pedestrian Protection Warning	Detect an impending accident with pedestrians (who are in the same lane as the vehicle) and warn the driver of impending collision.	PPW	Attribute (warning signal)	Warning feedback
Lane Change Assist	Detect when a vehicle is in the driver's blind spot or is approaching rapidly from the rear, and warn the driver.	LCA	Attribute (warning signal)	Warning feedback
Rear Cross Traffic Alert	Detect vehicles crossing to the left or right behind the driver's vehicle, and warn the driver.	RCTA	Attribute (warning signal)	Warning feedback
Parking Aid	Detect the presence of an object when the vehicle is moving in reverse, and warn the driver indicating the distance to the detected objects.	PA	Attribute (warning signal)	Warning feedback
Automatic Park Assist*	Automatically complete a parallel or reverse parking.	APA	Attribute (speed & steering)	Acceleration & steering angle
Anti-lock Braking System*	Reduces the brake pressure to avoid wheel lock up.	ABS	Actuator	Wheel slip
Traction Control System*	Prevents the wheels from spinning by reducing the drive torque at the spinning wheels.	TCS	Actuator	Wheel slid
Active Steering System*	Alters driver's or other features' steering actions to limit yaw speed and rate to maintain vehicle stability.	ASS	Actuator	Wheel angle
Electronic Stability Program*	Detects if skidding is imminent and intervenes by applying braking power to individual wheels and/or reducing engine power in order to restore vehicle's stability.	ESP	Actuator	-
Driver Drowsiness Detection*	Detects driver fatigue and sends a warning based on analysis of steering angle data.	DDD	Attribute (warning signal)	Warning feedback
Road Sign Assist*	Detects and interprets the road signs and displays them to the driver.	RSA	Attribute (warning signal)	Information display

Table 5.1: Feature List

2. Global system properties are satisfied: Global system properties include feature requirements (such as the vehicle does not exceed the speed limit, does not violate safe distance, and so on) as well as expectations of the vehicle's behaviour (such as the vehicle stays on the road, avoids collisions, and so on).

5.1 Evaluation Criteria

For features in the system, there already exist general feature-independent criteria for evaluating the acceptability of vehicle behaviour. For example, [26] provides evaluation for Adaptive Cruise Control in a special scenario. However, to our knowledge, there is no commonly accepted criteria to assess the behaviour of a vehicle when features are combined and conflict interactions are resolved at runtime. Therefore, we had to define our own criteria to evaluate the quality of vehicle behaviour since the individual feature specifications are not useable in cases where feature actions are changed in order to resolve interactions.

The evaluation criterion were defined in relation to the output of the simulator. First, we can observe the simulator's output variable values. Those variables are not restricted to attribute variables, but also include related vehicle information such as sensor outputs, vehicle speed, and so on. Nearly all variables that the simulator can monitor are variables that we care about. For example, to evaluate the stability of the outputs of the speed resolution module over a sequence of actions, we observe output values of speed - to see whether changes in speed over short time periods would result in a smooth ride. In addition, PreScan also provides a powerful 3D viewer, with which you can observe the actor's behaviour in multiple views (see in Figure 3.6), to see whether abnormal behaviour occurs, such as the vehicle suddenly stopping, going off the road, and so on.

We list our basic criterion in Table 5.2 and Table 5.3. In Table 5.3, the acceleration threshold of $\leqslant 0.3g$[2] comes from a study conducted by L.L. Hoberock [17], and the lateral-acceleration threshold of $\leqslant 0.12g$ is advocated in [10]. However, our observations were not limited to the listed criteria; human subjective judgments were also used.

We evaluated our features and resolution strategies by comparing observed behaviour against the listed criteria, as well as evaluating whether global system properties are satisfied. Observation and evaluation helped us identify existed problems and fix them accordingly.

[2]g is gravitational acceleration, which is regarded as 9.8 m/s^2

Test Passed	**Test not Passed**
Speed Features: Vehicle does not crash into actors or obstacles	Speed Features: Vehicle crashes into actors or obstacles
Steering Features: Vehicle stays in the center of the current lane except when changing lanes	Vehicle leaves the current lane when NOT changing lanes.
Steering Features: Vehicle stays on the road	Vehicle goes off the road.
Warning Features: Vehicle issues warning signals when a triggering condition occurs (such as when the distance between the vehicle and the preceding car is smaller than 30 m).	Vehicle does not issue warning signals correctly when a triggering condition is met.

Table 5.2: Evaluation Criteria

Test Passed	**Test not Passed**
Speed Features: Vehicle's acceleration does not exceed 0.3 g except in emergency requirements [17].	Vehicle's acceleration exceeds 0.3 g, outside of emergency requirements.
Speed Features: Vehicle's speed is not turbulent. (We define "turbulence" as a continuous speeding up and slowing down of the vehicle iteratively, which means that the speed plot consists of several same-shaped waves)	Vehicle's speed has turbulence.
Steering Features: Vehicle's lateral acceleration does not exceed 0.12 g, except in emergency requirements [10].	Vehicle's lateral acceleration exceeds 0.12 g outside of emergency requirements.

Table 5.3: Quantitative Criterion to Tests

5.2 Resolution Strategy

Before we evaluate the current architecture, we have to implement the resolution modules for attribute variables first. As introduced above, we have three attribute variables: speed, steering and warning signal, which means that three separate resolution modules should be implemented, respectively. Each resolution strategy introduced in this section is a default resolution strategy that we determined to be appropriate for the resolution module's corresponding attribute variable. Developers who use the current architecture can adjust the resolution strategies according to the specific attribute variables that are controlled by the features in their system.

Currently, we have three resolution modules, i.e. Speed Resolution Module, Steering Resolution Module, and Warning Resolution Module. They each use a different strategy for resolving conflicts among actions on their respective attribute variable. Recall that in 4.3, we introduced the concept of a task feature. These features are treated specially in the resolution strategy.

5.2.1 Speed Resolution Strategy

Speed is one of the main attribute variables of a vehicle and its value can be affected by decisions that are made in other resolution modules. The speed resolution module receives as inputs in each execution step: **a)** a set of acceleration targets (from features), **b)** a set of speed targets (from features), **c)** a set of speed bounds (from features), **d)** the current speed (from sensors), **e)** the projected steering angle (from the steering resolution module), **f)** task features' requests and acceleration values (from task features). The output of the speed resolution module is an acceleration. The resolution strategy we have devised is introduced below, followed by the pseudo code in Algorithm 1:

1. Basic resolution strategy: The basic resolution strategy for the speed resolution module is to choose the minimum value among all the accelerations and the calculated accelerations according to target speed values (See line 16. The acceleration is calculated to ensure that the vehicle achieves the target speed value in a predefined distance.) - because, in automobiles, a smaller acceleration has a greater possibility of being 'safe' than a larger acceleration. See line 20 of Algorithm 1.

2. Constraints/bounds: In Section 4.2, we introduced constraints on attribute variables. In our resolution strategy, if a bound is violated, it needs to be restored. If a bound is not violated, it will be used as a constraint in the calculations of the next acceleration

value (with a default time, see line 15), to help ensure that the constraint remains satisfied. If there are multiple constraints, the algorithm works with the smallest constraint. See line 14. Additionally, the speed target is also used as a constraint to calculate an acceleration (with a default time) in order to avoid thrashing on target value of corresponding features. See line 16.

3. Coupled variables: To keep the vehicle's lateral-acceleration within an acceptable range, the speed resolution also takes as input the next value of the steering angle from the steering resolution module, and calculates a corresponding speed upper bound. See line 13. Equation 5.1 show the calculation for determining the steering - induced bound on speed.

$$tangential\ speed = \frac{maximum\ lateral\ acceleration}{steering\ angle} \tag{5.1}$$

4. Task features: Task features have higher priority than other features. If a task feature sends a request to the Speed Resolution Module, the Resolution Module will check if another task feature is already active. If there is another active task feature, the request will be declined. If there is no other active task feature, this task feature will take control of the resolution module until the task finishes with a release signal. We used a mutex $Task\_feature$ to realize this mechanism. See lines 2 - 11.

Algorithm 1 is the resolution strategy that we have devised for speed features, and we use it in the Speed Resolution Module in our case study.

```
Data: Accel_array, SpeedTarget_array, Speedbound_array, NextSteeringAngle,
      CarSpeed, Task_Request, Task_Acceleration
Result: TargetAccel
1  if Task_feature == locked then
2  |  TargetAccel = exist Task_Acceleration;
3  |  if Task_feature request release then
4  |  |  Task_feature = release;
5  |  else
6  |  |  do nothing;
7  |  end
8  else
9  |  if Task_Request then
10 |  |  TargetAccel = Task_Acceleration;
11 |  |  Task_feature = locked;
12 |  else
13 |  |  PredictSpeedBound = LateralAcceleration/NextSteeringAngle;
14 |  |  Speedbound = min(min(Speedbound_array), PredictSpeedBound);
15 |  |  calcAccel = (Speedbound - CarSpeed)/DefaultTime;
16 |  |  calcAccel2 = (min(SpeedTarget_array)-CarSpeed)/DefaultTime;
17 |  |  if calAccel2 and Accel_array is empty then
18 |  |  |  TargetAccel = 0;
19 |  |  else
20 |  |  |  TargetAccel = min(calcAccel, calAccel2, Accel_array);
21 |  |  end
22 |  end
23 end
```

Algorithm 1: Speed Resolution Strategy

5.2.2 Steering Resolution Strategy

Steering features control the position of the vehicle on the road. The steering resolution module receives as inputs in each execution step: **a)** a set of steering-angle targets (from features), **b)** a set of steering bounds (from features), **c)** task features' requests and steering-angle targets (from task features). The output is a target angle of steering. Note that

derivatives for steering angle is not accepted as an input to the dynamics, therefore derivatives is not considered within the steering resolution module. The resolution strategy we have devised for resolving conflicts over actions on the vehicle's steering angle is introduced below, followed by the pseudo code in Algorithm 2:

1. Basic resolution strategy: The basic resolution strategy for the steering resolution module is to use the average value of the target steering angles from the features. Using the average value of the target values can balance the requirements of all of the steering features. See line 13 of Algorithm 2.

2. Constraints/bounds: The Steering Resolution Module takes as input the bounds from features, to keep the steering angle within an acceptable range. See line 15.

3. Task features: Task features have higher priority than other features. If a task feature sends a request to the Steering Resolution Module, the Resolution Module will check if another task feature is already active. If there is another active task feature, the request will be declined. If there is no other active task feature, this task feature will take control of the resolution module until the task finishes with a release signal. We used a mutex $Task\_feature$ to realize this mechanism. See line 1 - 7, 11.

Algorithm 2 is the resolution strategy that we used in the Steering Resolution Module in our case study.

```
Data: Angle_array, Angle_bounds, Task_Request, Task_Angle
Result: TargetAngle
if Task_feature == locked then
    TargetAngle = exist Task_Angle;
    if Task_feature request release then
        Task_feature = release;
    else
        do nothing;
    end
else
    if Task_Request then
        TargetAngle = Task_Angle;
        Task_feature = locked;
    else
        Angle = Average(Angle_array);
        if Angle > Angle_bounds then
            TargetAngle = Angle_bounds;
        else
            TargetAngle = Angle;
        end
    end
end
```

Algorithm 2: Steering Resolution Strategy

5.2.3 Warning Resolution Strategy

For the sake of this case study, we assume that all the warnings are being displayed by the same device (e.g., a single warning light), so that there are conflicts to resolve. The inputs of the warning resolution strategy are warning signals from features, and the output is a command to the warning light (on or off). The basic strategy is to turn on the warning light if there is at least one input warning. Algorithm 3 is the resolution strategy that we have devised for warning features, and we use it in the Warning Resolution Module in our

case study.

Data: Warning_array
Result: WarningLight
1 WarningLight = 0;
2 **for** *Warnings in Warning_array* **do**
3 | $WarningLight = WarningLight \vee Warnings$
4 **end**

Algorithm 3: Warning Resolution Strategy

5.3 Feature Tests

As shown in Table 5.1, we have a total of seven features targeting the vehicle's speed, namely Speed Limit Control (SLC), Cruise Control (CC), Headway Control (HC), Predictive Pedestrian Protection Braking (PPPB), Traffic Jam Assist (TJA), Predictive Braking System (PBS), Emergency Braking System (EBS). A total of two features targeting steering angle, namely Lane Keeping Assist (LKA) and Lane Changing Control (LCC). And there are six warning features: they are Lane Departure Warning (LDW), Collision Warning (CW), Pedestrian Protection Warning (PPW), Lane Change Assist (LCA), Rear Cross Traffic Alert (RCTA), and Parking Aid (PA).

A set of tests were performed to verify that each of our feature implementations performs correctly in isolation. This testing ensures that when testing the quality of resolutions produced by the current architecture, the resolutions are not affected by errors in the logics of individual features. In all feature tests, we devised three types of test scenarios: normal scenarios, transition scenarios, and boundary scenarios. To explain the different types of scenarios, we us CC as the representative feature. If the desired cruising speed is not met, CC will accelerate or decelerate the vehicle speed to the desired cruising speed. As the vehicle speed approaches the cruising speed, the absolute value of acceleration is reduced to zero. Each normal scenario tests one of the features' expected behaviours. For example, in a normal test for CC, where the initial vehicle speed is 75km/h, the desired cruising speed is 90km/h: the expected behaviour is that CC will accelerate the vehicle (because the vehicle's speed is below the desired cruising speed). Each transitional scenario tests that a feature correctly transitions between two normal cases. For example, in a transitional test for CC, where the initial vehicle speed is 75km/h, the initial desired cruising speed is 90km/h, and there is a second desired cruising speed of 80km/h later in the test scenario: CC will accelerate the vehicle until the vehicle speed reaches 90km/h; when the desired cruising speed is subsequently changed to 80km/h, the CC will decelerate the vehicle to

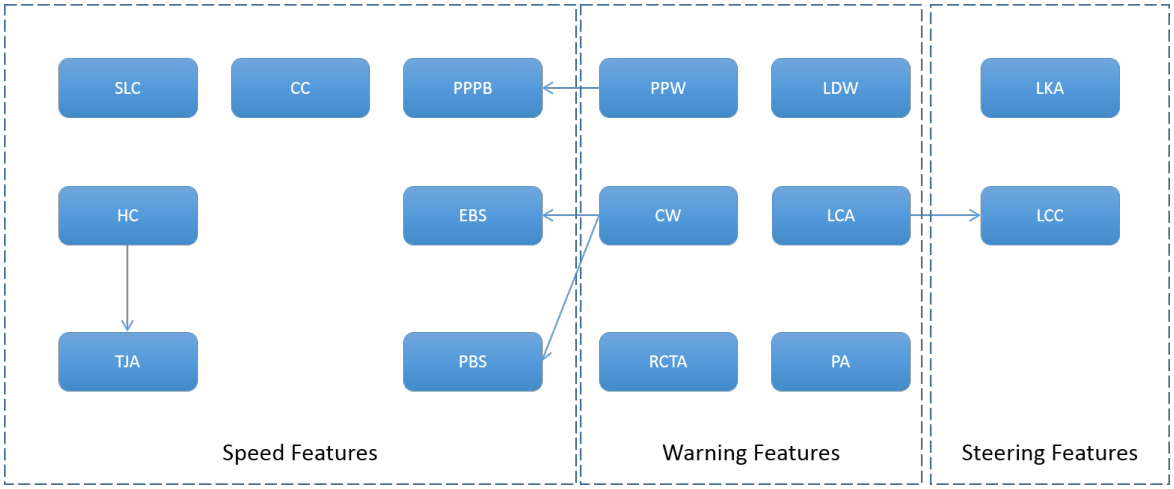

Figure 5.1: Feature dependencies configuration

80km/h. Each boundary scenario tests a feature at or very near the conditions at which the feature transitions between two normal scenarios. For example, in a boundary test for CC, the vehicle speed is 91 km/h and the desired cruising speed is 90km/h. We devise test suites such that every normal, transitional, and boundary scenario is covered by at least one test case; these test cases are representative of all the possible test cases.

All the features were implemented and tested separately. PreScan does not provide the ability to build complex test environments such as uphill or downhill gradients. Thus, we did not devise tests that would require control logic that accounts for environmental conditions, such as road gradient for speed features, or oversteering/understeering situations for steering features.

It is worth noting that some features cannot be run in isolation, because they rely on the presence of other features to operate correctly. For example, PBS is activated by a warning signal from CW. Such features were not tested separately, but were tested with the minimal set of dependent features. The dependencies of features are shown in Figure 5.1. Features that have an incoming arrow depend on the features at the other end of the arrow.

5.3.1 Speed Features

The case study includes seven features that try to control attribute variable speed. The functionalities of all of these features are briefly described in Table 5.1. Among these

seven features, SLC, CC, and HC are features that have the most interactions because these three features can have different goals for the vehicle's speed. TJA has similar goals to HC. PPPB, PBS, and EBS are all emergency braking features that suddenly activate the brake, or prepare for full activation of the brake, to avoid a collision. PPPB depends on PPW: when PPW issues a warning signal, PPPB will apply emergency brakes to the vehicle. Both EBS and PBS depend on CW, and they will activate only when CW issues a warning of a potential collision. All features were shown to work correctly in isolation, as evidenced by their passing their respective feature-specific feature tests. Below we describe the feature tests for CC and SLC, as being representative of the feature tests we devised for all speed features in the case study. Other speed feature tests can be found in Appendix A.

Speed Limit Control

SLC aims to help ensure that the vehicle's speed conforms to the speed limit. SLC continuously monitors the vehicle's speed. However, the behaviour of SLC is constant: all the feature does is to continuously output a constraint on speed, regardless of the vehicle's speed or the speed limit. To see the effect of the feature's behaviour on the vehicle's behaviour, the feature test scenarios need to include the resolution module, which outputs a command on the vehicle's acceleration.

If the vehicle speed exceeds the speed limit, the output command will be a deceleration between -3 m/s^2 and 0 m/s^2, in order to reduce the vehicle speed to be below the speed limit as quickly as possible, while satisfying the criterion on maximum acceleration. (The output command strategy is shown in 15)

We list the feature tests for SLC in Table 5.4. As shown in the table, the middle columns explain the test inputs, and rightmost column describes the expected output. For example, in Transition Case 1, the initial speed of the vehicle is 75 km/h, the acceleration of the vehicle is 3 m/s^2, and the speed limit is 90 km/h. The expected output of this scenario is that the vehicle accelerate to 90 km/h from 75 km/h, and then the speed remains 90 km/h. Simulation results are provided for some of the test scenarios.

SLC passed all tests with respect to keeping the vehicle's speed at or below the speed limit, or decreasing the vehicle speed to be below the speed limit, and satisfying all of the qualitative and quantitative criteria.

