

A Mode-Based Pattern for Feature Requirements, and a Generic Feature Interface

by

David Dietrich

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, 2013

© David Dietrich 2013

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

Feature-oriented requirements decompose a system's requirements into individual bundles of functionality called features, where each feature's behaviour can be expressed as a state-machine model. However, state machines are difficult to write; determining how to decompose behaviour into states is not obvious, different stakeholders will have different opinions on how to structure the state machine, and the state machines can easily become too complex.

This thesis proposes a pattern for decomposing and structuring the model of a feature's behavioural requirements, based on modes of operation (e.g., Active, Inactive, Failed) that are common to features in multiple domains. Interestingly, the highest-level modes of the pattern can serve as a generic behavioural interface for all features that adhere to the pattern. The thesis proposes also several pattern extensions that provide guidance on how to structure the Active and Inactive behaviour of the feature.

The pattern was applied to model the behavioural requirements of 21 automotive features that were specified in 7 production-grade requirements documents. The pattern was applicable to all 21 features, and the proposed generic feature interface was applicable to 50 out of 58 inter-feature references. A user study with 18 participants evaluated whether use of the pattern made it easier than otherwise to write state machines for features and whether feature state machines written with the help of the pattern are more readable than those written without the help of the pattern. The results of the study indicate that use of the pattern facilitates writing of feature state machines.

Acknowledgements

Joanne M. Atlee

Cecylia Bocovich

Mom

Dad

Yao

Joyce

Table of Contents

List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Thesis Description	1
1.2 Contributions	3
1.3 Thesis Organization	4
2 Background	5
2.1 Feature-Oriented Requirements	5
2.2 Related Work	10
2.3 Existing Requirements Documents	12
3 The Pattern	15
3.1 Pattern	16
3.1.1 The Inactive extension	19
3.1.2 The Active state extension	24
3.2 Interface	31
3.3 Rational DOORS Templates	33
3.3.1 Cruise Control Example	37
3.4 Pattern Summary	38

4	Case Study	39
4.1	Utility of the Pattern	39
4.1.1	Example — Adaptive Cruise Control	43
4.1.2	Example — Heating, Ventilation, and Air Conditioning	45
4.2	Generality of the Public Interface	52
4.3	Threats to Validity	54
4.4	Case Study Summary	55
5	User Study	57
5.1	Performing the Study	57
5.2	Tutorial	58
5.3	Main Study	59
5.3.1	Participant Background	60
5.3.2	State-Machine Comprehension	63
5.3.3	State-Machine Modelling	71
5.3.4	Participant Confidence	77
5.3.5	Timing Results	78
5.4	Threats to Validity	79
5.5	User Study Summary	80
6	Conclusion	83
	References	85
	Appendices	91
A	Catalogue of Case Study Models	91
A.1	Electronic Braking Features	92
A.1.1	Automatic Braking (AB)	92

A.1.2	Anti-lock Braking System (ABS)	93
A.1.3	Active Trailer Stability Assist (ATSA)	94
A.1.4	Brake Assist (BA)	95
A.1.5	Brake Cleaning (BC)	96
A.1.6	Electric Park Brake (EPB)	97
A.1.7	Enhanced Traction System (ETS)	98
A.1.8	Hill Hold (HH)	99
A.1.9	Traction Control System with Electronic Stability Control (TCS_ESC)	100
A.1.10	Competitive Traction Control System with Electronic Stability Control (Competitive_TCS_ESC)	102
A.1.11	Manual Park Brake (MPB)	104
A.2	Freeway Limited Ability Autonomous Driving Features	105
A.2.1	Forward Collision Alert (FCA)	106
A.2.2	Lane Centring Control (LCC)	107
A.2.3	Lane Keep Assist (LKA)	108
A.2.4	Lane Change Control (LXC)	110
A.2.5	Road Change Alert (RCA)	112
A.3	Heating, Ventilation, and Air Conditioning Features	114
A.3.1	Air Recirculation Control (ARC)	114
A.3.2	Air Quality System (AQS)	117
A.3.3	Recirculation Control Run (RUN)	118
B	Cruise Control in DOORS	119
B.1	The Primary DOORS Template	119
B.2	Templates for the Enabling Stages	122

C	User Study Materials	125
C.1	User Study Tutorial	125
C.2	User Study - Control Group	140
C.3	User Study - Pattern Group	152
C.4	User Study - Pattern+Interface Group	164

List of Tables

4.1	The 21 features that we modelled over the course of the case study.	41
4.2	The number of features that use the various pattern constructs.	42
4.3	The average number of states and transitions in each feature.	42
4.4	The number of features that are modelled using a pattern variant.	43
5.1	The six questions that we asked participants about the provided state-machine model of the ACC feature. The criteria we used to evaluate participant's responses are given in italics.	68
5.2	The total number of participants who answered each question correctly. . .	69
5.3	The first three of seven requirements details that we used as evaluated criteria to assess the correctness of a participant's state-machine model. The criteria we used to evaluate participant's responses is given in italics. . . .	72
5.4	The final four of seven requirements details that we used as evaluated criteria to assess the correctness of a participant's state-machine model. The criteria we used to evaluate participant's responses is given in italics.	73
5.5	The total number of participants who modelled each evaluation criterion correctly.	75
5.6	The average confidence of each participant group in his or her solution to the comprehension and modelling tasks.	77

List of Figures

2.1	An example domain model for an automotive product line.	7
2.2	State-machine notation.	8
2.3	Two anonymized state-machine models from production-grade requirements documents.	13
3.1	The Mode-Based Behaviour Pattern, where the modes are coloured with the degree to which the feature’s interactions with its environment are restricted.	18
3.2	The Ordered Enabling variant of the Inactive state extension.	21
3.3	The Unordered Enabling variant of the Inactive state extension.	21
3.4	The Hybrid Enabling variant where some stages are ordered and other stages are not ordered.	23
3.5	The Hybrid Enabling variant where the overall enabling process is unordered but substages are ordered.	23
3.6	The framework of the Active state extension showing the kinds of behaviour that can occur in each state.	25
3.7	The Active state with multiple instances of the Primary Active Region. . .	27
3.8	The Cruise Control (CC) feature.	29
3.9	The Lane Centring Control (LCC) feature.	31
3.10	The primary pattern DOORS template.	34
3.11	A single enabling step in the pattern.	35
3.12	A template for describing the Unordered Enabling process.	36
3.13	A fragment of the CC requirements showing several user action descriptions.	37

3.14	A fragment of the CC requirements showing several of the Active state's requirements.	37
4.1	Adaptive Cruise Control (ACC) feature.	44
4.2	The Heating, Ventilation, and Air Conditioning (HVAC) feature.	46
4.3	The Heating, Ventilation, and Air Conditioning (HVAC) feature.	47
4.4	The Heating, Ventilation, and Air Conditioning (HVAC) feature modelled without air quality control.	50
4.5	The Air Quality region of HVAC modelled as the Air Quality System (AQS) feature.	51
5.1	The previous experience with state-machine modelling of all of the study participants in each group.	61
5.2	The previous experience with automotive modelling of all of the study participants in each group.	61
5.3	A Venn diagram showing the relationships between each study participant's experience with state-machine modelling, automotive modelling, and industrial modelling.	62
5.4	Each group's participant's comfort with state-machine models.	63
5.5	The Adaptive Cruise Control feature's state-machine model provided to the PI group's participants.	64
5.6	The Adaptive Cruise Control feature's state-machine model provided to the P group's participants.	65
5.7	The Adaptive Cruise Control feature's state-machine model provided to the C group's participants.	66
5.8	The average amount of time that the participants of each group spent on the state-machine comprehension task.	79
5.9	The average amount of time that the participants of each group spent on the state-machine modelling task.	79
A.1	The Automatic Braking (AB) feature.	92
A.2	The Anti-lock Braking (ABS) feature.	93