Title	Initial Speed	Acceleration	Speed Limit	Result
Normal Case 1	75 km/h	0 m/s^2	90 km/h	Hold status.
Normal Case 2	100 km/h	0 m/s^2	90 km/h	Decelerate to the speed limit and hold.
Transition Case 1	75 km/h	3 m/s^2	90 km/h	Accelerate to the speed limit and hold. See Figure 5.2
Transition Case 2	100 km/h	3 m/s^2	90 km/h	Decelerate to the speed limit and hold. See Figure 5.3
Boundary Case 1	89 km/h	3 m/s^2	90 km/h	Accelerate to the speed limit and hold.
Boundary Case 2	90 km/h	3 m/s^2	90 km/h	Hold status
Boundary Case 3	91 km/h	3 m/s^2	90 km/h	Decelerate to the speed limit and hold. See Figure 5.4

Table 5.4: Feature tests - SLC

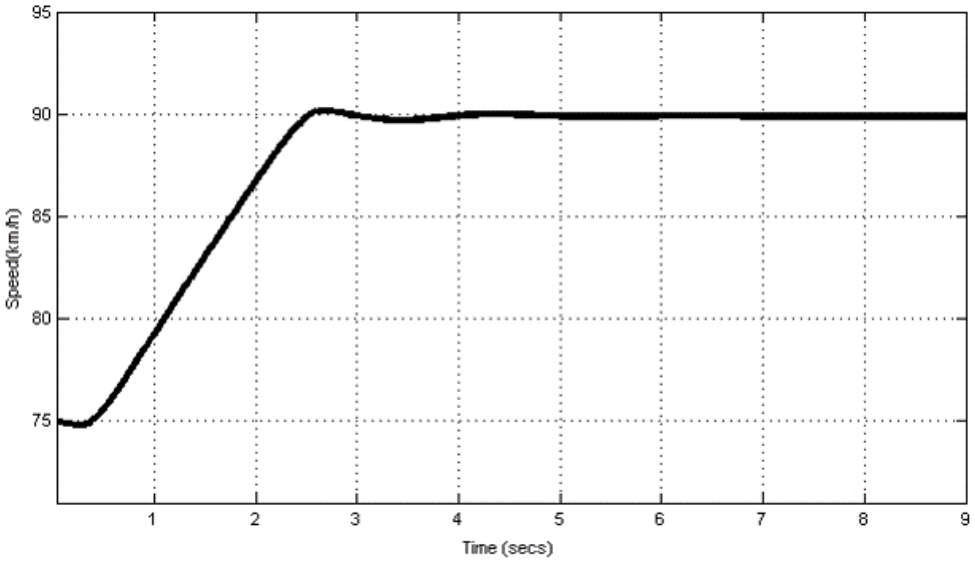

Figure 5.2: Speed Limit Control Transition Scenario 1

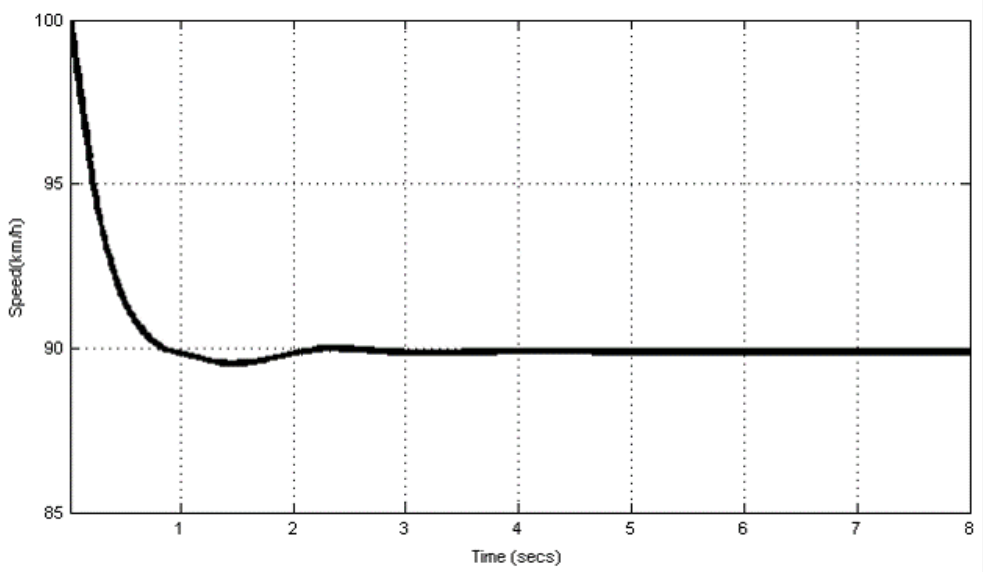

Figure 5.3: Speed Limit Control Transition Scenario 2

Title	Initial Speed	Desired Cruise Speed 1	Desired Cruise Speed 2	Result
Normal Case 1	75 km/h	90 km/h	-	Accelerate to the desired cruising speed and hold. See Figure 5.5
Normal Case 2	100 km/h	90 km/h	-	Decelerate to the desired cruising speed and hold. See Figure 5.6
Normal Case 3	90 km/h	90 km/h	-	Hold status
Transition Case 1	100 km/h	90 km/h	100 km/h	Decelerate to the desired cruising speed 1, then accelerate to the desired cruising speed 2, and hold. See Figure 5.7
Transition Case 2	90 km/h	100 km/h	90 km/h	Accelerate to the desired cruising speed 1, then decelerate to the desired cruising speed 2, and hold.
Boundary Case 1	89 km/h	90 km/h	-	Accelerate to the desired cruising speed and hold.
Boundary Case 2	91 km/h	90 km/h	-	Decelerate to the desired cruising speed and hold. See Figure 5.8

Table 5.5: Feature tests - CC

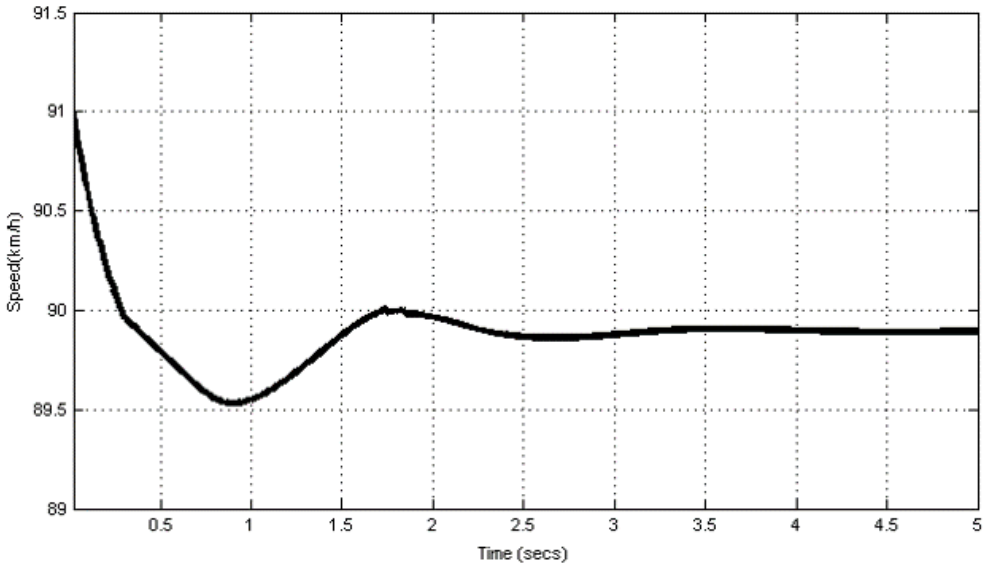

Figure 5.4: Speed Limit Control Boundary Scenario 3

Cruise Control

Cruise Control aims to help keep that the vehicle's speed consistent with the desired cruising speed of the vehicle, which is set by the driver. CC continuously monitors the vehicle's speed, and actively controls the vehicle's acceleration whenever the vehicle's speed is not consistent with the desired cruising speed. In each test case, the feature outputs an acceleration between -3 m/s^2 and 3 m/s^2, in order to achieve a vehicle's speed that matches the desired cruising speed as quickly as possible, while satisfying the criterion on maximum acceleration. We list the feature tests for CC in Table 5.5, and provide the simulation results for some of the tests.

CC passed all of the tests with respect to keeping the vehicle's speed at the desired cruising speed, or quickly achieving a vehicle speed that matches the desired cruising speed, and satisfying all of the qualitative and quantitative criteria.

5.3.2 Steering Features

The case study includes two steering features: LCC and LKA. We wanted to include more features that target steering but we were unable to find additional features. However, even between these two features, there are interesting feature interactions, and the default

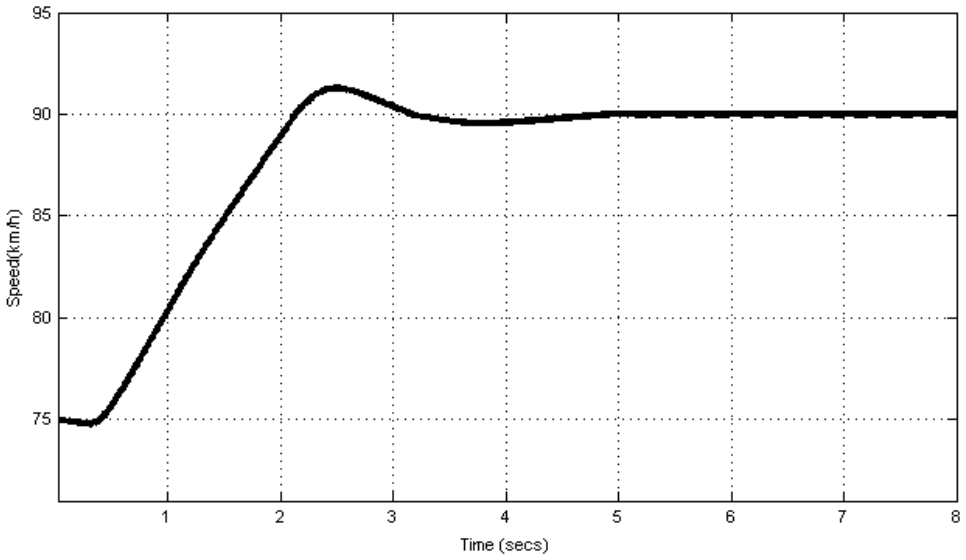

Figure 5.5: Cruise Control Normal Scenario 1

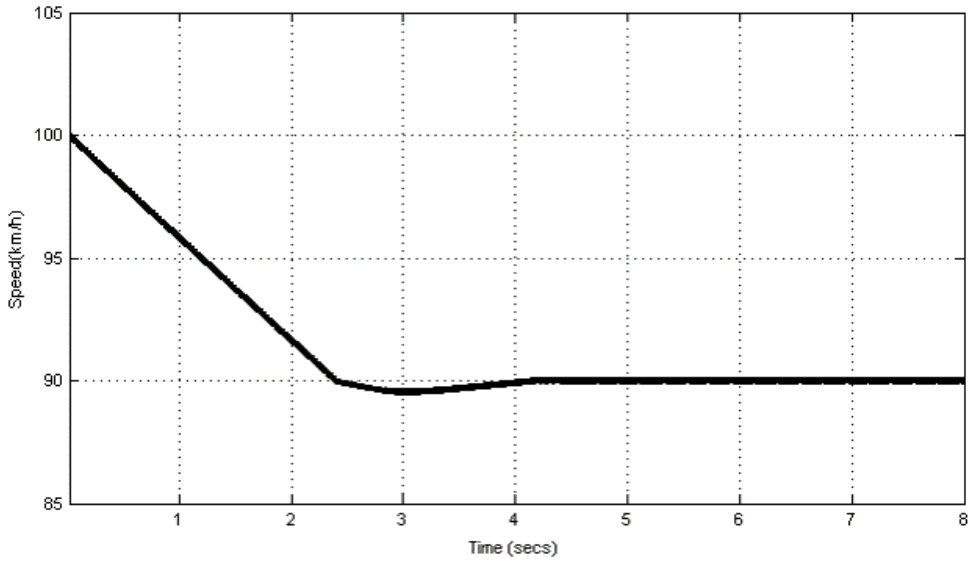

Figure 5.6: Cruise Control Normal Scenario 2

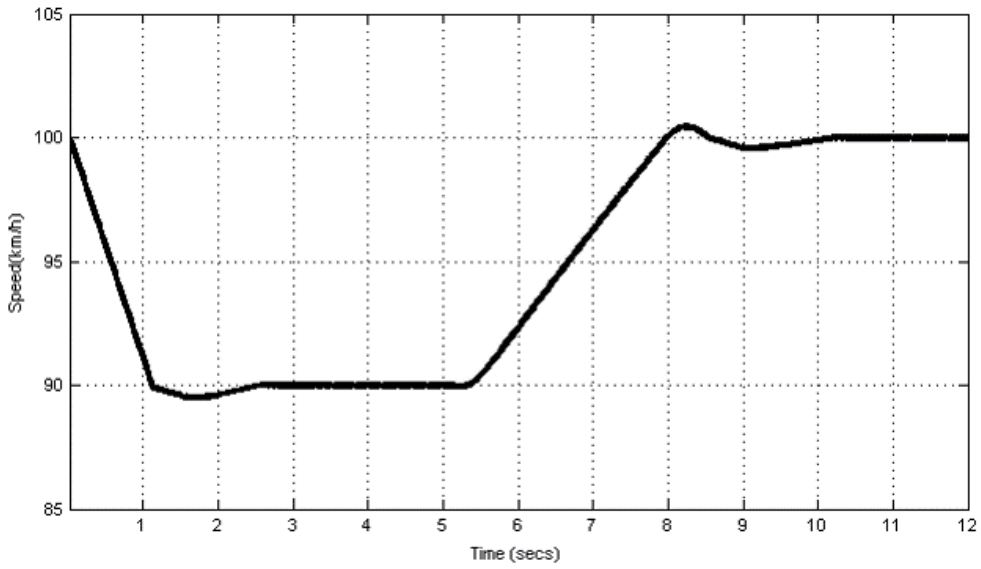

Figure 5.7: Cruise Control Transition Scenario 1

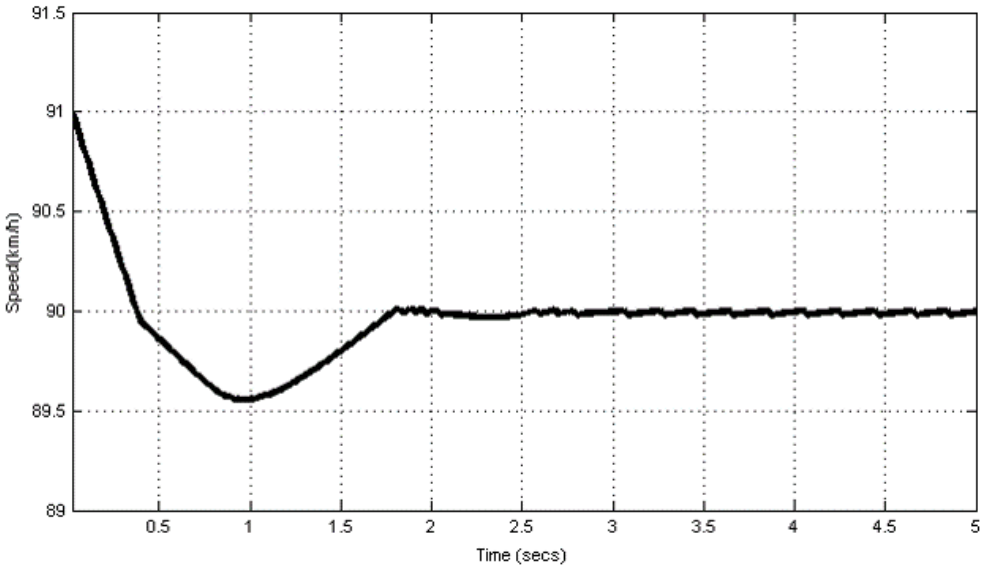

Figure 5.8: Cruise Control Boundary Scenario 2

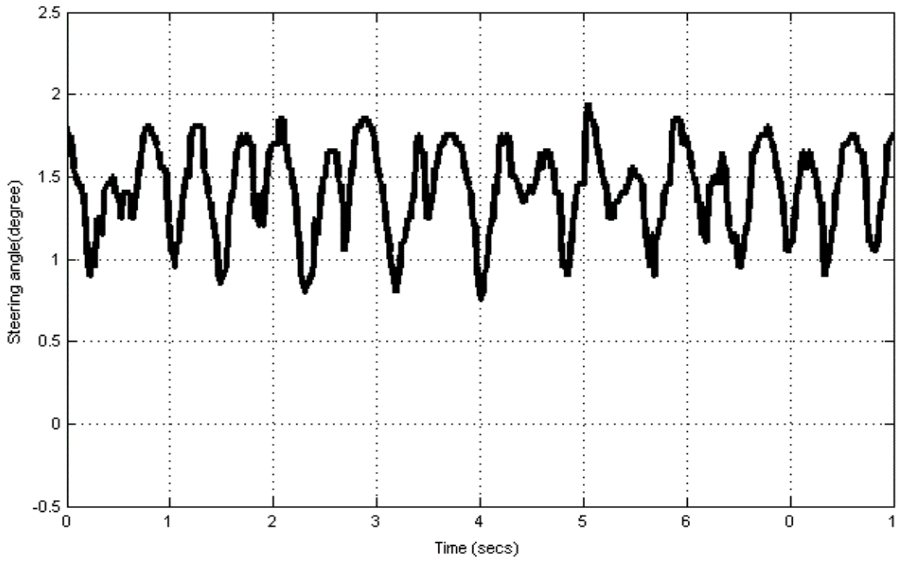

Figure 5.9: Lane Keeping Assist Normal Scenario 1

strategy for resolving interactions over the value of the steering variable are more complex than the strategy for resolving interactions over the value of the speed variable. Because LCC itself depends on LCA and will rely on LKA to help it adjust after finishing a lane-change action, LCC cannot be tested in isolation. Thus, below, we present the feature tests for LKA as being representative of feature tests for steering features. Interaction tests between LCC and LKA can be found in Appendix C.2.

Lane Keeping Assist

Lane Keeping Assist aims to keep the vehicle in its current lane. The feature continuously monitors the locations of the lane markings and it actively directs the vehicle's steering towards the centre of the lane if the vehicle drifts towards the edge of the lane.

We list feature tests for LKA in Table 5.6, and some of the test result figures are provided.

LKA passed all of the tests with respect to keeping the vehicle in its lane and satisfying all qualitative and quantitative evaluation criteria.

Title	Initial Speed	Road State 1	Road State 2	Result
Normal Case 1	80 km/h	Curve Left	-	Steer left to keep the vehicle in the lane during a left curve in the road. See Figure 5.9
Normal Case 2	80 km/h	Straight	-	Hold status
Normal Case 3	80 km/h	Curve Right	-	Steer right to keep the vehicle in the lane during a right curve in the road.
Transition Case 1	80 km/h	Straight	Curve Left	Transit from driving straight to driving towards the left. See Figure 5.10
Transition Case 2	80 km/h	Straight	Curve Right	Transit from driving straight to driving towards the right.
Transition Case 3	80 km/h	Curve Right	Straight	Transit from driving to the right to driving straight.
Transition Case 4	80 km/h	Curve Right	Curve Left	Transit from driving to the right to driving to the left.
Transition Case 5	80 km/h	Curve Left	Straight	Transit from driving to the left to driving straight.
Transition Case 6	80 km/h	Curve Left	Curve Right	Transit from driving to the left to driving to the right. See Figure 5.11

Table 5.6: Feature tests - LKA

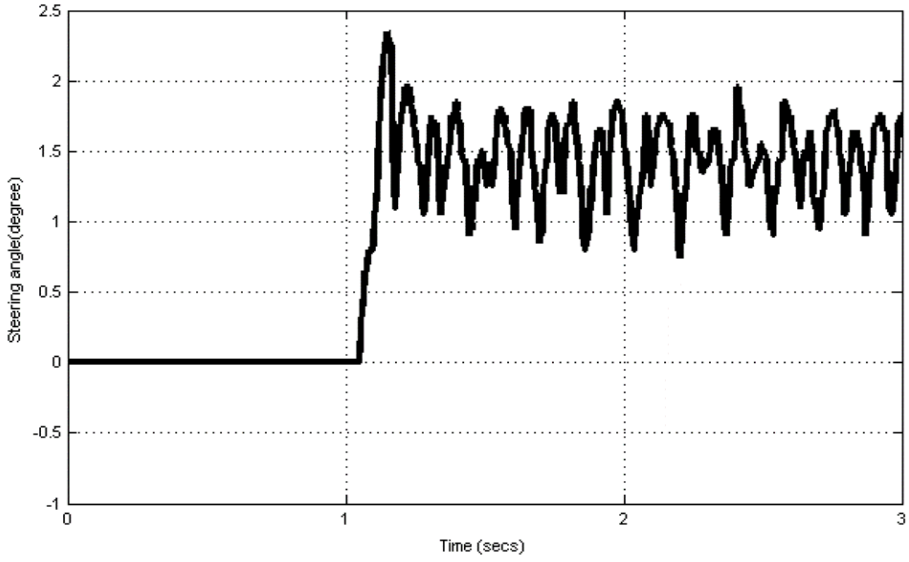

Figure 5.10: Lane Keeping Assist Transition Scenario 1

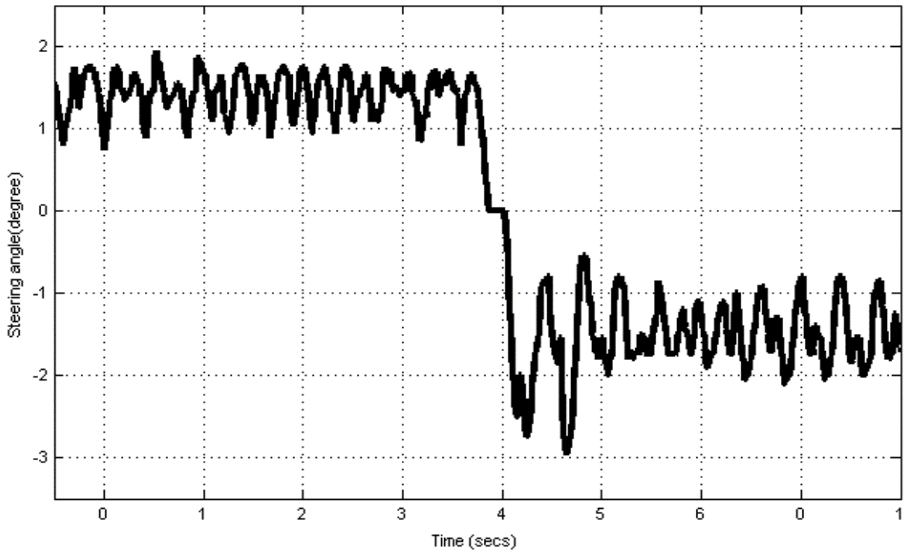

Figure 5.11: Lane Keeping Assist Transition Scenario 6

Title	Vehicle Speed	Preceding Vehicle Speed	Initial Distance	Distance to Issue Warning	Result
Normal Case 1	80 km/h	80 km/h	25 m	30 m	Issue a continuous warning. See Figure 5.12
Normal Case 2	80 km/h	80 km/h	35 m	30 m	No warning.
Transition Case 1	70 km/h	80 km/h	25 m	30 m	Transit from warning to no warning. See Figure 5.13
Transition Case 2	90 km/h	80 km/h	35 m	30 m	Transit from no warning to warning. See Figure 5.14

Table 5.7: Feature tests - CW

5.3.3 Warning features

The case study includes six warning features. The functionalities of these features are described in Table 5.1. Warning Features are relatively straight forward: they detect data from sensors, and send out a warning signal if feature-specific constraints are violated. For example, LDW will send out warning signal if it detects that the vehicle is starting to cross a lane marker. LCA will send out a warning signal if an obstacle is detected in the blind spot. We compare plots of output values of the sensors (e.g., distance to the preceding car, distance to the lane markings) with corresponding plots of output values of warning signals, to determine whether the warning light is displayed in the correct circumstances. All of warning features have a simple and similar feature logic. Therefore, we present below the feature tests for Collision Warning, as being representative of the tests for warning features. Tests for the other warning features are analogous, and are not shown.