A.3	The Active Trailer Stability Assist (ATSA) feature.	94
A.4	The Brake Assist (BA) feature.	95
A.5	The Brake Cleaning (BC) feature.	96
A.6	The Electric Park Brake (EPB) feature.	97
A.7	The Enhanced Traction System (ETS) feature.	98
A.8	The Hill Hold (HH) feature.	99
A.9	The Traction Control System with Electronic Stability Control (TCS_ESC) feature.	100
A.10	The Competitive Traction Control System with Electronic Stability Control (Competitive_TCS_ESC) feature.	102
A.11	The Manual Park Brake (MPB) feature.	104
A.12	The Forward Collision Alert (FCA) feature.	106
A.13	The Lane Centring Control (LCC) feature.	107
A.14	The Lane Keep Assist (LKA) feature.	108
A.15	The Lane Change Control (LXC) feature.	110
A.16	The Road Change Alert (RCA) feature.	112
A.17	The Air Recirculation Control (ARC) feature.	114
A.18	The Air Quality System (AQS) feature.	116
A.19	The Recirculation Control Run (RUN) feature.	118
B.1	The Rational DOORS requirements for the Cruise Control (CC) feature (Main Template: part 1/3).	120
B.2	The Rational DOORS requirements for the Cruise Control (CC) feature (Main Template: part 2/3).	121
B.3	The Rational DOORS requirements for the Cruise Control (CC) feature (Main Template: part 3/3).	122
B.4	The Rational DOORS requirements for the Cruise Control (CC) feature (Enabling State: Disabled).	122
B.5	The Rational DOORS requirements for the Cruise Control (CC) feature (Enabling State: User Enabled 1).	123

B.6	The Rational DOORS requirements for the Cruise Control (CC) feature (Enabling State: Environment Enabled 1).	123
B.7	The Rational DOORS requirements for the Cruise Control (CC) feature (Enabling State: User Enabled 2).	123

Chapter 1

Introduction

This thesis presents a pattern for expressing the requirements of a feature according to its modes of behaviour, and a generic feature interface that separates a feature’s behaviour into its public and private components. This chapter provides an overview of our research method, lists our contributions, and describes the organization of this thesis.

1.1 Thesis Description

In software engineering, a *pattern* is a proven solution to a common problem. Patterns provide several secondary benefits such as improved documentation and a standard vocabulary that improves communication among developers [47]. Patterns are most commonly used during the design and implementation phases of software development. However, patterns can be used earlier, during the requirements elicitation and specification phases of development.

A *requirements pattern* addresses a problem that arises when eliciting or specifying the requirements of software. Two particular benefits of requirements patterns are *efficiency* in eliciting or documenting the requirements problem, and *predictability* in knowing that the resulting specification should be comparable to previous specifications derived from the same patterns.

This work presents a requirements pattern for specifying the behavioural requirements of individual features in a product line. A *feature* is defined as a “coherent and identifiable bundle of system functionality” [48] that is specified in isolation, can be developed as an independent increment to the system or product line, and may be optional in the final

product. A *product line* is a collection of features, and the dependencies between them, that can be combined to form different software products [12]. Previous work on requirements patterns has either been general [41, 45] or very domain specific [14, 29]. As far as we are aware, this is the first requirements pattern that focuses on feature-oriented development.

The pattern has been designed for use with state-machine models (namely, UML State Machines [35]) and this thesis uses state-machine models to illustrate the pattern concepts. State machines are widely known among the requirements-engineering community and state-machine models are accepted by most practitioners as an effective way to describe the behaviour of a system. However, state machines are difficult to write; determining how to decompose behaviour into states is not obvious, different stakeholders will have different opinions on how to structure the state machine, and the state machines can easily become too complex. Therefore, one of our motivations for creating the pattern was to improve the writability of state-machine models.