Collision Warning

Collision Warning aims to warn the driver of an impending collision. The feature continuously monitors the distance to the preceding car and alarms the driver when the distance is critically short. We list the feature tests for LKA in Table 5.7, and provide the simulation results for some of the tests.

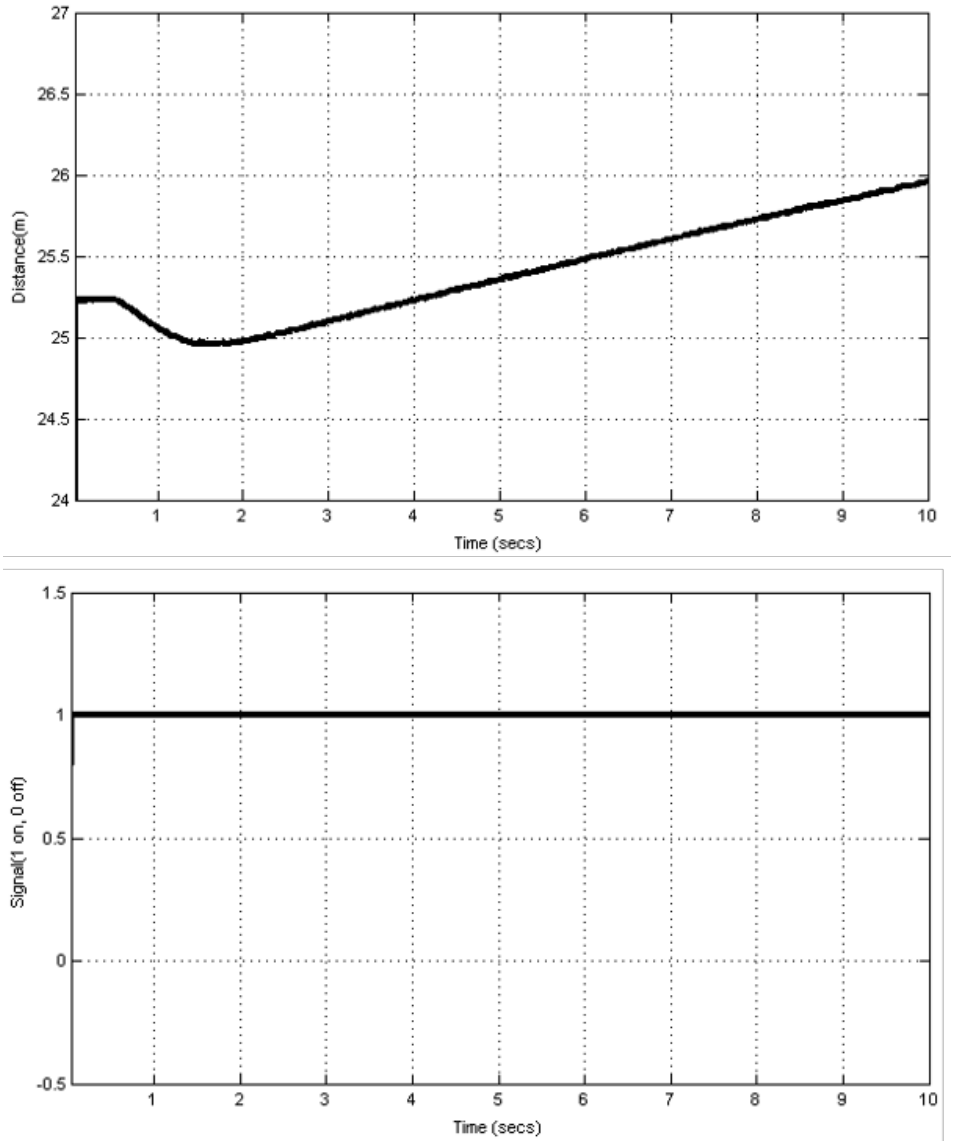

Figure 5.12: Collision Warning Normal Scenario 1

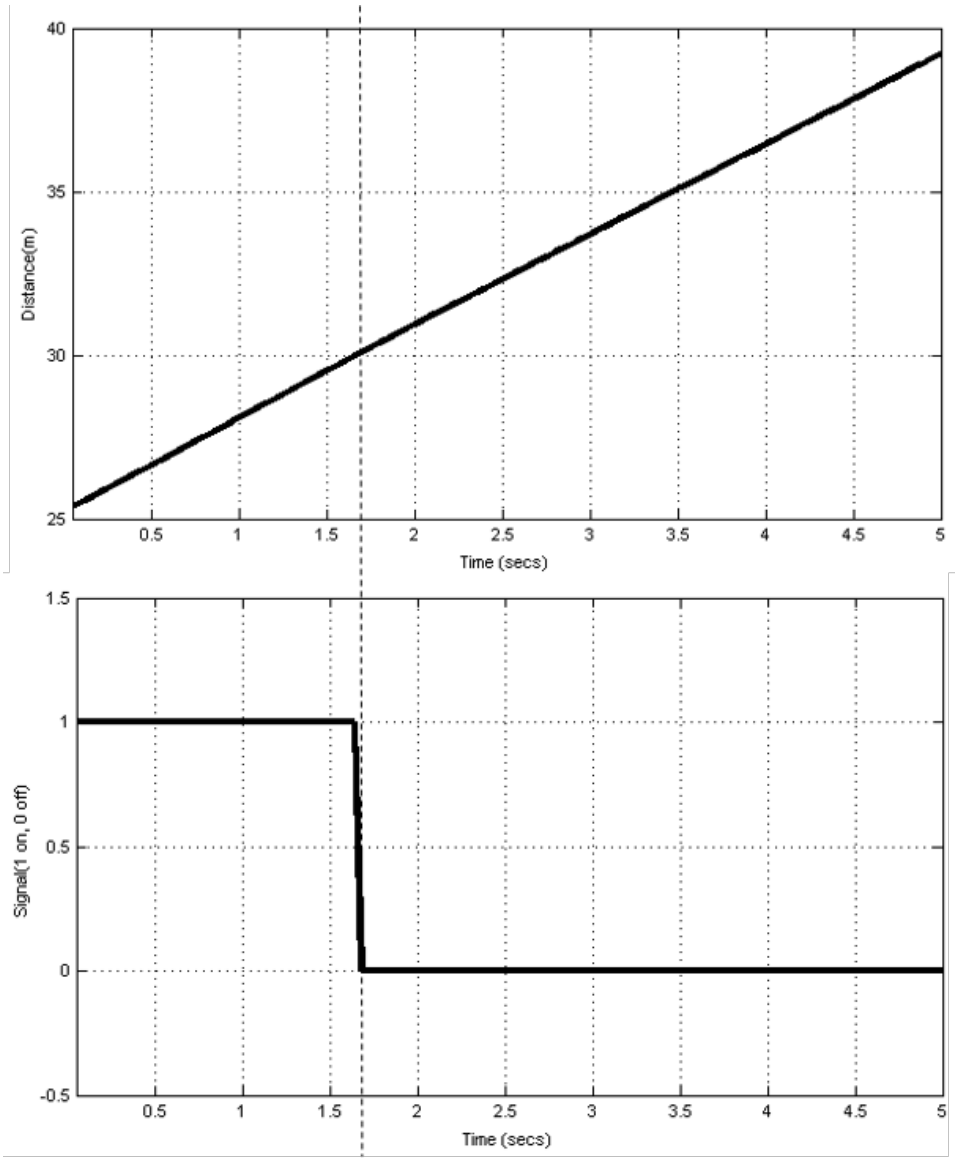

Figure 5.13: Collision Warning Transition Scenario 1

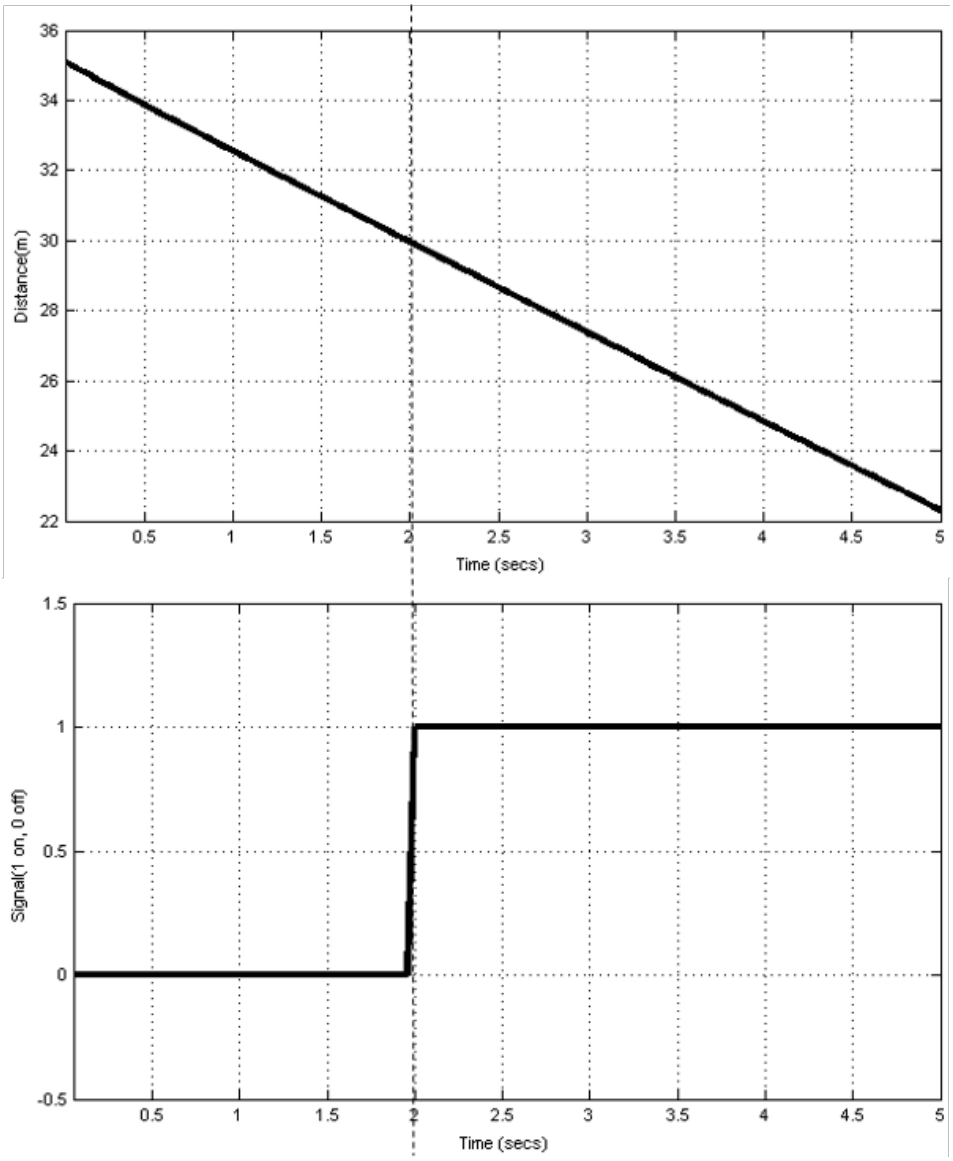

Figure 5.14: Collision Warning Transition Scenario 2

CW passed all of the tests with respect to warning the driver when the vehicle is too close to the preceding car, and satisfying all qualitative and quantitative evaluation criteria.

5.4 Evaluation

The evaluation focuses on the effectiveness of the current architecture in:

1. computing acceptable resolutions at runtime
2. producing sequences of resolved outputs that are smooth
3. devising acceptable resolutions over variables that are coupled

The methodology for designing test cases to evaluate the current architecture is conceptually the same as the methodology for designing feature tests. We divide all test cases into normal scenarios, transition scenarios, and boundary scenarios. Normal scenarios have no interactions. Transition scenarios specify different ways that features interact, including different ways the environment can cause interactions. Lastly, interactions barely occur in boundary scenarios. We use these three types of scenarios to test differences of the performance of the current architecture.

5.4.1 Resolving Feature Interactions within One Variable

To test that the current architecture can solve conflicts that are caused by feature interactions, we tested all possible pairs of features that can cause feature interactions. For conciseness, we describe one representative pair below (CC and HC), and other pairs are briefly described in Appendix C. Test cases for the feature pair CC and HC are shown in Table 5.8. Note that the way the preceding car "appears" in front of the vehicle is by cutting in front of the vehicle. The simulations in figures in 5.15 - 5.18 all start from the time that the preceding car cuts in. The dash line in the figure is the preceding car's speed which is sensor-detected, and the blibs of the line are noises from the sensor.

In Table 5.8's normal scenario, the distance to the preceding car never exceeds the safe distance, and the speed of the vehicle always keeps at or below the preceding car's speed (except for minor deviations), so HC never tries to control the vehicle's speed. The output of the resolution module is the output of the CC's output.

Title	**Initial Speed**	**Desired Cruise Speed**	**Preceding Vehicle Speed**	**Distance to preceding car (safe distance 30 m)**	**Result**
Normal Scenario	75 km/h	80 km/h	80 km/h	35 m	Accelerate to 80 km/h and hold. See Figure 5.15
Transition Scenario	90 km/h	90 km/h	80 km/h	35 m	Decelerate to 80 km/h and hold. See Figure 5.16
Boundary Scenario 1	80 km/h	90 km/h	80 km/h	31 m	Hold. See Figure 5.17
Boundary Scenario 2	79 km/h	90 km/h	80 km/h	31 m	Hold. See Figure 5.18