The pattern is referred to as a *Mode-Based Behaviour Pattern* because it decomposes the behaviours of a feature according to their operating modes. We have identified three distinct modes: *Inactive*, which describes a feature’s enabling process; *Active*, which describes a feature’s essential requirements; and *Failed*, which describes the failure requirements of a feature.

A key attribute of the pattern is that the states of the pattern can serve as a public interface of a feature. The interface exposes only a feature’s high-level details — the three states based on operational modes — that are common to all features that apply the pattern. The interface is useful because the person specifying the requirements of a *referencing* feature — a feature that references another feature in the system — needs to have knowledge about the state of the other features being referenced. If the information required is only whether a feature is Inactive, Active, or has Failed, then the interface simplifies the specifier’s task because every feature that is modelled using the pattern could reveal this information on its interface. The interface thus provides the most benefit when many of the features in a product line apply the pattern.

Our research group is partnered with an automotive company that is seeking to improve their requirements elicitation and specification processes. As part of this partnership, they have given us access to requirements documents for several of their automotive features. To validate the pattern’s applicability, we have performed a case study in which we modelled the requirements of 21 production-grade features, based on the documented textual requirements. The requirements of each feature was manually examined and used to create a state-machine model of its behaviour employing the pattern. Our results seem to be promising as we found that all of the 21 features could be modelled using the pattern.

The benefits of the pattern were explored by performing a user study involving undergraduate and graduate students at the University of Waterloo. Most of the participants had some level of knowledge of state-machine modelling. The study had 18 participants split into three groups: a Control group, a Pattern group, and a Pattern+Interface group. The participants of each group received a slightly different tutorial and questionnaire: members of the Control group received no information about the pattern or interface, members of the Pattern group received information about the pattern but no information about the interface, and members of the Pattern+Interface group received information about both the pattern and the interface. Those participants who received information about the pattern produced the most correct and most complete models.

Lastly, several *templates* have been created within IBM's Rational DOORS¹ that help specifiers to employ the pattern when specifying a feature. Rational DOORS is a requirements management system that separates requirements into *modules*, where each module describes a set of requirements. A DOORS *template* is a module that pre-defines the kind of information to be provided for the requirements and how that information is presented. Three DOORS templates have been created that provide a pre-defined structure for a feature's requirements. We chose to embed support for our pattern within Rational DOORS because it is the most widely-used requirements management system, and because DOORS is used by our industrial partner to manage their requirements.

Our thesis statement is:

Automotive features have enough behavioural similarities that a single pattern can be created for structuring feature behaviour. The high-level components of the pattern can serve as an interface to features. When applied, the pattern eases the task of specifying new requirements.

1.2 Contributions

This thesis presents four main contributions:

- A general pattern for structuring the behavioural requirements of a feature. The pattern was developed from similarities that we observed among requirements of automotive features, but there is nothing domain specific about the pattern. We believe that the pattern is more widely applicable to embedded systems features

¹<http://www-03.ibm.com/software/products/us/en/ratidoor/>

outside of the automotive domain but that is not shown in this thesis. We explore the forces that drive adoption of the pattern and we list several variants of the pattern.

- A general interface by which a feature reveals to other features its current mode of operation. We show that the vast majority of inter-feature references depend on only information that is revealed by the public interface of a feature.
- A user study that shows that state-machine models that employ the pattern are easier to write than models that are created without using the pattern. The study was performed on 18 participants with varying degrees of experience writing state machines.
- A catalogue of state-machine models that abstractly model the behavioural requirements of 21 production-grade automotive features. This kind of industry data is not widely available, and so despite our models being abstract — we do this to avoid revealing any proprietary details — they still provide a good model for examining the requirements of features used in industry. Complete, detailed versions of the models have been provided to our industrial partner.

1.3 Thesis Organization

Chapter 2 of this thesis provides an overview of the notation and vocabulary used throughout the thesis and discusses related work. Chapter 3 introduces the Mode-Based Behaviour Pattern as well as some variants and examples. Chapter 3 presents the generic feature interface that is a natural consequence when the pattern is used extensively, and it presents the DOORS template for documenting features that are structured according to the pattern. The results of the case study are presented in Chapter 4 and the user-study results are presented in Chapter 5. Our work is summarized in Chapter 6. Appendix A includes abstract models for all of the features that we examined in the case study, and Appendix B provides a complete example of an automotive feature documented using our DOORS template. Appendix C contains the tutorial and study materials created for the user study.

Chapter 2

Background

This chapter provides the background needed to understand the rest of the thesis. First, this chapter gives a brief overview of feature-oriented software development and product lines, and then discusses the UML State Machine language used in the models provided in the thesis. Next, this chapter presents various related work. Lastly, it discusses the existing requirements documents from our industrial partner as some motivation for this work.

2.1 Feature-Oriented Requirements

Software requirements describe what is needed to solve a problem or achieve some objective. Typically, requirements describe a system’s functionality in the context of its environment (i.e., the portion of the real world that is relevant to the software development project). This work is interested only in a system’s behavioural requirements, and assumes that the system has some environment that it reacts to and affects. The behaviour of a system is assumed to be decomposed into *features*, where each feature is defined as a “coherent and identifiable bundle of system functionality” [48] that is specified in isolation, can be developed as an independent increment to the system or product line, and may be optional in the final product. This is referred to as *feature-oriented software development* (FOSD). Some of the benefits of using FOSD are:

- By decomposing behaviour into features, software systems can be assembled using features as building blocks [3].

- Reuse of features is achieved by encouraging multiple different products to be created from a single *product line* [3]. A product line is a collection of features, and the dependencies between them, that can be combined to form different software products [12]. Features in a product line can be combined in different ways to form products that meet the needs of individual users and specific situations in which an application will be used. The development effort when creating new products from the product line is reduced because many of the existing features can be reused.

The requirements of each feature are specified separately, in a separate requirements document. Thus, features are treated as first-class entities starting from their conception. Our requirements modelling language is very similar to the Feature-Oriented Requirements Modelling Language (FORML) in which requirements are specified in terms of their desired effect on the environment of the to-be-developed product [44]. A FORML requirements specification of a product line includes a single feature model, a single domain model, and a state-machine model for each feature in the product line.

The FORML *feature model* describes the relationships between features in the product line. The feature model is a tree where the root is the product line and every other node is a feature. There are two types of relationships that are modelled in the feature model: a requires relationship and an excludes relationship. The *requires* relationship is specified by the tree structure, where a feature requires that its ancestors be present in any product in which it is present. The *excludes* relationship is modelled by connecting two features with a dashed line, indicating that those two features cannot exist in the same product. In some cases, a requires relation can also be modelled using a cross-tree relation in the same way as the excludes relation is modelled. More information on feature models can be found in previous work by Kang [24].

A FORML requirements specification includes a model of a product's environment, called a *domain model*, that defines environmental phenomena that can be sensed or controlled by the product. Figure 2.1 shows a simple domain model for an automotive product line with two features: Adaptive Cruise Control (ACC) and Lane Centring Control (LCC). The behaviour of these features is not important at this time. The environment includes a Vehicle that has several attributes (e.g., speed). The Vehicle is on a RoadSegment, which has one or more lanes, one of which is the current lane of the vehicle. The environment also has a Driver (who may or may not be attentive), and a heads up display for presenting information to the driver. The domain model uses UML Class Diagram syntax to express classes of objects in the environment, object attributes, and relationships between objects. Complex constraints over the domain can be modelled using OCL constraints.