Table 5.8: Resolving feature interactions between CC and HC

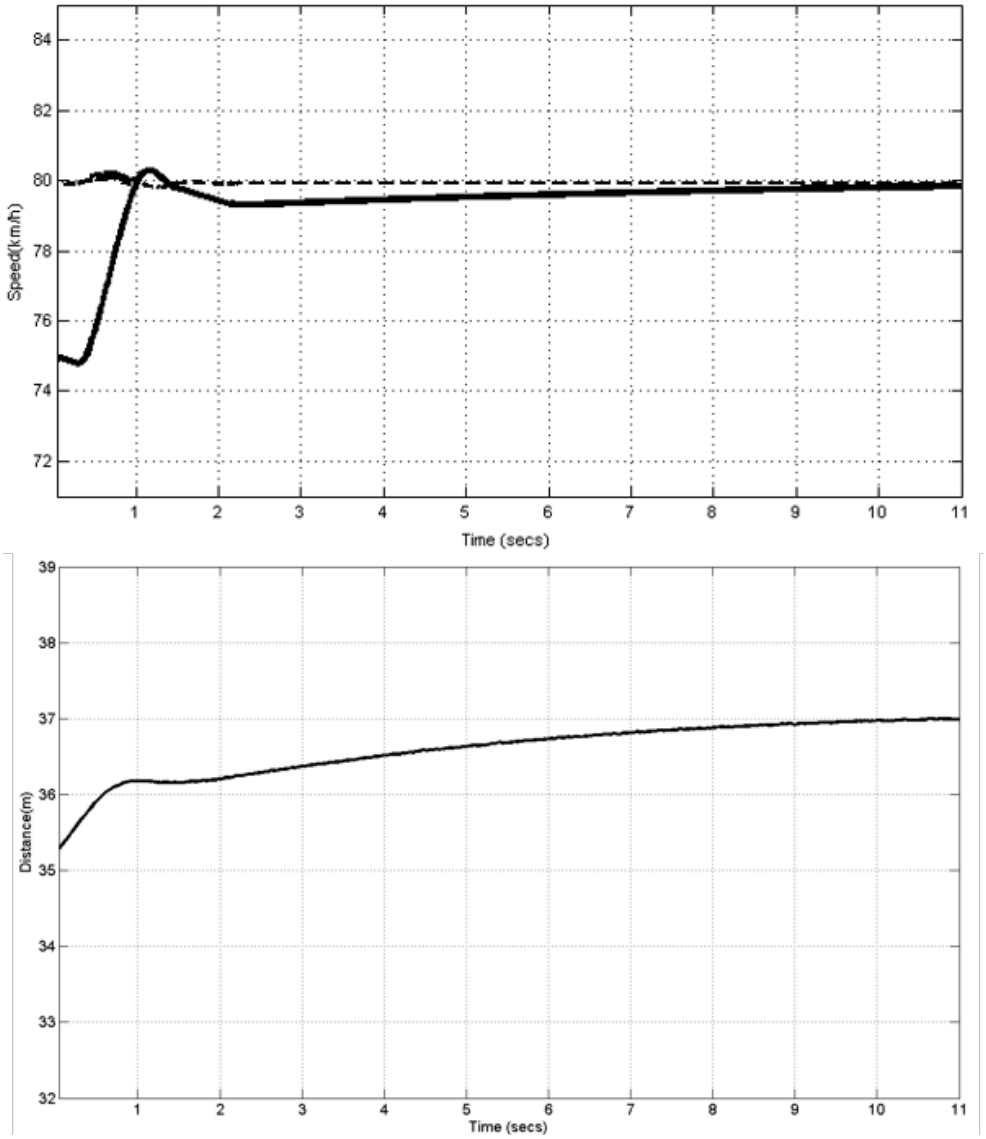

Figure 5.15: Cruise Control and Headway Control Resolution Normal Scenario

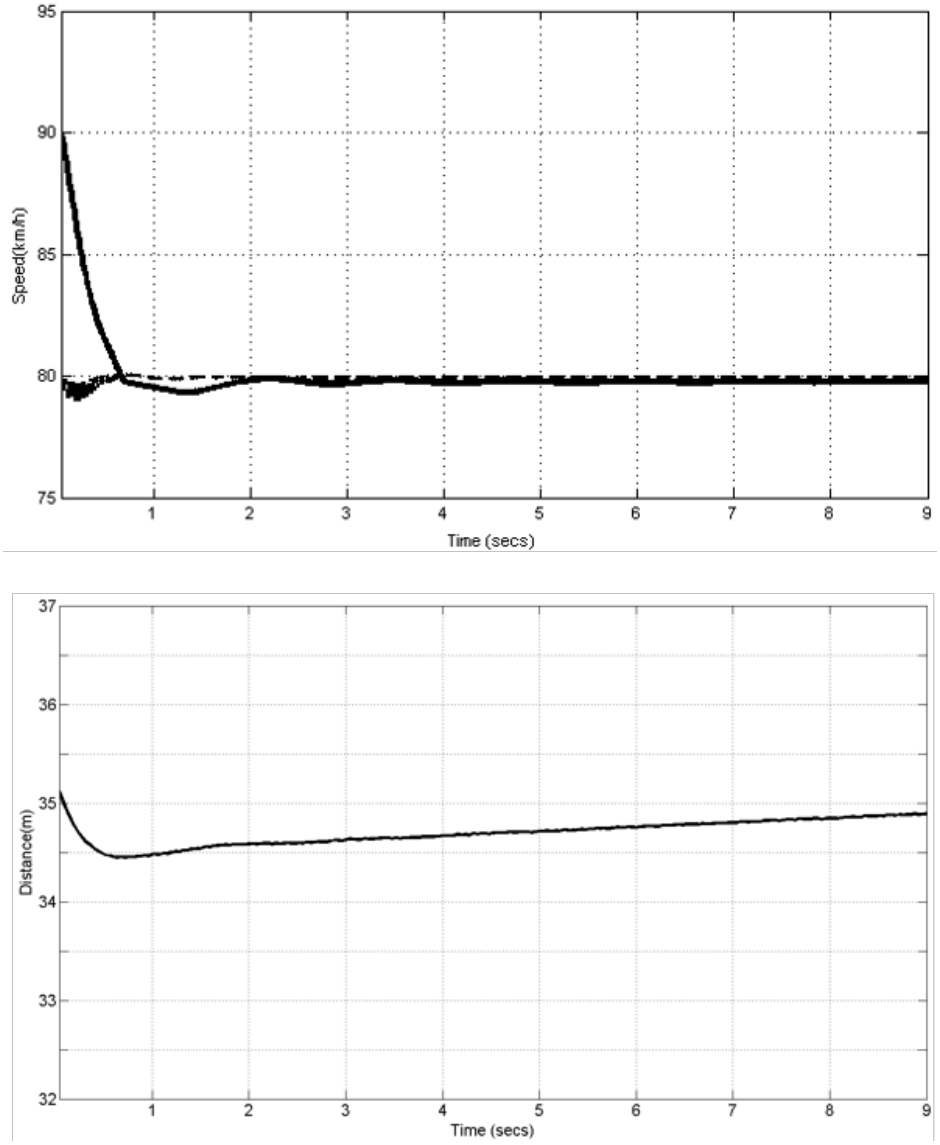

Figure 5.16: Cruise Control and Headway Control Resolution Transition Scenario

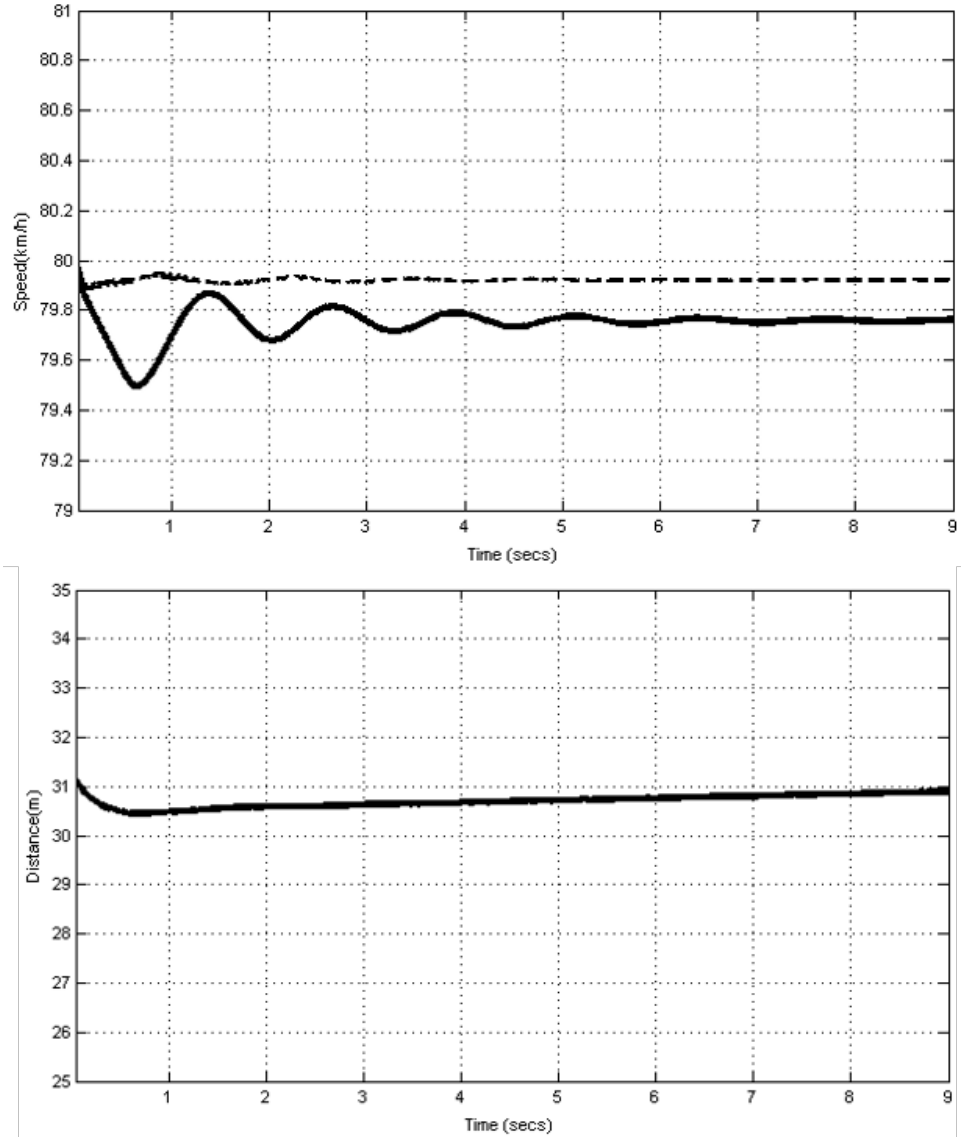

Figure 5.17: Cruise Control and Headway Control Resolution Boundary Scenario 1

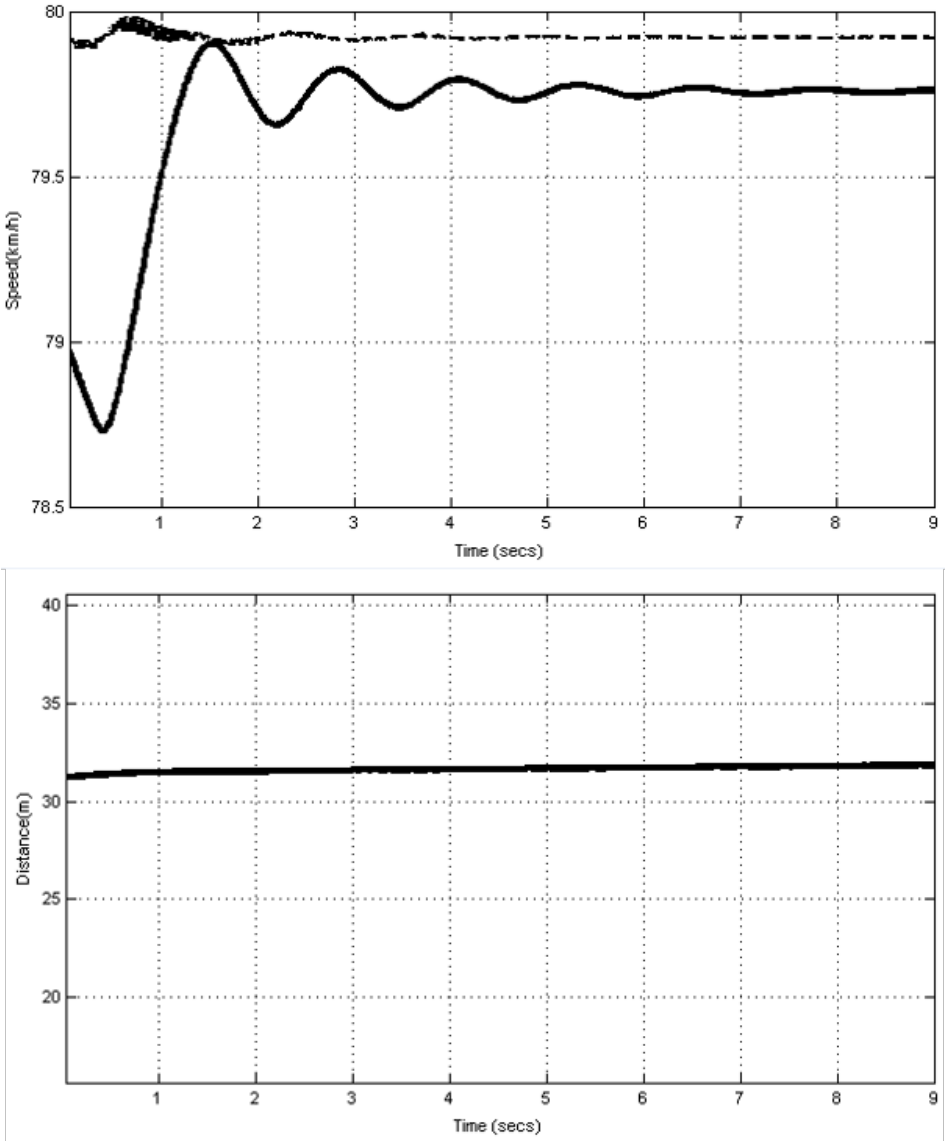

Figure 5.18: Cruise Control and Headway Control Resolution Boundary Scenario 2

In the transition scenario, the vehicle's speed is faster than that of the preceding car, so the vehicle will eventually catch up and will tend to violate the safe distance. Therefore, HC reduces the speed to lower than 80 km/h in order to keep or re-establish the safe distance. During the scenario, HC will reduce the vehicle's speed whenever the vehicle gets to be too close to the preceding car. As described in the speed resolution strategy 5.2.1, the resolution module will always choose the smaller acceleration. For example, in this scenario, when the speed is at 80 km/h, the acceleration from CC will be bigger than the acceleration from HC, so the HC will control the speed to keep the speed at or below 80 km/h.

The boundary scenarios are similar to the transition scenario, in that for much of the scenarios, CC and HC compete to control the vehicle's speed, with the resolution module always favouring the minimum proposed acceleration. This strategy keeps the vehicle at a safe distance from the preceding car, and guarantees that the vehicle's speed will not exceed the preceding car's speed.

We monitored the outputs (speed, acceleration) and vehicle's behaviour in each scenario to evaluate the current architecture. We evaluated the simulation results with respect to two aspects. First, all of the minimum evaluation criterion are met. For example, the vehicle did not crash, which satisfies the qualitative criterion, and the acceleration stays within the range defined by the quantitative criteria in Table 5.3. Second, global system properties are satisfied. For example, the vehicle stays within a safe distance and the speed of the vehicle does not exceed the preceding car's speed.

5.4.2 Smooth resolution

Allowing Derivatives

We devised scenarios that test whether allowing derivatives as input to the resolution module can resolve interactions involving the same target values of the same attribute variable but with different dynamic requirements. We chose HC and SLC as a representative pair of features to run experiments. Both features require the vehicle to slow down but under different requirements: they can output the same target speed for the vehicle at the same time, but with different derivatives. Other pairs, such as SLC and PPPB, can also benefit from actions that include derivatives.

In the transition scenario, both HC and SLC want the car to slow down to 80 km/h. However, these two features have different deceleration requirements, in that HC wants the vehicle to slow down faster since it is important to keep a safe distance between the vehicle

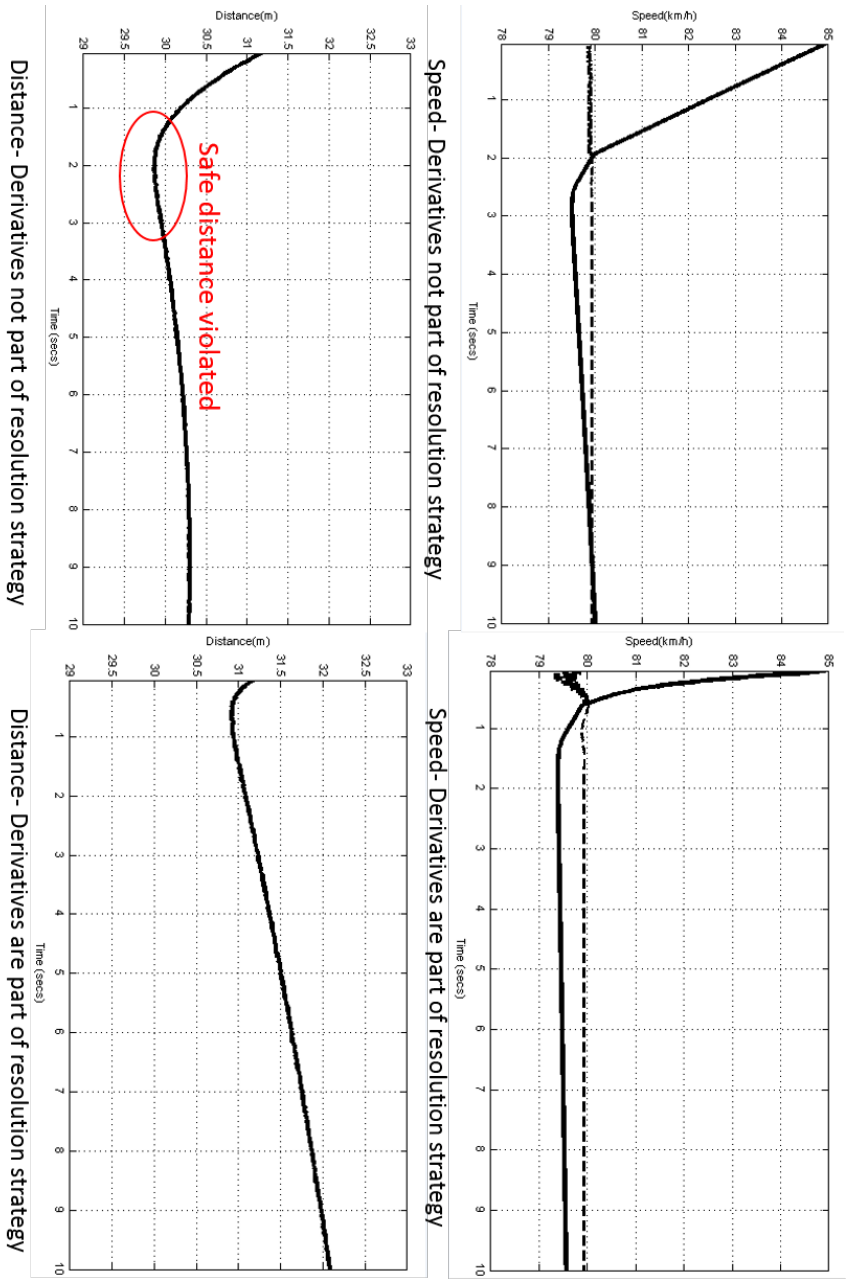

Figure 5.19: Improvement of resolutions due to the inclusion of derivatives in the resolution strategy (Transition Scenario)

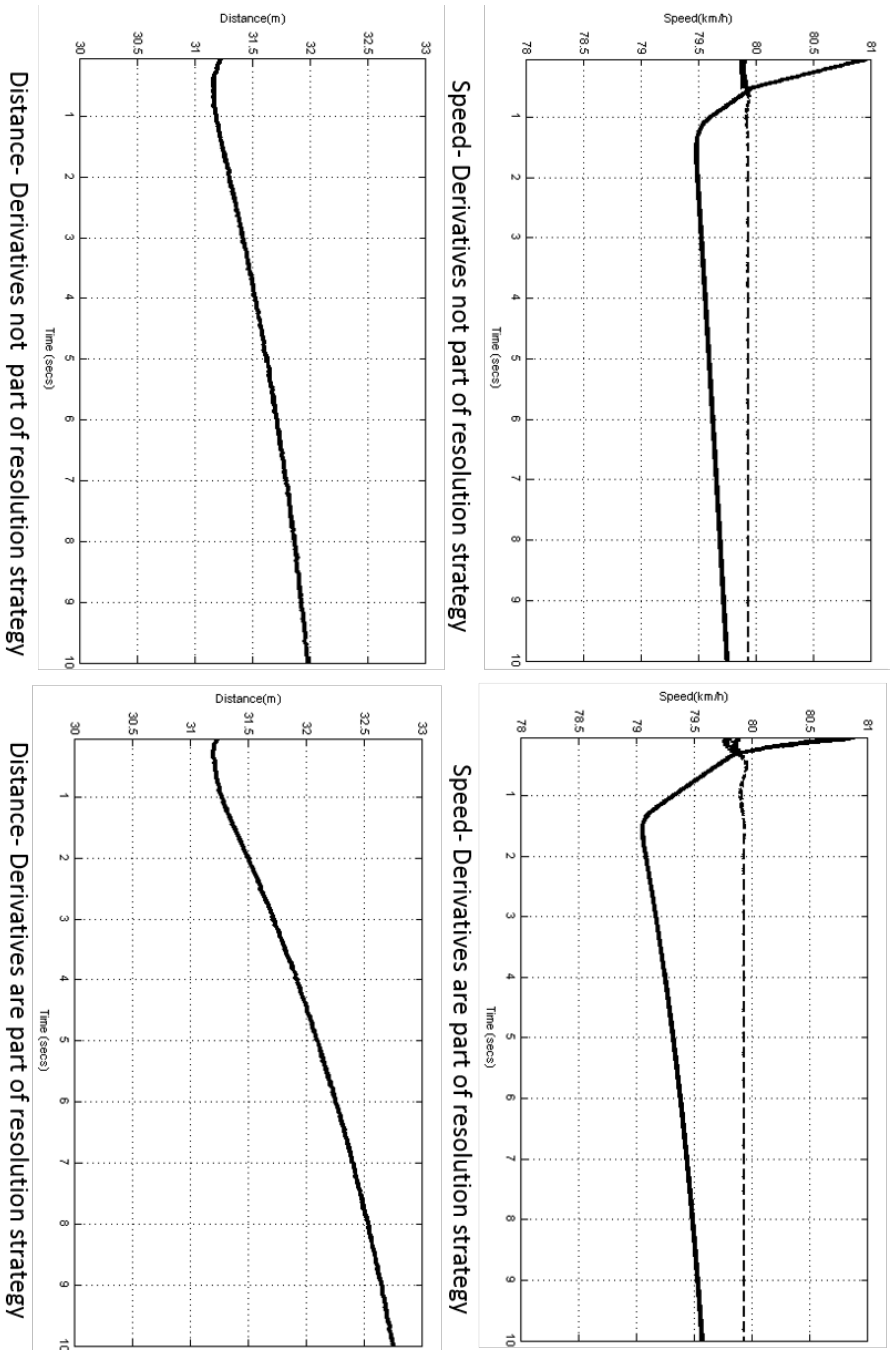

Figure 5.20: Improvement of resolutions due to the inclusion of derivatives in the resolution strategy (Boundary Scenario)

Title	Initial Speed	Speed Limit	Preceding Vehicle Speed	Distance to preceding car (safe distance 30 m)	Result
Transition Scenario	100 km/h	80 km/h	80 km/h	32 m	Decelerate to 80 km/h and hold. See Figure 5.19
Boundary Scenario	81 km/h	80 m/s^2	80 km/h	30 m	Decelerate to 80 km/h and hold. See Figure 5.20

Table 5.9: Resolving feature interactions between HC and SLC - allowing derivatives

and the preceding car. If derivatives are not used, the vehicle will slow down to the assigned speed at the default rate of deceleration implemented in the speed resolution module (see line 15, 16), which may not be fast enough, while neglecting different requirements, which may cause some severe consequences. In this scenario, if the vehicle slows down to 80 km/h at SLC's required rate, the vehicle will end up too close to the preceding car, which is a dangerous situation for the driver's safety.