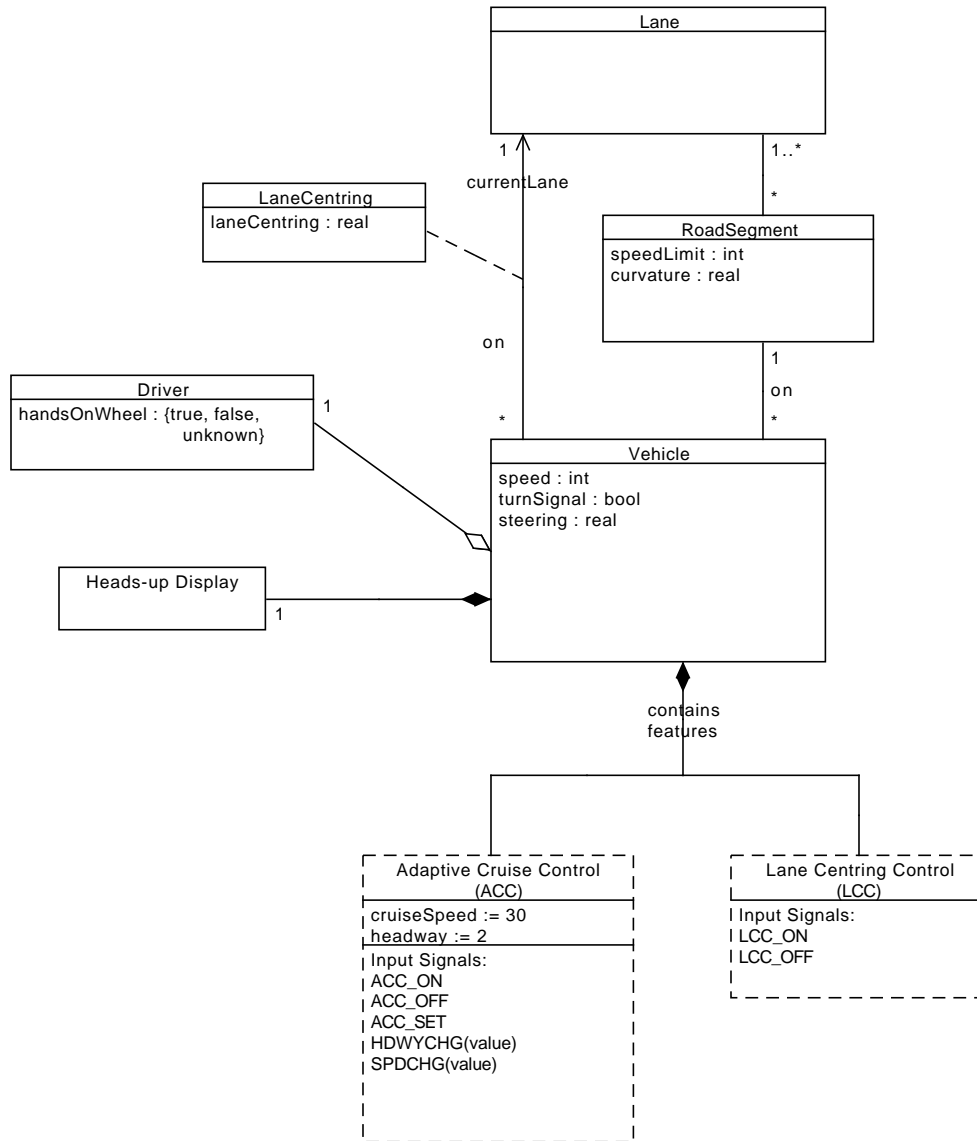


Figure 2.1: An example domain model for an automotive product line.

The domain model also includes class declarations for the product line’s two features: ACC and LCC. The domain model includes feature classes (which correspond to the features in the feature model) because, in feature-oriented requirements where each feature is modelled separately, the environment of a feature includes the other features in the prod-

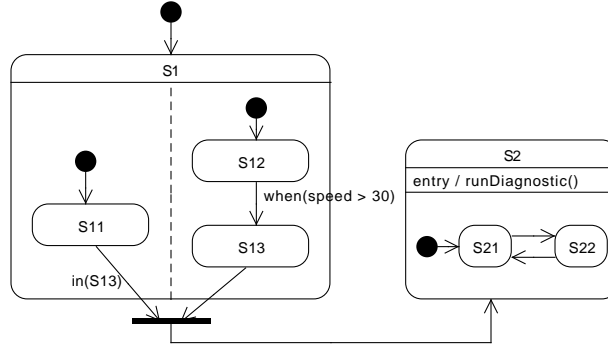


Figure 2.2: State-machine notation.

uct line. Features are denoted as classes with dashed borders. Feature-specific data (e.g., the speed at which the cruise control feature will maintain the vehicle) are modelled as attributes of the feature classes. Feature classes also list their input and output events.

In practice, the domain model for an entire product line could be very large. A requirements specifier who wishes to include a domain model with a feature’s requirements will likely want to present a slice of the domain model that includes only the portions that are relevant to the feature.

Every feature in a product line is modelled as a state-machine model. There is a one-to-one mapping between features and state machines¹. The state-machine modelling language is similar to that of UML State Machines [35] or statecharts [20]. At its core, a state-machine model is a collection of states and transitions between those states. The UML State Machine notation adds several language features to support the modelling of complex behaviours. Consider the state machine exhibited in Figure 2.2. A state may contain sub-states; in this case, the former is called a *superstate* (e.g., S1, S2). A superstate may be decomposed into one or more *concurrent regions* that are separated by dashed lines (e.g., S1); regions model orthogonal behaviour that can occur in parallel. A transition from a black circle to a state designates the initial state of a machine (e.g., S1). If a transition’s destination is a superstate, then the next state is the initial state of the superstate’s sub-machine (e.g., S21) or the initial states of the superstate’s regions (e.g., S11 and S12).

¹In FORML a feature can be modelled as a state-machine fragment, but without loss of generality we assume that each feature is modelled as a full state machine

Transitions can be annotated with an *event*, which is a user action; a *guard condition*, which is a boolean condition over environmental variables; and a set of *actions* on environmental variables; all of these annotations are optional. Events initiated by the user are denoted in upper-case and all other annotations are lower-case. Guard conditions are typically delimited by square brackets to distinguish them from events. Actions are prefaced with a slash. The following are several examples of valid transition labels:

1. when(Vehicle.speed > 30) / Headlights.lightLevel := ON
2. LCC_ON
3. SPDCHG(value) / ACC.cruiseSpeed += value

Note how literals in the labels refer to events (i.e., user actions) and variables (i.e., environmental conditions) from the domain model presented in Figure 2.1. More information about the types of transition labels can be found in a recent paper by Shaker and Atlee on FORML [44].

There are also several types of special transition annotations. Annotation *when(c)* refers to the event of condition *c* becoming true (e.g., *when(Vehicle.speed > 30)*). Annotation [*in(S)*] is a condition that is satisfied when the product’s execution is in state *S* (e.g., *in(S13)*); state *S* might refer to a state in another feature.

A transition is *enabled* when the transition’s triggering event occurs. Assuming that the state machine is deterministic, if the guard condition of the transition is also true then the transition *executes*. When the transition executes, the transition’s actions are performed. If a transition is unlabelled, it is enabled as soon as its source state is entered. The *join* pseudo-state (modelled as a black bar) is used to aggregate multiple transitions (e.g., the transitions from source states S11 and S13 to destination state S2). The join’s outgoing transition executes only when all of its incoming transitions are enabled.

A state may be annotated with *state actions* (e.g., S2). A state action has the same types of labels as a normal transition: a triggering event, guard condition, and actions, all of which are optional. The primary reason to use a state action is that the transition does not cause a change of state therefore, any currently executing actions are not interrupted. Also, no state action triggered by the state entry or state exit is enabled. A secondary benefit is that use of state actions reduces the number of visible transitions in the state machine, thereby reducing the visual complexity of the model.