In the boundary scenario, both HC and SLC want the vehicle to slow down, and they issue actions on the vehicle's acceleration that are small and similar. Figure 5.20 shows the difference between resolutions that do not consider target derivatives and resolutions that do consider target derivatives. As we can see, there is no critical difference between considering and not considering derivatives in the boundary scenario, because the features' target derivatives are similar in the boundary scenario, so there is little conflict to be resolved.

In summary, if features output only target values to the resolution module, the resolution module will compute default acceleration for the resolved target speed, which may be inappropriate in some circumstances. We can see that in the transition scenario, allowing derivatives in the resolution module allows the vehicle to slow down faster than if resolutions do not consider derivatives. The simulation result shown in Figure 5.19 shows the utility of allowing derivatives.

Title	Features Involved	Initial Speed	Speed Limit	CC Desired Speed	Preceding Vehicle's Speed	Distance to preceding car (safe distance 30 m)	Result
Transition Scenario 1	CC, SLC	75 km/h	90 km/h	100 km/h	-	-	Accelerate to 90 km/h and hold. See Figure 4.3.
Transition Scenario 2	CC, SLC	95 km/h	90 km/h	100 km/h	-	-	Decelerate to 90 km/h and hold. See Figure 5.21.
Transition Scenario 3	CC, HC, SLC	90 km/h	80 km/h	90 km/h	80 km/h	35 m	Decelerate to 80 km/h and hold. See Figure 5.22.
Boundary Scenario	CC, SLC	89 km/h	90 km/h	91 km/h	-	-	Accelerate to 90 km/h and hold. See Figure 5.23.

Table 5.10: Resolving feature interactions - allowing attribute bounds

Attribute bounds

We devised scenarios to assess whether resolutions that consider bounds perform better than resolutions that do not consider bounds. We consider CC, HC and SLC as representative features that can have conflicting goals to show that including features' goals (as constraints on attribute variable values) can improve the resolutions.

For example in the Table 5.10 transition scenario 1, the CC desired cruising speed is greater than the vehicle's current speed so that CC will cause the vehicle to accelerate. When the vehicle's speed reaches 90 km/h, there will be a competition between the CC and SLC. CC and SLC have different target speeds, if they issue actions only when they want to change the vehicle's speed, then the features will intermittently issue actions, the resolution will continuously alternate between their actions, and the vehicle speed will continuously accelerate and decelerate around the threshold of 90 km/h. If instead features constantly assert their goals as constraints rather than only asserting actions when features want to change the vehicle's behaviour, then the resolution module can take the features' goals into account in every execution step, thereby ensuring stability in the resolution module's

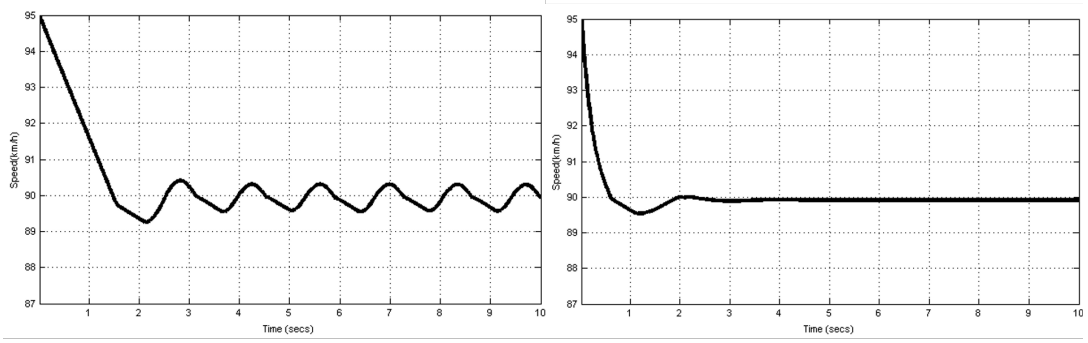

Figure 5.21: Improvement in the quality of resolutions due to the inclusion of constraints on attribute variable values in the resolution strategy (Transition Scenario 2)

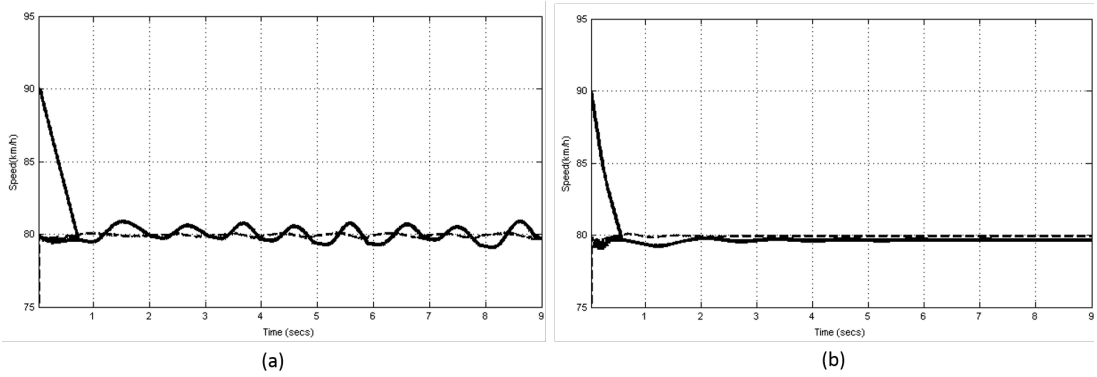

Figure 5.22: Improvement in the quality of resolutions due to the inclusion of constraints on attribute variable values in the resolution strategy (Transition Scenario 3)

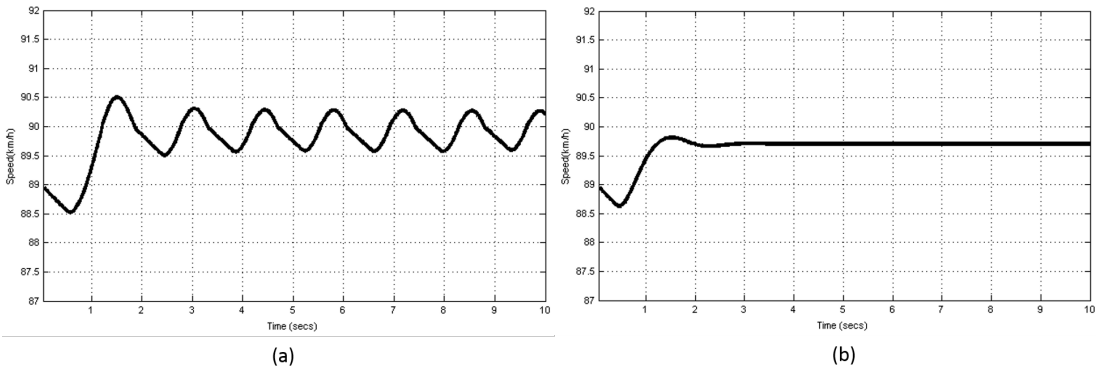

Figure 5.23: Improvement in the quality of resolutions due to the inclusion of constraints on attribute variable values in the resolution strategy (Boundary Scenario)

outputs over a sequence of actions.

The transition scenario 2 is similar to the transition scenario 1 except that initially the vehicle is already violating the speed limit. Thus, in this scenario, the resolution initially favours SLC's actions rather than CC's actions. But once the vehicle speed is reduced to 90 km/h, the output of resolved actions again experiences the thrashing of speeding and slowing down around the threshold value of 90 km/h (see the simulation results in Figure 5.21 (a)) – unless the features assert constraints on attributes as well as actions (see the simulation results in Figure 5.21 (b)).

The simulation results for the transition scenario 3 are shown in Figure 5.22. Initially, both SLC and HC want the vehicle to slow down but at different rates. Once SLC's speed limit is achieved and HC's safe distance to the preceding car is met, CC will actively speed up the vehicle whenever its speed is below 80 km/h – unless the features continuously asserts their constraints as well as actions (see the simulation results in Figure 5.22 (b)). This scenario shows the improvement in resolution-module outputs when the resolution strategy considers derivatives as well as attribute constraints. It would be very complex to consider all of the possible situations in which these three features can interact, and all of specific resolutions to those interactions; whereas the default resolution produces acceptable results in different scenarios.

The boundary scenario is similar to the transition scenario 1, except that CC's desired speed and the speed limit are off by 1 km/h. Both CC and SLC try to achieve a vehicle speed of about 90 km/h, but CC tries to speed up the vehicle to 91 km/h, whereas SLC tries to keep the vehicle's speed at or below 90 km/h. This scenario shows that without feature

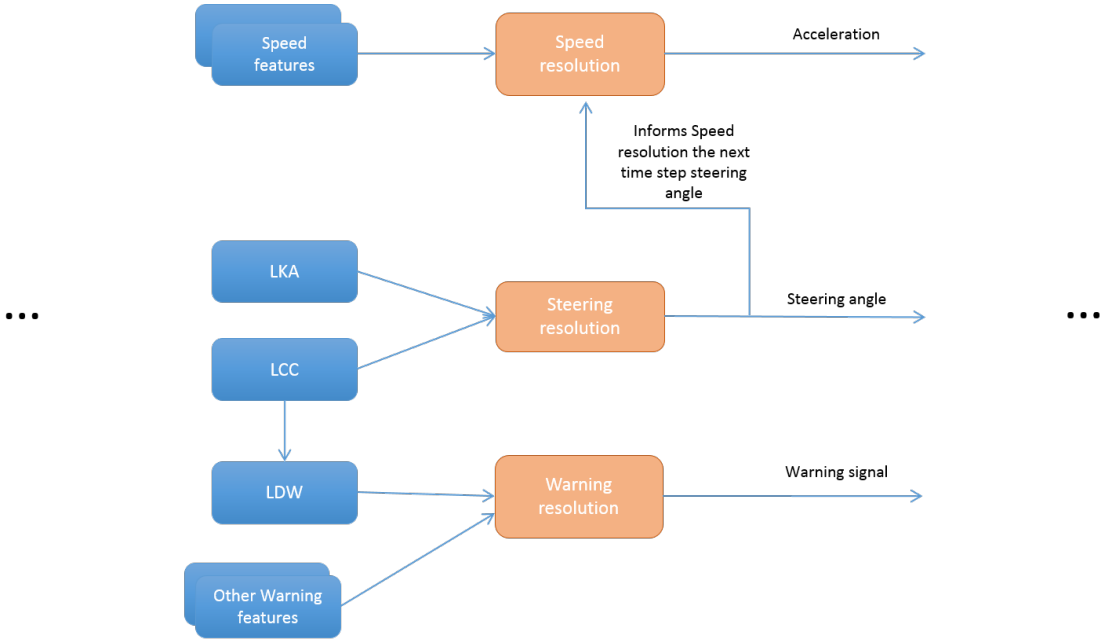

Figure 5.24: Example showing communications between resolution modules

constraints on attribute variable values, the resolution module will produce outputs that thrash around the threshold speed of 90 km/h, even when the difference in the features' desired speed is small. However, the amplitude of the thrashing is less than in the transition scenarios. The comparison can be seen in the Figure 5.23.

5.4.3 Resolving Feature Interactions between Coupled Variables

This subsection assesses whether communication between resolution modules can improve the quality of outputs of resolution modules for coupled attribute variables. In order to give a basic idea of how data flows in the current architecture, we depict a concrete example of coupled attribute variables and communications between their resolution modules in Figure 5.24. In this example, the steering resolution module informs the speed resolution module of the resolved value of the steering angle so that the speed resolution module can ensure that its calculations of the next speed value not exceed our lateral acceleration criterion in Section 5.1.

Specifically, the speed resolution module calculates a tangential speed bound with the provided lateral acceleration and the predetermined steering angle (according to the equa-

Title	Initial Speed	CC Desired Speed	Description	Result
Normal Scenario	60 km/h	60 km/h	The vehicle enters a curve from straight road. Steering resolution outputs approximately 2° to keep the vehicle in center of the road.	No actions towards speed. See Figure 5.25.
Transition Scenario	80 km/h	80 km/h	The vehicle enters a curve from straight road. Steering resolution outputs approximately 2° to keep the vehicle in center of the road.	Decelerate the vehicle whenever the lateral acceleration is greater than 0.12g. See Figure 5.26.

Table 5.11: Resolving feature interactions - resolving feature interactions between coupled variables

tion 5.1), and uses the result as an upper bound on the resolved value of the vehicle speed.

The test scenarios (see in Table 5.11) aim to test whether resolutions of values of coupled variables can be coordinated by simply having the variables' resolution modules share outputs with each other.

In the normal scenario, the road curvature is 2°, therefore LKA will output a steering angle of about 2° to keep the vehicle in the center of the lane. Because the vehicle speed is always smaller than calculated speed bound, the resolution module does not need to adjust the speed in order to maintain the acceleration to 0.12 g or less. The relationship between steering angle, speed, and lateral acceleration in the simulation is shown in Figure 5.25.

However in the transition scenario, the lateral acceleration limit of 0.12 g (1.176 m/s^2) will be exceeded some time. To avoid exceeding this limit, the steering angle is unchanged and the speed is reduced. The relationship between the steering angle, speed, and lateral acceleration in the simulation is shown in Figure 5.26. We can see that in the figure, when the vehicle enters the curve, the lateral acceleration violates the criterion, so the speed resolution module reduces the speed according to the calculated speed bound. Again in the latter part of the scenario, the current architecture adjusts the speed because of the

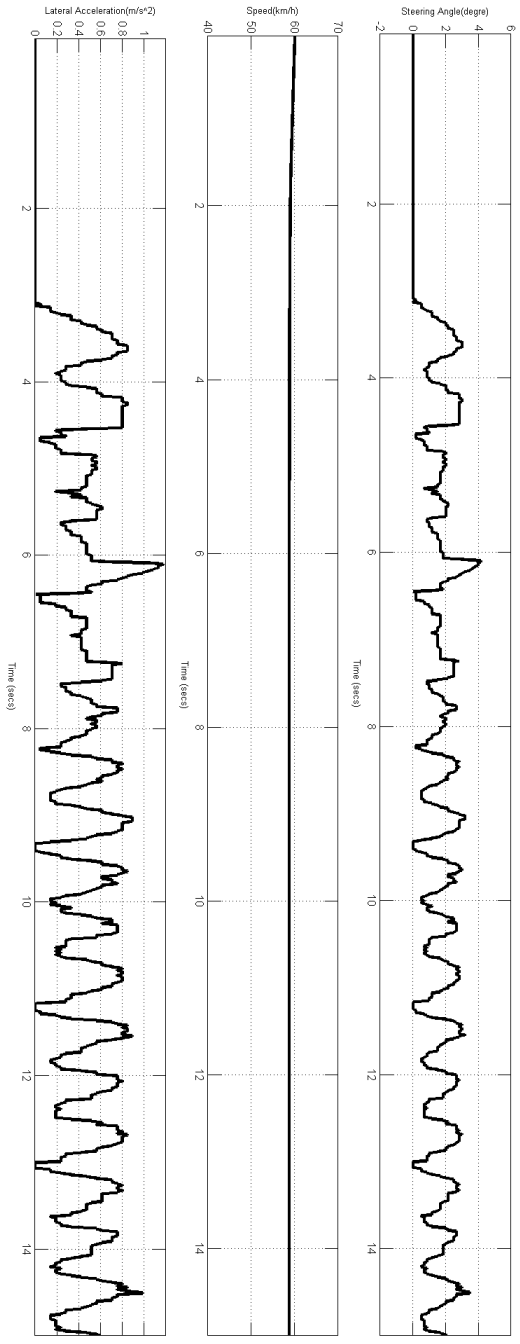

Figure 5.25: Constrain speed according to steering angle - Normal Scenario

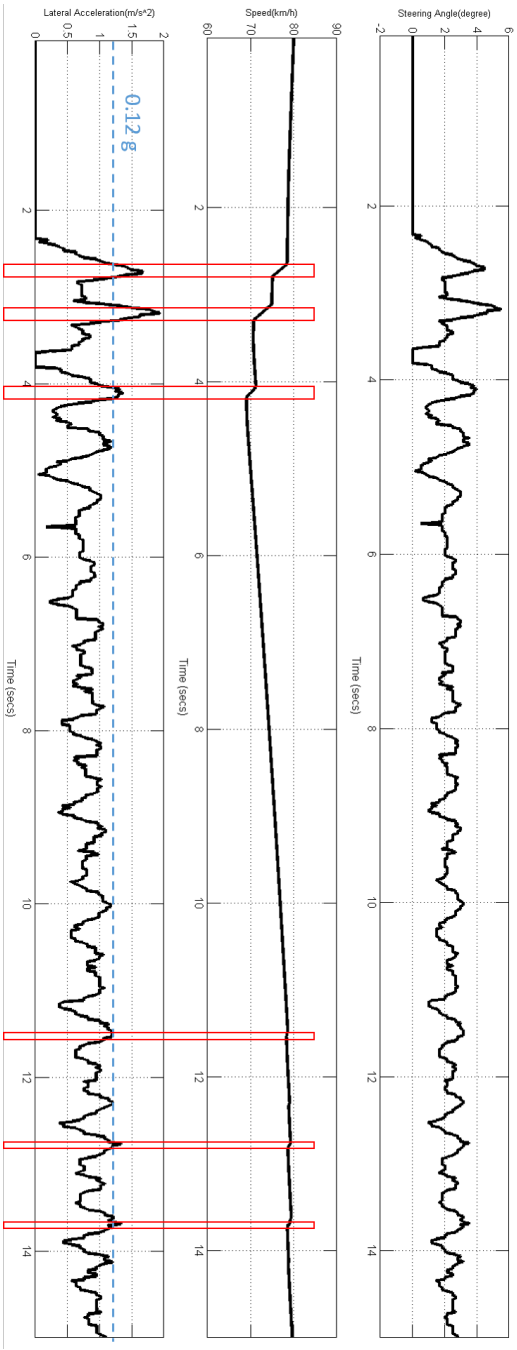

Figure 5.26: Constrain speed according to steering angle - Transition Scenario

steering angle. The adjustments are highlighted by red rectangles.

5.4.4 Stress Case Study

Recall the motivational scenario that we described in Section 2.1. We use this scenario as our stress case study, which shows how current architecture performs in the presence of multiple feature interactions.

The scenario comprises three phases. In phase one, the vehicle has a speed of 75 km/h and a CC desired speed of $100km/h$. The road has a speed limit of 90 km/h. This speed limit is detected by SLC and sent to the resolution module as a bound. Thus, this phase of the stress test assesses the usefulness of attribute bounds in the resolution decision.

In the phase two of the simulation, the vehicle is cruising at the speed limit (speed 90 km/h), and a car with a speed of 80 km/h suddenly appears in front of the vehicle. Because the 'safe distance' in front of the vehicle is violated, a warning is issued by CW and HC is activated. HC sends actions to the speed resolution module: a target (negative) acceleration and a speed bound. When the space between the vehicle and the preceding car is longer than the 'safe distance', the HC continues to send a speed bound to the speed resolution module to ensure that a safe distance is maintained. This bound is continuously updated as the distance between the vehicle and the preceding car changes. At this point, the driver decides to change lanes using LCC, but there is a blue car in the blind spot of the vehicle, which activates LCA, which in turn prevents the lane change.

In phase three of the simulation, the blue car leaves the blind spot, and the warning feature LCA stops sending signals. The driver needs to signal again, signifying he is still interested in changing lanes, before LCC will activate. After the lane change is complete and there is no longer a car directly in front of the vehicle, HC stops issuing speed action, and the vehicle smoothly accelerates to the speed limit of 90 km/h.

The simulation results can be seen in Figure 5.27. As we can see in the figure, the current architecture provides a conflict free, smooth and continuous resolution to conflicts. The resolutions keep the vehicle behaving acceptably and satisfying (or quickly re-establishing) all the criterion.

5.5 Discussion

Our goal in using a semi-autonomous vehicle simulator as our test bed was to verify whether the current architecture could actually solve the key research questions raised at the be-

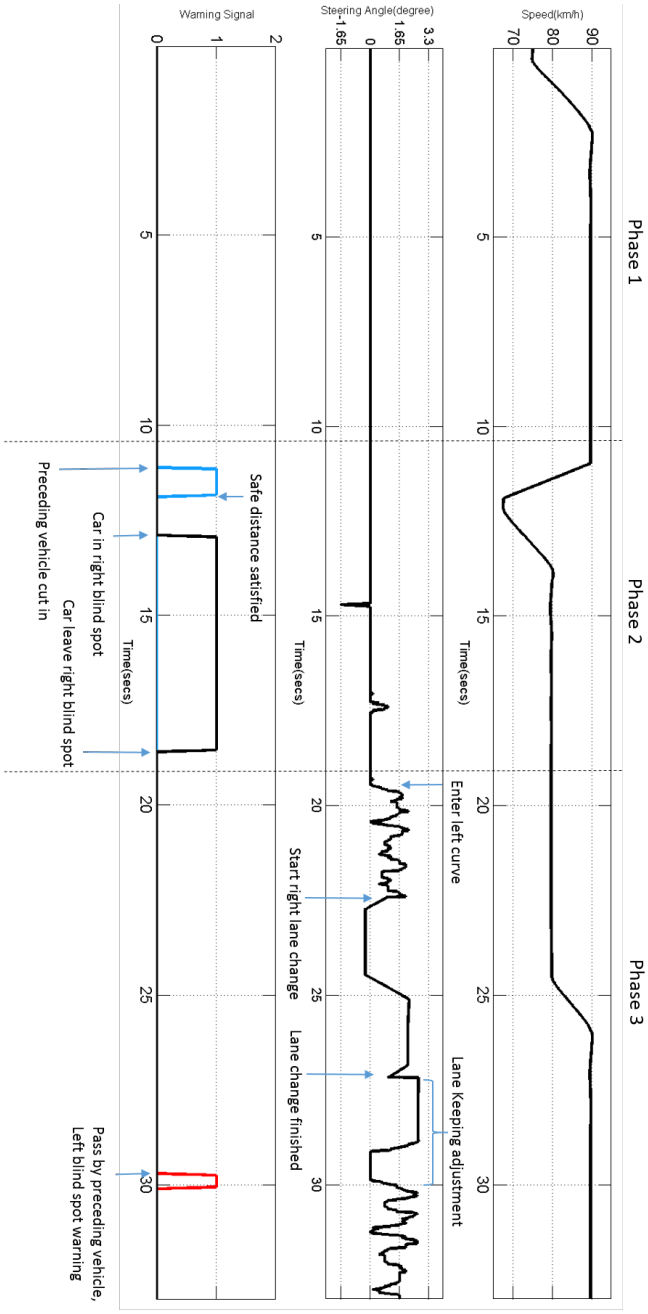

Figure 5.27: Result plot for stress case study

ginning of Chapter 4 when applied to a dynamic system.

To ensure the stability of resolution-module outputs over a sequence of actions rather than a single action step, we allowed different types of input to the resolution modules. Allowing derivatives acknowledges dynamics which can solve conflicts among features that have the same target value for an attribute variable but different changing rates. Our case study shows that by considering derivatives in the resolution strategies, the current architecture provides better resolutions. By allowing attribute bounds as constraints, the current architecture prevents feature interactions from thrashing around a threshold value and thus ensures the stability of the vehicle's behaviour. Enriching the features' action language (i.e., the resolution modules' input language) not only preserves the preliminary architecture's advantages, but also enables a smoother resolution output. It is worth noting that the current architecture considers only attribute bounds as constraint goals. However, features such as CC can have other constraint goals such as target values. In the current architecture, if the target values are currently satisfied, this could lead to the feature issue no action and to the other features and resolution module periodically deviating from the target resulting in thrashing. Therefore, to allow target-value goals in the current architecture requires further investigation. We will leave this to future work.

The extension of adding communications between resolution modules for coupled variables helps the current architecture to provide better outputs (for example, reducing the vehicle's speed according to the lateral acceleration limit to keep the driver more comfortable). The preliminary architecture focused on feature interactions involving one controlled variable and ignored possible interactions between different variables. Communications between resolution modules enables one resolution module to take another resolved attribute variable value into account, resulting in resolutions that account for the coupling among variables.

We have concrete resolution strategies for vehicle speed and for warnings, and they have been tested using a number of features and test scenarios and found to perform well. However, our resolution strategy for steering has been less thoroughly assessed. Although in our case study it performs acceptably, we had only two steering features, and features that perform a task like LCC are treated differently from normal features like LKA, because task features cannot be interrupted in the course of a task. The scarcity of steering features in our case study may be a threat to validity of our steering resolution strategy. We leave further research on a steering-resolution strategy to future work. What's more, fairness issue within task features were not investigated. In the current architecture, we only consider the first task feature and block the rest task feature before the mutex in the resolution module is released. This is the simplest way to resolve conflict interactions among task features, but fairness with respect to task features requires further research.

The current architecture and evaluation are based on MATLAB Simulink, which is a real-time synchronous system, meaning that all the calculations are always successfully completed within one time step (before the next time step begins). The frequency in the simulator is 100 Hz which means that in each time step, all the calculations are considered done in 0.01 s. This constraint can be relieved by extending wall time of the experiment. For example, one of our experiments takes 10 min to run a 20 s simulation. However, this situation is a big problem in real-world applications if an execution step cannot finish within 0.01s. Another timing issue that arises when the current architecture is extended to real world realization is that different features may have different calculating frequencies, which may cause feature outputs to arrive at the resolution modules asynchronously.

5.5.1 Threats to Validity

The features used have mostly been implemented and tested on our own. The feature logics within these features come from academic papers or other online sources. However, our feature implementations are mainly focused on feature requirements for semi-autonomous vehicles in a simulator, and we did not implement features on an industrial scale. Therefore, there is a good chance that differences between our implemented features and a real industrial vehicle's features could affect the way features interact. These probable differences threaten the validity of the current architecture. Moreover, as described earlier, we have 15 features involved in our case study, but only 2 steering features and only 1 task feature. There is insufficient evidence that our resolution strategies for accommodating task features is appropriate for a default resolution scheme. Further work in evaluating task features should be done in the future.

Second, the current architecture focuses on the automotive domain. We cannot say with confidence that the current architecture can be extended to other domains without more thorough testing. It is possible that resolutions in another domain would be more complex to implement. The conclusions about smooth and continuous resolutions should be backed up with future research on more comprehensive collections of features and across different domains.

Finally, the evaluation criteria were defined by the author of this thesis. To prove that no bias and subjective assumptions were introduced to conform the evaluation of the current architecture, more case studies and evaluations should be conducted by different developers.

Chapter 6

Related Work

Software engineers studied the feature interaction problem extensively and offered a variety of solutions for different problem domains [18] [22]. In particular, default resolution strategies have been proposed to tackle feature interaction in systems with a large number of features.

For example, Homayoon and Singh [30] proposed an approach, whereby a number of provided tables describe relations between two features. The status of one of the features determines whether a second feature should be activated or not. Cain [6] proposed an approach that relay the event message to a feature, and collect response from features, return one of the feature responses according to instructions. These approaches to resolve feature interactions have pros and cons respectively. As for the disadvantages side, they are mostly limited in their applicability to expanding systems. This is because the feature manager needs to be provided with (detailed) information on new features at design time. Clearly, this is a problem if the features are developed by multiple vendors [7].

However, there are five resolution strategies that are regarded as the most successful approaches to resolve feature interactions: priority-based resolution, precedence-based resolution, negotiation-based resolution, rolling back conflicts, and variable-specific resolution.

6.1 Priority-Based Resolutions

Priority-based resolutions drew great attention from research on resolving feature interactions. As its name implies, priority-based resolution resolves interactions according to

feature priority [11, 14, 16, 30, 19]. This resolution strategy requires a predefined total or partial order among features. In the event of a feature interaction, only the actions of the highest-priority feature are executed, blocking the behaviour of all other features. This obviously sacrificed behaviour of all the rest features, and is a win/lose resolution in that only the highest-priority requirements are satisfied in the case of a conflict. This is different from our approach since our approach takes into consideration all the features' behaviour into a combined resolution which is a win/win approach. Furthermore, when a developer wants to add, remove or modify a feature, he must also determine the priority of this feature with respect to the priorities of all other features.

Some priority-based approaches provide elaborate predefined resolutions. Laney et al. [23, 24] proposed a run-time prioritization resolution. Requirements are decomposed into features that are specified individually and then composed together. During feature composition, a developer must specify which aspects of a feature's behaviour may be relaxed in the event of a conflict. Interactions are then resolved at runtime by relaxing low-priority behaviours. Therefore, their approach will cause the Feature Interaction Problem: the need to analyze, resolve, and verify interactions may potentially require exponential effort to complete such tasks according to the number of features.

6.2 Precedence-Based Resolutions

Precedence-based resolution [4, 9, 18] strategies have some common aspects with priority-based resolutions, except that in precedence-based resolution features are serialized and execute sequentially in response to an event whereas in priority-based resolution low priority features are sacrificed in favor of high priority features. These kinds of resolution strategies require a predefined total or partial order of features according to precedence. It is obvious that for an n features system, there can be up to $n!$ possible precedence orderings, which is a considerable number.

Weiss [31] aims to ease the task of determining a precedence ordering by categorizing features and using automated feature interaction detection to identify an ordering in which features are less likely to conflict. Although these works do mitigate the work of identifying good feature precedence orders, the resulting resolutions still suffer from being coarse grained and being win/lose.

6.3 Negotiation-Based Resolutions

To find optimal resolutions at runtime rather than try to identify them all at design time, Griffeth and Velthuijsen [15] proposed an approach through automated negotiation. The basic idea is to provide acceptable alternative feature behaviours when feature conflicts occurred while maintaining feature's essential functionality. A negotiation process consists of a number of separate tasks: specification of policies, generating proposals, determining acceptability of proposals, and generating counter-proposals. This approach has been applied to multi-agent systems using situation calculus as the action language [29]. The biggest problem of negotiation-based resolution is that it requires multiple rounds of communication between negotiation agents that represent features before it can give out an ultimate resolution. However, for many feature-oriented systems, the time required to negotiate a mutually acceptable resolution exceeds time limits for an execution step. Our approach resolves an interaction in a single simple calculation (i.e., a simple variable-specific resolution strategy implemented in a resolution module).

6.4 Rolling Back Conflicts

Marples et al. [25] introduced a run time technique to resolve feature interactions, which is to rollback, or undo, previous actions when the system detects that multiple features are trying to modify the same variable. When such an interaction occurs, a resolver explores possible resolutions by communicating with all interacting features, to reach a stable, conflict-free state. The problem of such an approach is similar to that of negotiation-based resolutions: the exploration of possible resolutions requires lots of messages to be sent and received from the interacting features, which will cause timing issues in a system with timing constraints. Additionally, systems can be vulnerable because features may fail to participate in the communication process and will be terminated by the system.

Some approaches combine priority-based resolution and undoing conflicts, by preventing the activation of low-priority features, or terminating the lowest priority feature involved in a conflict [30]. This kind of approach can remove undesired feature interactions by undoing any behaviour that the low-priority feature contributed (or would have contributed) to the system. However, our approach preserve feature activations, and takes all feature actions into account regardless of the features' functionality and priority levels.

6.5 Variable-Specific Resolution

Compared to the previous related work, Bocovich and Atlee [5]'s approach views the feature interaction problem in an innovative way. Their approach enables programmable resolution strategies that are variable specific and feature agnostic. Such architecture enables new features to be easily integrated if they act on existing variables. It allows developers to specify appropriate win/win resolutions that do not require a total or partial ordering on features, thus resolutions can be the result of multiple features' contributions. And the architecture also enable runtime resolutions of interactions among ad-hoc feature combinations. Thus, it is possible to reduce the work done by the developer to be linear in the number of types of output variables modified by multiple features, rather than exponential in the number of features. The architecture they proposed can also support feature modularity therefore enabling independent feature development without feature interaction resolution work.

Chapter 7

Conclusions

In this thesis, we further refined the extended architecture to the current architecture, in order to produce a smooth sequence of resolutions. We also presented a simulation of the extended architecture and the current architecture on a semi-autonomous simulator to assess the quality of resolutions they produce in practice.

To promote smooth resolutions over consecutive execution steps, we extended the feature-action language to support richer descriptions of features' output commands: namely, attribute bounds and derivatives. For the features that require that attribute variables remain within a specific value range, the feature logic sends those boundary values to the resolution module. With this information, the resolution module can ensure that resolved values fall within these bounds. And for features with same target value of an attribute variable but within different time frames, allowing derivatives takes into account the features' desired rates of change of the attribute, therefore provides resolutions that satisfy more feature requirements.

To test the validity of the extended architecture and the current architecture, we implemented a set of features for the semi-autonomous vehicle simulator as a prototype. Those features are also used to test the performance of different default resolution strategies. The features came from Bosch Mobility Solutions and we implemented 15 of them. They were divided into three types for attribute variables, namely: speed, steering, and warning signal. During the evaluation, we were not able to find an existing criteria for evaluating the resolutions, therefore we defined our own criterion in 5.1. The current architecture and resolution modules performed acceptably with respect to these evaluation criteria. Based on the results of the case study, we concluded that the current architecture showed promise as an architectural approach to resolve feature interaction conflicts at runtime, such that

the system's overall behaviour is smooth and continuous.

This thesis has some possible directions to extend the work. First, more case studies with different system setups are needed. Currently, we have 15 features and 3 resolution modules. It is necessary to evaluate the current architecture with more features and resolution modules. Second, this thesis focuses on the automotive domain. This work can be extended to different domains and evaluation of the availability of the current architecture is desired. Third, some extensions in the extended architecture have not been evaluated (for example, actuator features and control logics) because of the limitations of the simulator. Assessment for these extensions is needed and should be in further studies. Finally, as mentioned in Section 5.5, the current architecture does not allow features to issue target values as constraints. This could lead to features issuing no action while other features deviating from the target value, which might cause thrashing on the target value. How to extend the input language to allow not only attribute bounds but also target values is also a good research question that deserves further study.

References

[1] Neta Aizenbud-Reshef, Brian T Nolan, Julia Rubin, and Yael Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[2] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[3] Mark Ardis, Nigel Daley, Daniel Hoffman, Harvey Siy, and David Weiss. Software product lines: a case study. *Software-Practice and Experience*, 30(7):825–47, 2000.

[4] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[5] Cecylia Bocovich. A feature interaction resolution scheme based on controlled phenomena. Master's thesis, 2014.

[6] Michael Cain. Managing run-time interactions between call-processing features (intelligent networks). *IEEE Communications Magazine*, 30(2):44–50, 1992.

[7] Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[8] E Jane Cameron and Hugo Velthuijsen. Feature interactions in telecommunications systems. *IEEE Communications Magazine*, 31(8):18–23, 1993.

[9] A Chavan, L Yang, K Ramachandran, and WH Leung. Resolving feature interaction with precedence lists in the feature language extensions. In *ICFI*, pages 114–128, 2007.

[10] Wonshik Chee and Masayoshi Tomizuka. Vehicle lane change maneuver in automated highway systems. *California Partners for Advanced Transit and Highways (PATH)*, 1994.

[11] Yi-Liang Chen, Stephane Lafortune, and Feng Lin. Modular supervisory control with priorities for discrete event systems. In *Conf. on Decision and Control*, pages 409–415, 1995.

[12] Krzysztof Czarnecki and Ulrich W Eisenecker. Intentional programming. *Generative Programming. Methods, Tools, and Applications*, 2000.

[13] Anthony Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multi-perspective specifications. In *Software Engineering– ESEC'93*, pages 84–99. Springer, 1993.

[14] Norbert Fritsche. Runtime resolution of feature interactions in architectures with separated call and feature control. In *FIW*, pages 43–63, 1995.

[15] Nancy D Griffeth and Hugo Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In *FIW*, pages 217–235, 1994.

[16] Jonathan D Hay and Joanne M Atlee. Composing features and resolving interactions. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 110–119. ACM, 2000.

[17] LL Hoberock. A survey of longitudinal acceleration comfort studies in ground transportation vehicles. *Journal of Dynamic Systems, Measurement, and Control*, 99(2):76–84, 1977.

[18] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, (10):831–847, 1998.

[19] Yinghua Jia. *Run-time Management for Feature Interaction*. PhD thesis, University of Waterloo, 2003.

[20] Alma L Juarez Dominguez. *Detection of feature interactions in automotive active safety features*. PhD thesis, 2012.

[21] Alma L Juarez-Dominguez, Nancy A Day, and Jeffrey J Joyce. Modelling feature interactions in the automotive domain. In *Proceedings of the 2008 international workshop on Models in software engineering*, pages 45–50. ACM, 2008.

[22] Mario Kolberg and Evan H Magill. A pragmatic approach to service interaction filtering between call control services. *Computer Networks*, 38(5):591–602, 2002.

[23] Robin Laney, Leonor Barroca, Michael Jackson, and Bashar Nuseibeh. Composing requirements using problem frames. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 122–131. IEEE, 2004.

[24] Robin Laney, Thein Than Tun, Michael Jackson, and Bashar Nuseibeh. Composing features by managing inconsistent requirements. *ICFI*, pages 129–144, 2007.

[25] Dave Marples and Evan H Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *FIW*, pages 115–134, 1998.

[26] John-Jairo Martinez and Carlos Canudas-de Wit. A safe longitudinal control for adaptive cruise control and stop-and-go scenarios. *IEEE Transactions on control systems technology*, 15(2):246–258, 2007.

[27] Linda Northrop and P Clements. Software product lines. *URL http://www. sei. cmu. edu/library/assets/Philips*, 4(05), 2001.

[28] Pourya Shaker. *A feature-oriented modelling language and a feature-interaction taxonomy for product-line requirements*. PhD thesis, University of Waterloo, 2013.

[29] Steven Shapiro and Yves Lespérance. Modeling multiagent systems with casl-a feature interaction resolution application. In *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 244–259. Springer, 2000.

[30] Harshavardhan Singh. Methods of addressing the interactions of intelligent network services with embedded switch services. *Communications Magazine, IEEE*, 26(12):42–46, 1988.

[31] P Ann Zimmer and Joanne M Atlee. Ordering features by category. *Journal of Systems and Software*, 85(8):1782–1800, 2012.

APPENDICES

Appendix A

Speed Feature Tests

Headway Control

Headway Control keeps vehicle within safe distance from the car ahead. In the test plots, dashed line stands for speed of preceding car.

1. Normal Scenario: Initial Speed: 90 km/h, Acceleration 2.5 m/s^2, Preceding Vehicle Speed 80 km/h, Safe Distance 30 m, Distance When Detected 45 m.
   In this case, our vehicle will slow down to 80 km/h before reaching safe distance. This scenario aims to test whether HC will guarantee vehicle keeps a safe distance when starting from a long distance.
   The speed and distance plot is shown in Figure A.1.

2. Transition Scenario: Initial Speed: 90 km/h, Acceleration 2.5 m/s^2, Preceding Vehicle Speed 80 km/h, Safe Distance 30 m, Distance When Detected 29 m (car cut in).
   In this case, because when the vehicle cut in, our speed is suddenly greater than the preceding car's speed, so before our car's speed is reduced to no bigger than preceding car's speed, we will violet safe distance a little bit. After some time, when our car's speed is smaller than preceding car's speed and we reach the safe distance, HC will stop deceleration. This scenario aims to test whether HC will guarantee that the vehicle keeps a safe distance when starting from a short distance.
   The speed plot is shown in Figure A.2.

3. Boundary Scenario: Initial Speed: 81 km/h, Acceleration 2.5 m/s^2, Preceding Vehicle Speed 80 km/h, Safe Distance 30 m(car cut in), Distance When Detected 31 m(car cut in).

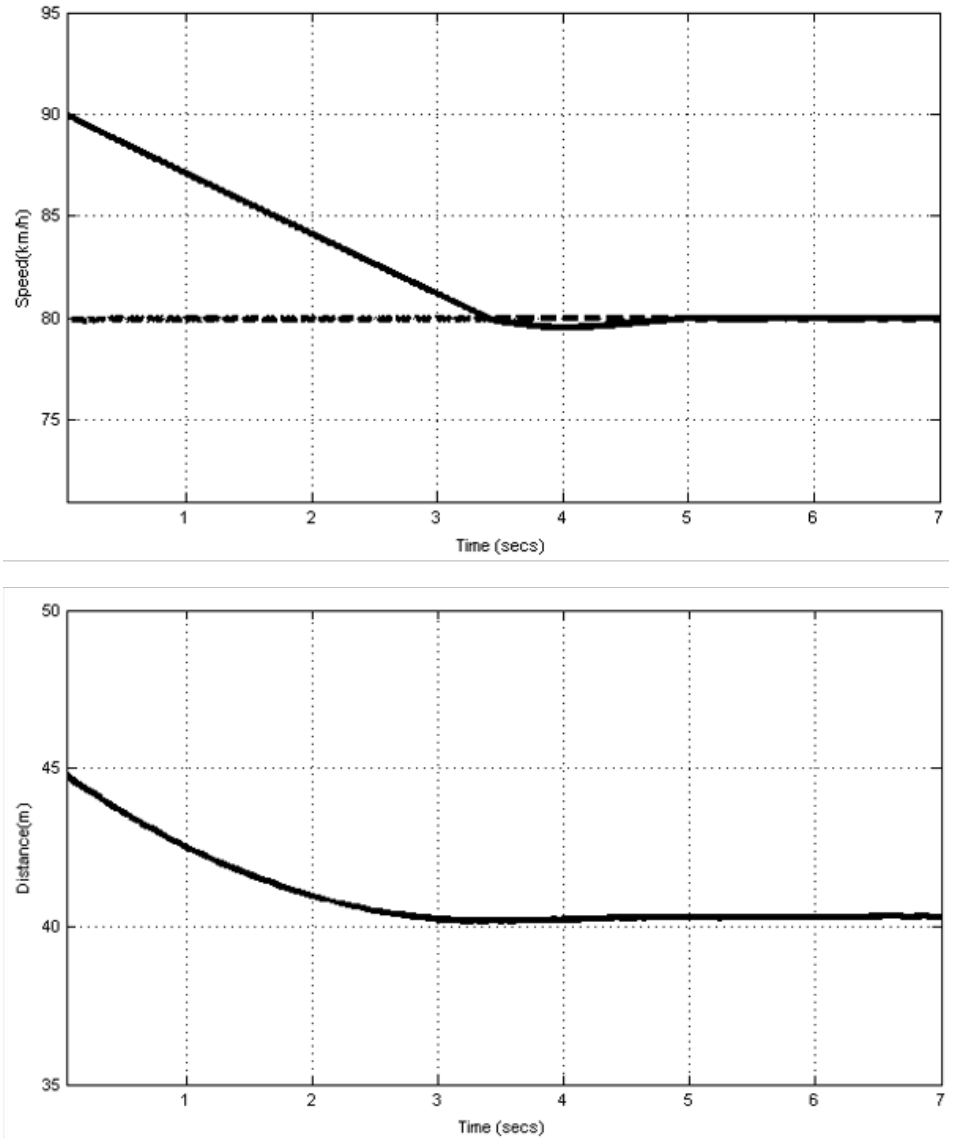

Figure A.1: Headway Control Normal Scenario

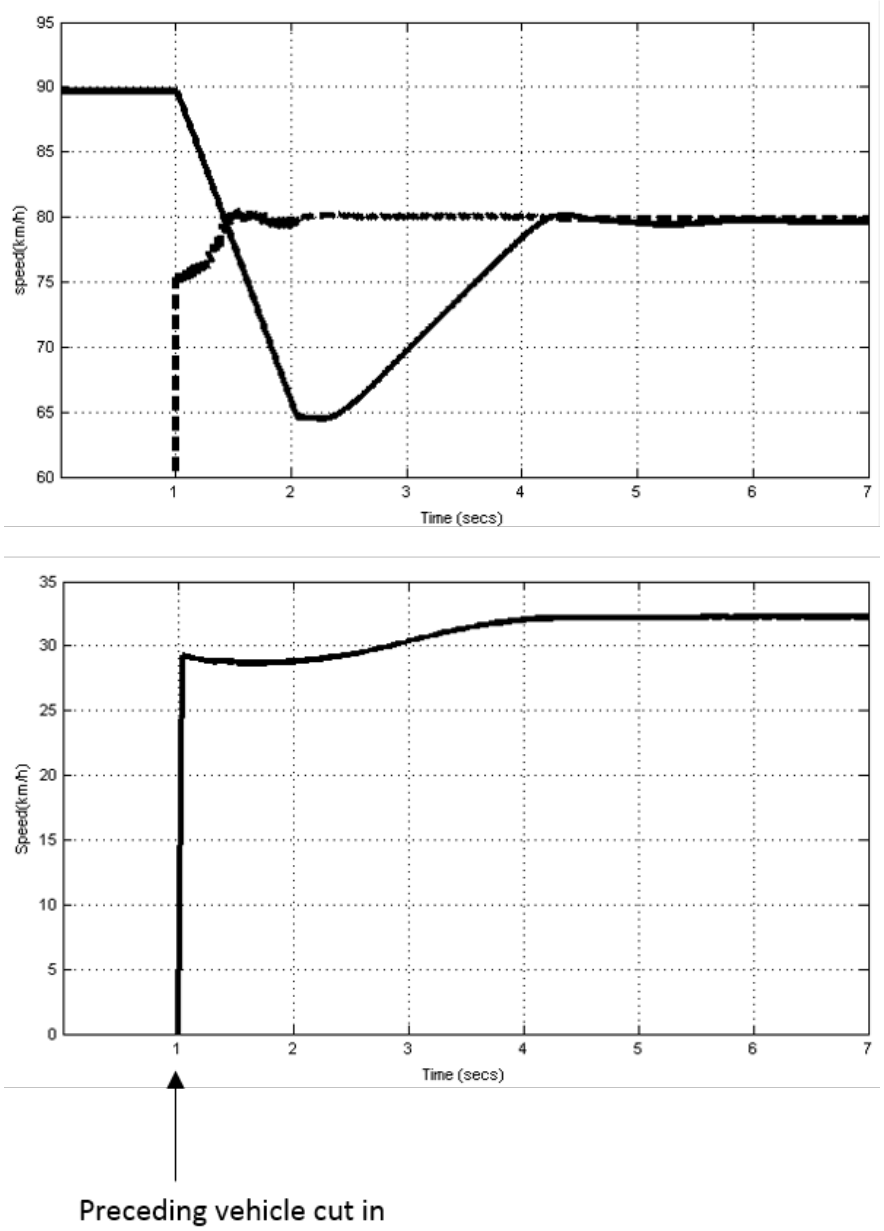

Figure A.2: Headway Transition Normal Scenario

This scenario aims to test performance of HC when speed and safe distance are close to boundary.
The speed plot is shown in Figure A.3.

We can see in all the scenarios that HC can guarantee vehicle stays within safe distance and the speed will not exceed preceding car's speed. It is worth knowing that in some cases like second scenario, HC will reduce the vehicle's speed to the value that is smaller than the preceding car's speed until the safe distance is reached, HC will stop working, since both distance and speed constraints are satisfied. If no acceleration is applied, the vehicle will remain running in the low speed. However, in the overall system, the CC will operate on the speed to ask the vehicle keep in the same speed as preceding car.

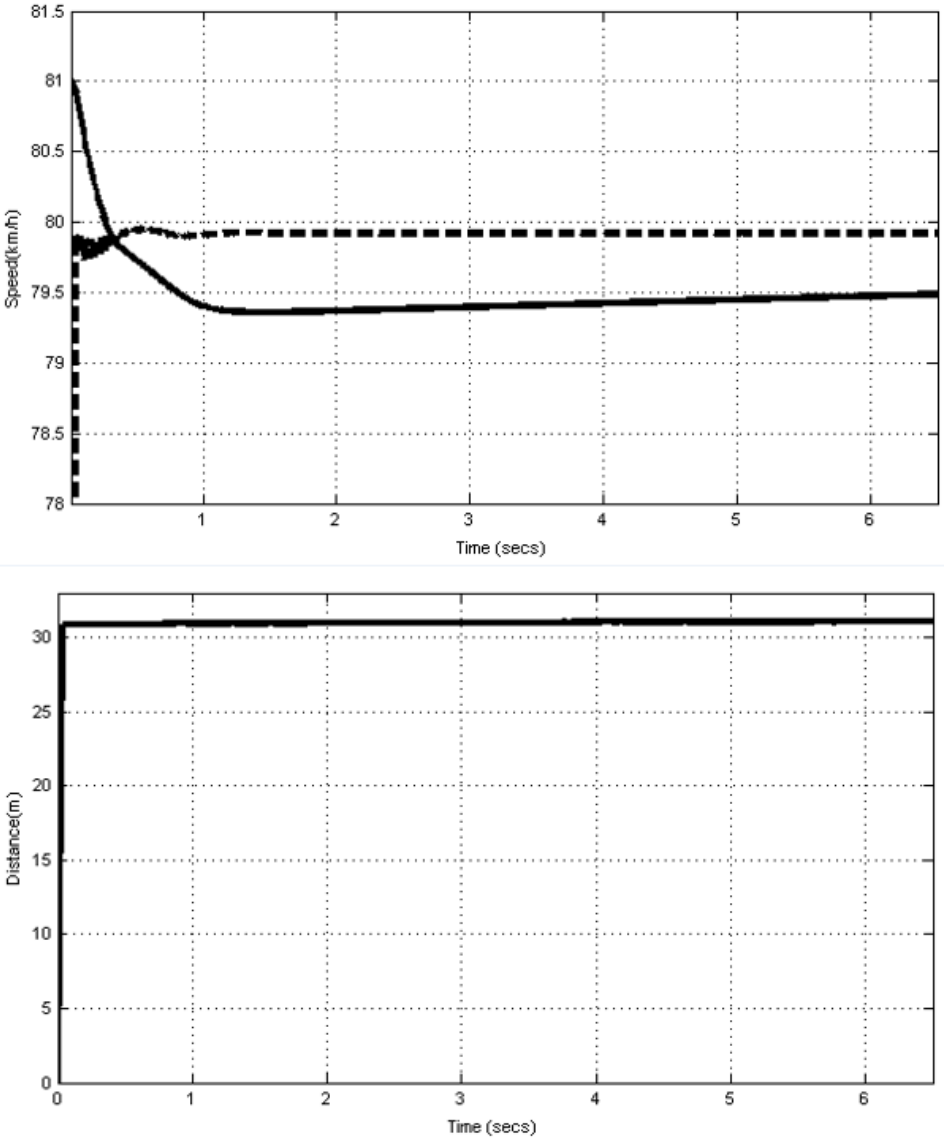

Figure A.3: Headway Control Boundary Scenario

Appendix B

Features with Dependencies

Some features are depend on other features. For example, LDW should not warn the driver when the LCC is doing lane-change. Therefore, when LCC is working, it should also send a notice to LDW to make LDW deactivate even though the vehicle is leaving the lane. The following scenarios will show how dependency between features work.

B.1 Dependency LDW and LCC

Normal Scenario 1 Car Speed 80 km/h, On a straight road, No obstacle in blind spot, Issue right lane change command.

The steering and warning signal are shown in Figure B.1.

We can see that during the whole experiment, even though the vehicle finished lane change action, LDW always stays off.

Normal Scenario 2 Car Speed 80 km/h, On a straight road, Initially place the vehicle on the lane marker.

In this scenario, since LCC is not working and the car is leaving the lane, LDW should go off to inform the driver about the circumstance.

The behaviour picture and warning signal are shown in Figure B.2.

Since the vehicle is not doing lane-change and leaving the lane is not our desired action, LDW sends out warning signal until the vehicle gets back to lane.

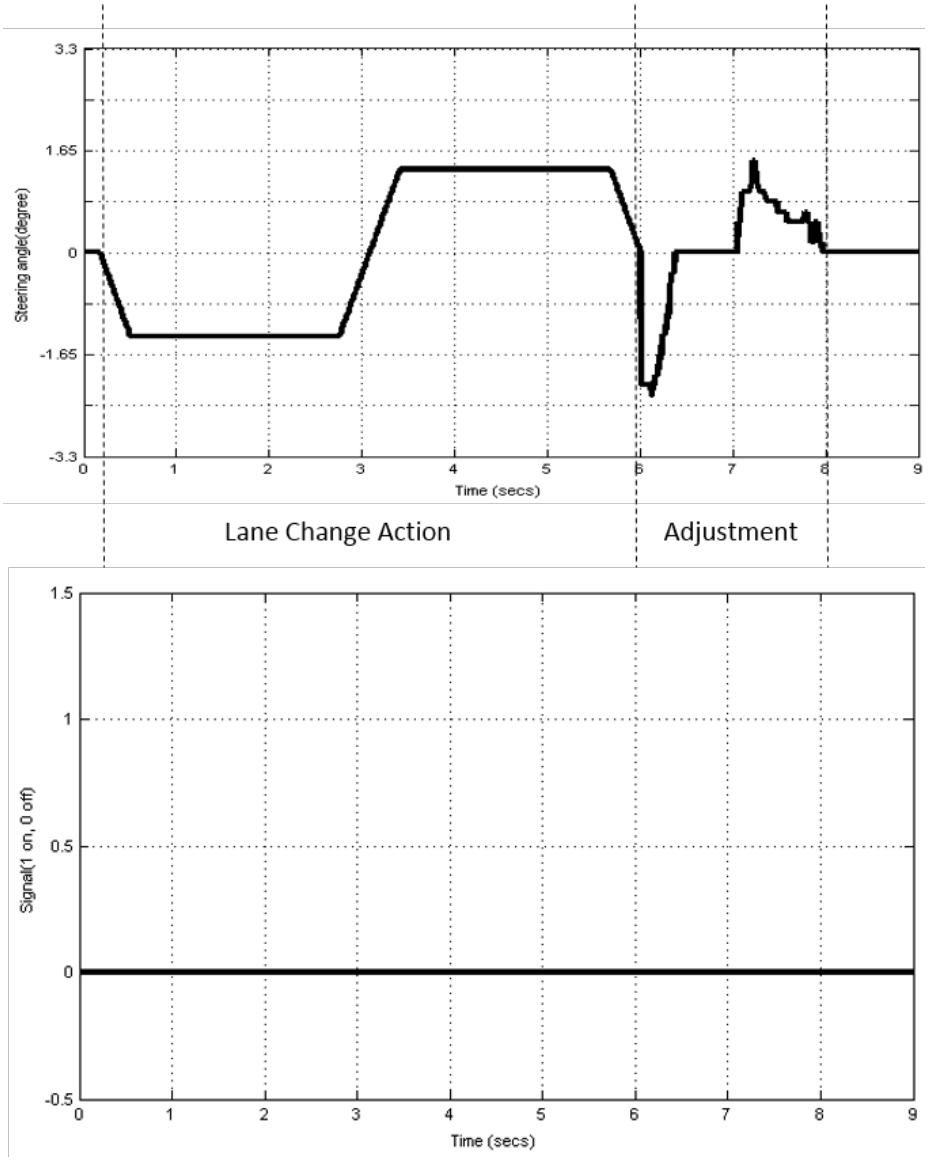

Figure B.1: Example of dependency between warning features and steering features - Normal Scenario 1

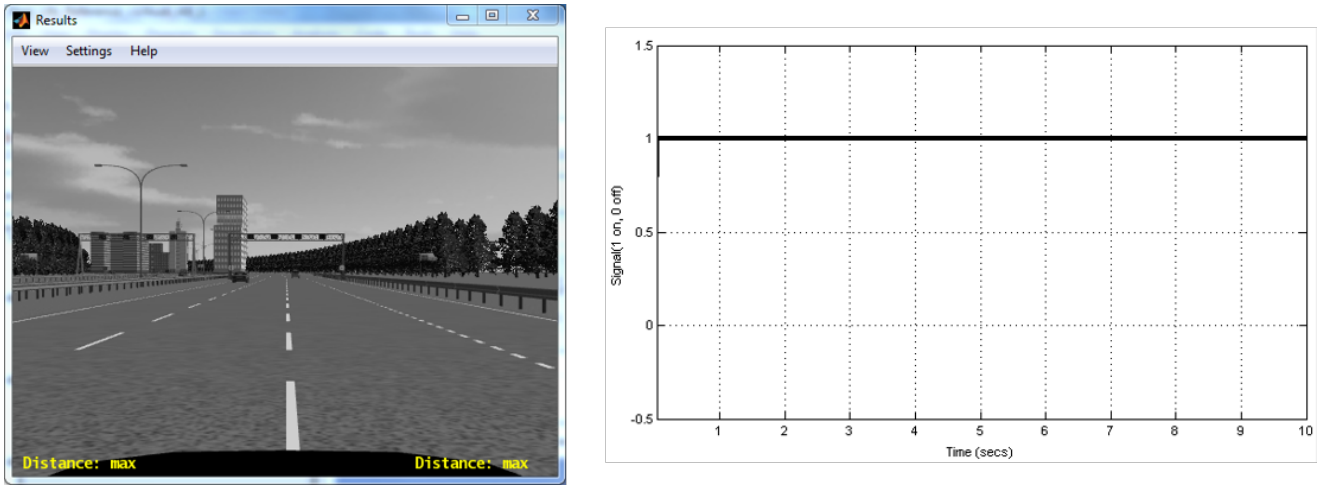

Figure B.2: Example of dependency between warning features and steering features - Normal Scenario 2

Appendix C

Feature Interactions Within One Variable

C.1 Speed Features Interaction

CC and SLC interactions will be introduced in section C.1.1. And feature interactions between SLC and HC are already discussed in section allowing derivatives 5.4.2.

C.1.1 CC-SLC

1. Normal Scenario: Initial Speed 75 km/h, Acceleration 0 m/s^2, Speed Limit 100 km/h, Cruise Control Desired Speed 90 km/h.
   This scenario aims to test whether resolution module can resolve feature interactions between SLC and CC when speed limit is bigger than CC desired speed.
   The speed plot is shown in Figure C.1.

2. Transition Scenario: Initial Speed 75 km/h, Acceleration 0 m/s^2, Speed Limit 90 km/h, Cruise Control Desired Speed 100 km/h.
   This scenario aims to test whether resolution module can resolve feature interactions between SLC and CC when speed limit is smaller than CC desired speed.
   The speed plot is shown in Figure C.2.

3. Boundary Scenario: Initial Speed 91 km/h, Acceleration 0 m/s^2, Speed Limit 90 km/h, Cruise Control Desired Speed 90 km/h.

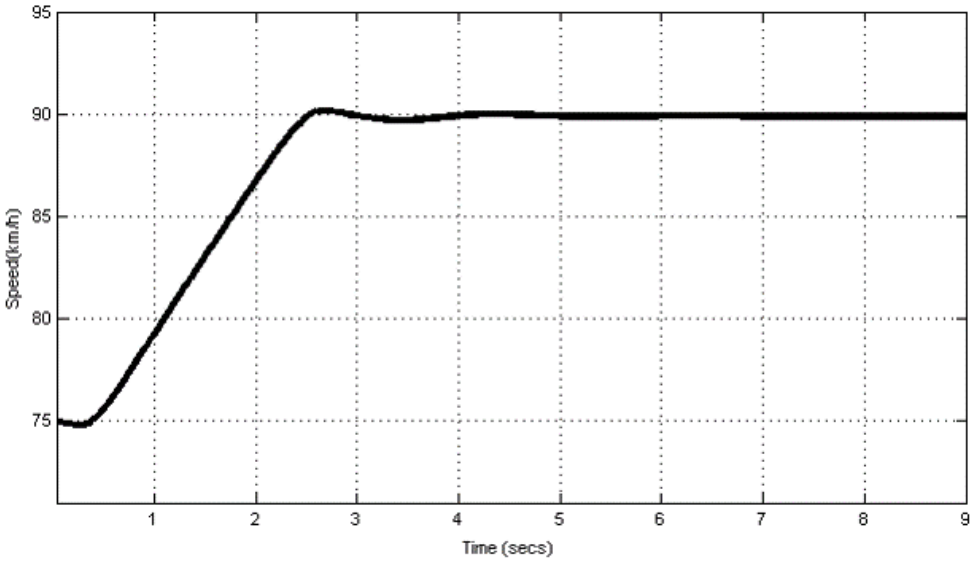

Figure C.1: SLC and CC interaction Normal Scenario

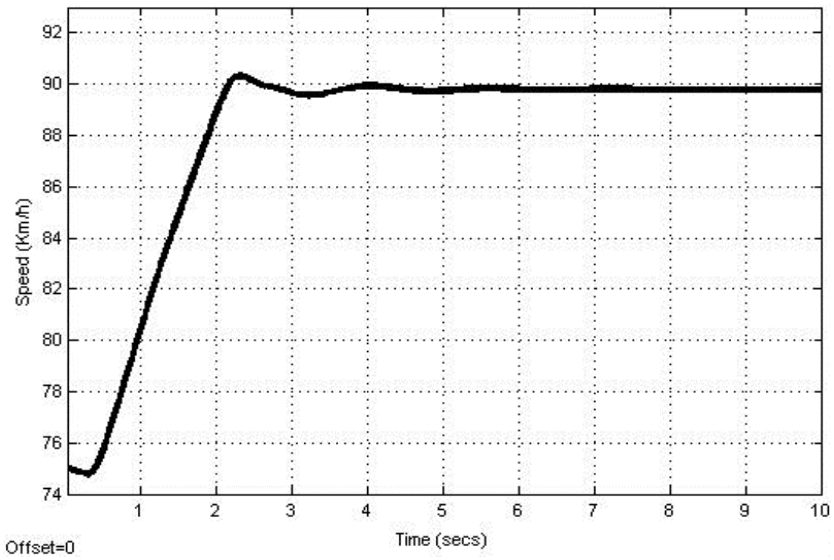

Figure C.2: SLC and CC interaction Transition Scenario

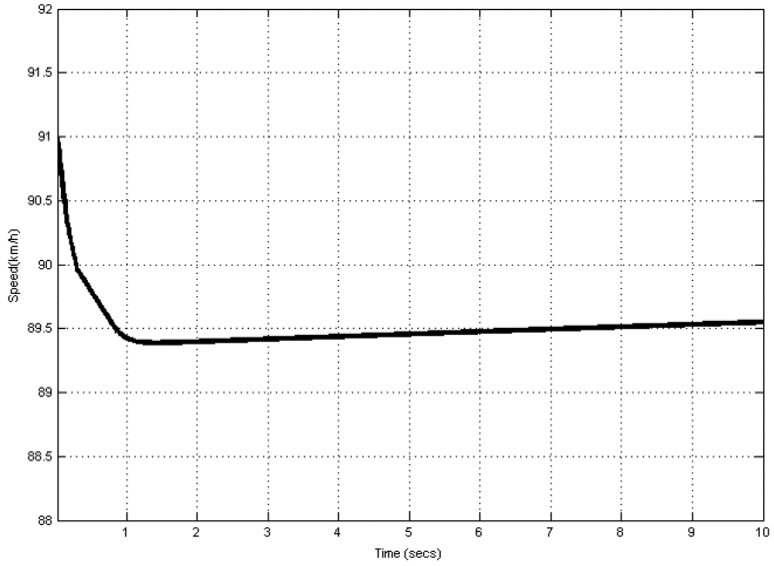

Figure C.3: SLC and CC interaction Boundary Scenario

This scenario aims to test whether resolution module can resolve feature interactions between SLC and CC when speed limit is smaller than CC desired speed.
The speed plot is shown in Figure C.3.

C.2 Steering Features Interaction

In this section, we introduce feature interactions between steering features. Steering features interactions happens between LKA and LCC. Since implementation of LCC on curve road is not idea enough, so errors may apply in our test results.

1. Normal Scenario: Car Speed 80 km/h, Straight Road, Another car in right blind spot, Issue right lane-change signal.
   This scenario aims to test whether LCC will start a lane-change action when detected an obstacle in blind spot.
   The steering plot is shown in Figure C.4. We can see that in the steering plot, no steering action will be taken in order to keep the vehicle safe.

2. Transition Scenario 1: Car Speed 72 km/h, Straight Road, no car in blind spot, Issue right lane-change signal.

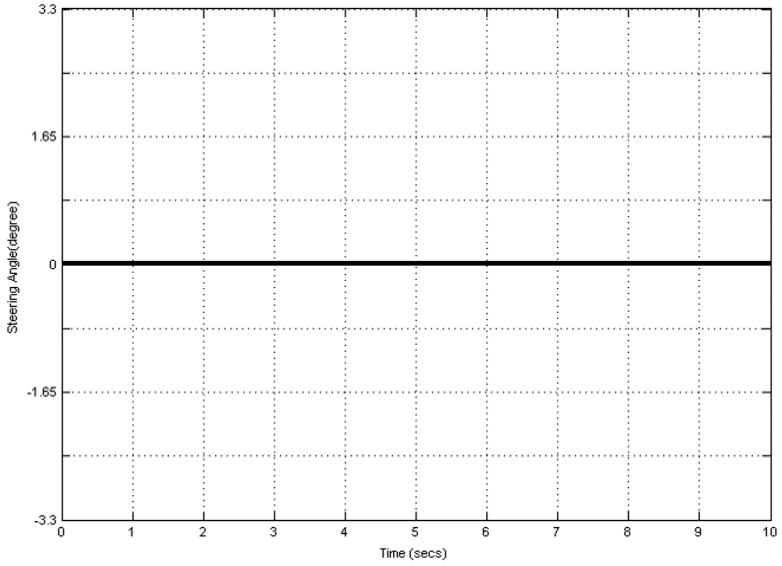

Figure C.4: Lane Changing Control Normal Scenario

This scenario aims to test whether LCC will successfully finish a lane-change action on a straight road when no obstacle is in the blind spot.
The steering plot is shown in Figure C.5.

3. Transition Scenario 2: Car Speed 72 km/h, Straight Road, no car in blind spot, Issue left lane-change signal.
   This scenario aims to test whether LCC will successfully finish a lane-change action on a straight road when no obstacle is in the blind spot.
   The steering plot is shown in Figure C.6.

4. Transition Scenario 3: Car Speed 72 km/h, Left Curve Road, no car in blind spot, Issue Right lane-change signal.
   This scenario aims to test whether LCC will successfully finish a lane-change action on a curve road to do lane change in different direction when no obstacle is in the blind spot.
   The steering plot is shown in Figure C.7.

5. Transition Scenario 4: Car Speed 72 km/h, Right Curve Road, no car in blind spot, Issue Left lane-change signal.
   This scenario aims to test whether LCC will successfully finish a lane-change action on a curve road to do lane change in different direction when no obstacle is in the

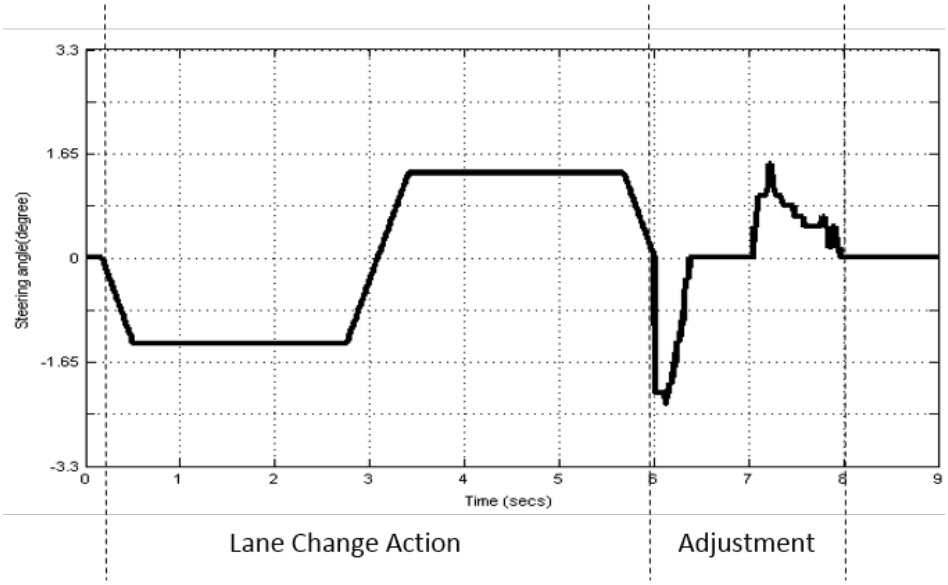

Figure C.5: Lane Changing Control Transition Scenario 1

blind spot.
The steering plot is shown in Figure C.8.

6. Transition Scenario 5: Car Speed 72 km/h, Right Curve Road, no car in blind spot, Issue Right lane-change signal.
   This scenario aims to test whether LCC will successfully finish a lane-change action on a curve road to do lane change in same direction when no obstacle is in the blind spot.
   The steering plot is shown in Figure C.9.

7. Transition Scenario 6: Car Speed 72 km/h, Left Curve Road, no car in blind spot, Issue Left lane-change signal.
   This scenario aims to test whether LCC will successfully finish a lane-change action on a curve road to do lane change in same direction when no obstacle is in the blind spot.
   The steering plot is shown in Figure C.10.

As we can see in test plots, the Lane Changing Control will not perform a lane-change action even the lane-change command is issued if there is another car in the blind spot. The blind spot detection signal is received from Lange Change Assist. And if there is no obstacle in the blind spot, LCC will finish lane-change action under command.

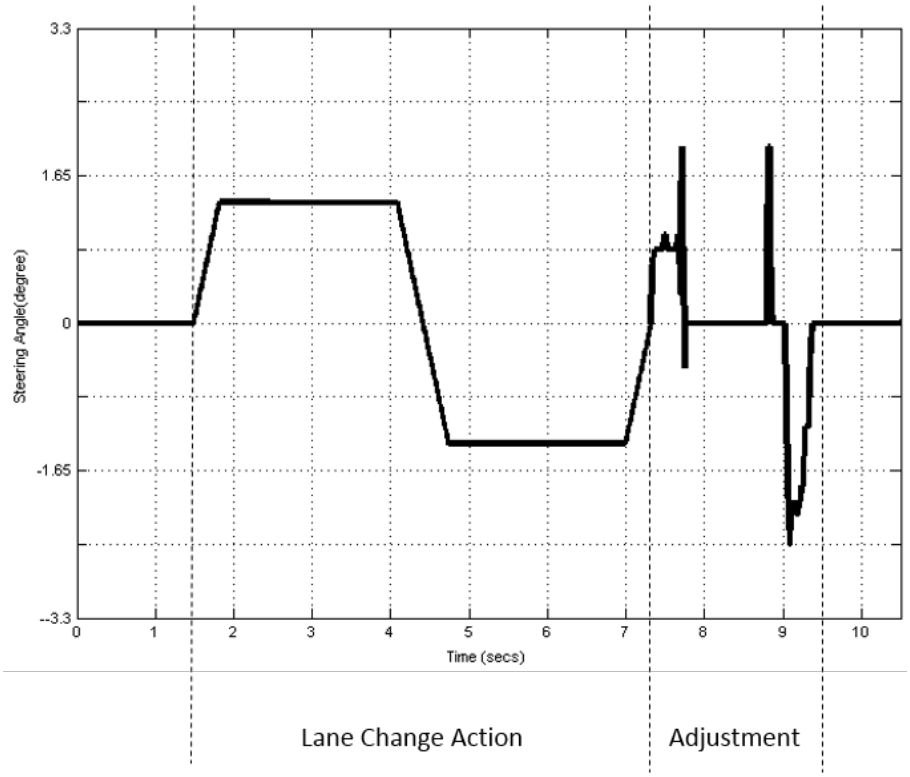

Figure C.6: Lane Changing Control Transition Scenario 2

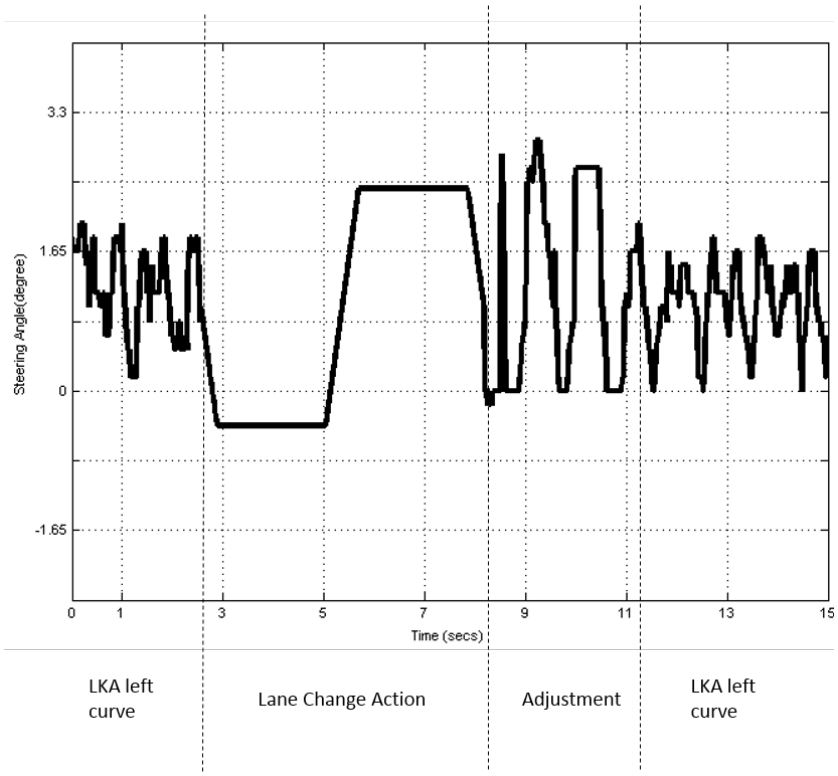

Figure C.7: Lane Changing Control Transition Scenario 3

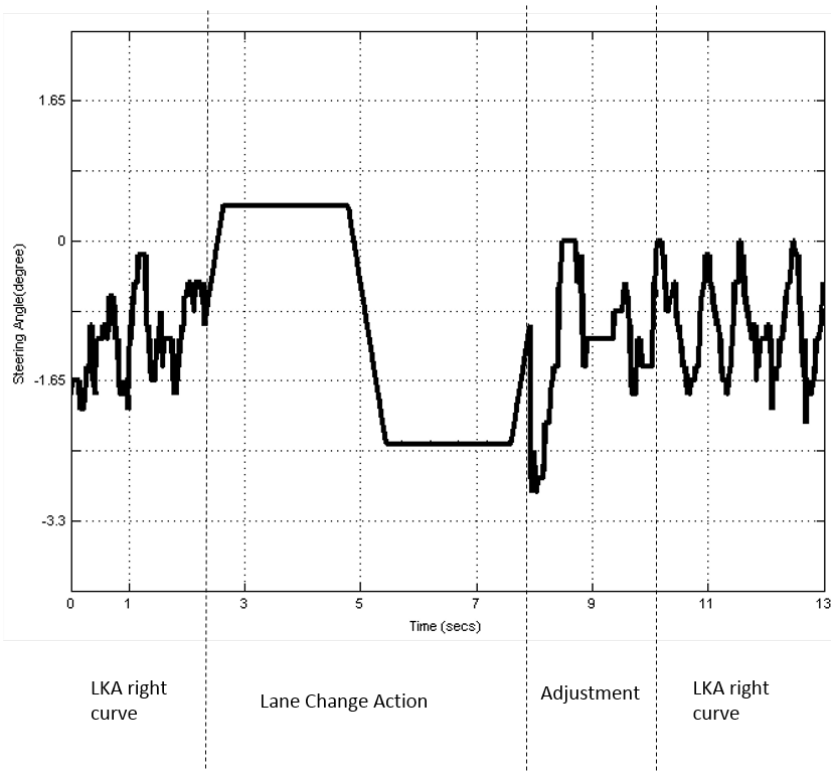

Figure C.8: Lane Changing Control Transition Scenario 4

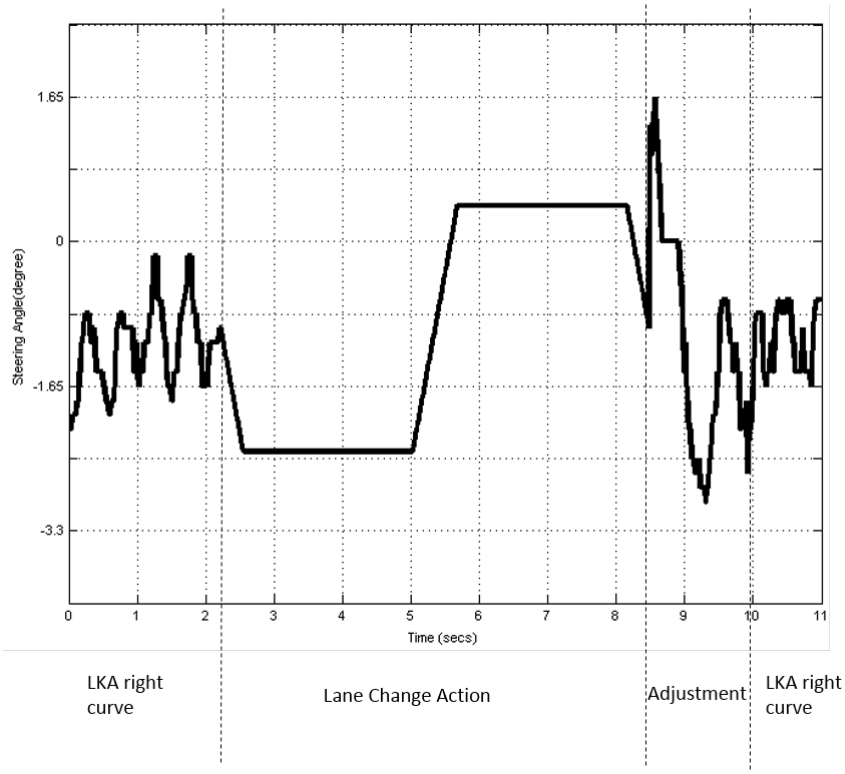

Figure C.9: Lane Changing Control Transition Scenario 5

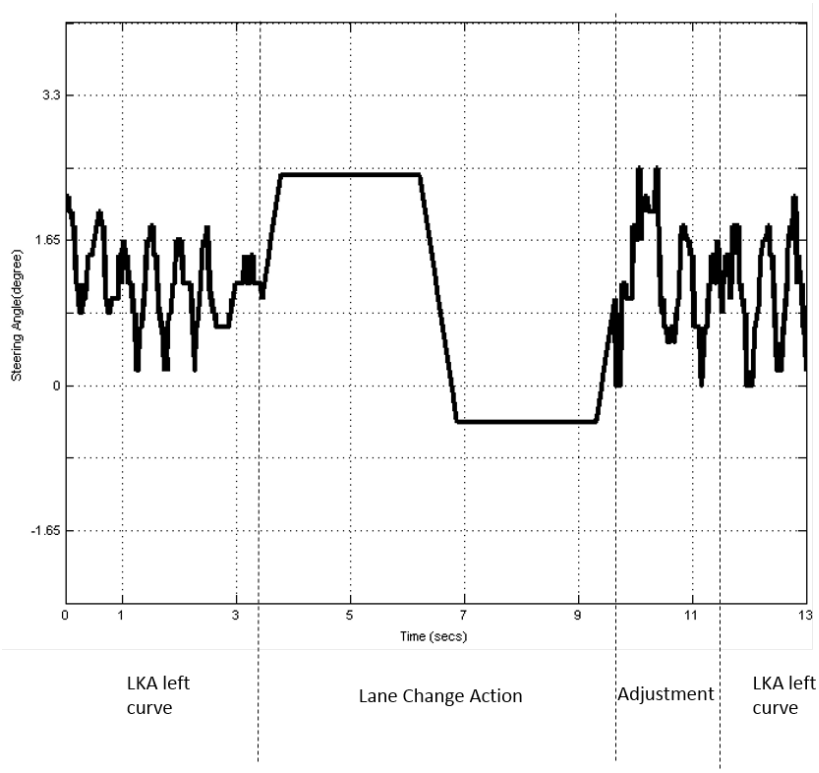

Figure C.10: Lane Changing Control Transition Scenario 6

However, our LCC's algorithm has some constraints to be mentioned. Firstly, we assume during the lane-change action, that the curvature of the road does not change. For example, if we start lane-change on a straight road, then during the whole lane-change process, the road should remain straight. Secondly, we assume during the lane-change action that the speed of the car does not change. However, these constraints can be violated in many test scenarios for solving feature interactions. Therefore, we may see some arithmetical error in computation of steering. Those arithmetical error will be fixed by LKA after changing the lane.