A feature’s requirements may refer to some complex data computation that would be difficult and time consuming to model at the requirements phase. Data computations can

appear in expressions for events, guard conditions, or actions. Such a computation can be represented abstractly as an *undefined function* that models the computation without explicitly defining it. For example, in Figure 2.2, the *ComputeValue()* function is an action that is executed when state S2 is entered.

A product’s behaviour can therefore be thought of as the composition of the state machines of all of the features in the product².

2.2 Related Work

This section briefly summarizes related work on requirements patterns and on interfaces for features.

Patterns for Easing Requirements Elicitation

Early work on requirements patterns includes domain abstractions or clichés [41] and domain models [4, 45], which record general domain knowledge. The *Requirements Apprentice* [41] employs a library of clichés that can be reused in the specification of multiple systems, where a *cliché* is a set of roles – such as a repository, its contents, and its users – and constraints between roles. Clichés are normally documented using semi-structured text rather than graphical models. Sutcliffe and Maiden [45] extended these ideas to a catalogue of generic reusable *domain models* that encode structural and behavioural requirements of domain entities. More generally, Jackson introduced *problem frames* [22] as a way of classifying problems and sub-problems according to desired changes to or constraints on environment phenomena (e.g., a *transformation* problem, or a *workpieces* problem). A problem frame depicts a context diagram that relates the system-to-be-developed, distinct domains of the environment (à la domain models [45]), and desired requirements among domains (e.g., a transformation requirement relating an input domain and an output domain). Clichés, domain models, and problem frames assist the engineer to elicit requirements; and the use of domain terms improves the consistency of vocabulary in requirements documentation. Our work on feature patterns is complementary in that it aids in the structuring and documentation of behavioural requirements after they have been elicited and decomposed into feature modules.

²Composition of FORML feature modules is defined to be a model of the product line comprising those features [44]. However, in this thesis the composition of a set of features’ state machines is a model of the product comprising these features

The use case unification method (UCUM) [46] specifies a predictable method for transforming use-case descriptions and sequence diagrams into a unified behaviour model of a system using statecharts. Because of the predictability of the UCUM transformation, all those stakeholders who are involved in the transformation process should have a similar mental model of the unified behaviour model of a system. Our pattern provides similar predictability benefits to the UCUM because stakeholders who know the pattern know what to expect when they view a model created using the pattern. However, the UCUM is not specific to features. Because the UCUM has an effect on the structure of the resulting behaviour model it is likely not complementary to our pattern.

The Software Cost Reduction (SCR) requirements model [21] decomposes a system's behaviour into mode classes and modes, and our use of the term *mode* comes from their work. A mode class in an SCR model is comparable to a feature or a sub-system of tightly coupled features in our work. However, SCR does not propose any reusable pattern for decomposing behaviour into modes, does not employ hierarchy for organizing a mode class's modes, and does not have any concept like an interface for mode classes.

Domain-Specific Patterns

Domain experts have collected and codified patterns for modelling specific types of requirements, such as business problems (e.g., accounts, transactions, plans, contracts) [16], embedded-system requirements (e.g., controllers, fault handling, watchdogs) [14, 29], information systems (e.g., information, presentation, access control) [50], security requirements [9], and nonfunctional requirements [17]. Most of these patterns focus on how to structure inter-related components, although Douglass [14] and Konrad et al. [29] include behaviour models of interactions among components. Our concept of a feature could be analogous to the behaviour of a single component in these other approaches, or could be analogous to some system-level functionality that involves multiple components. In either case, our pattern for modelling a feature's behaviour would be complementary to these domain-specific requirements patterns because our pattern advises on how to structure the requirements of an individual feature, whereas these patterns focus on connections and relations among multiple specialized components.

Feature Interfaces

Much has been written about modularity, interfaces, and information hiding [38]. We focus our discussion on feature modularity. Within the feature-oriented software-development

community, the emphasis of feature modularity is on cohesion and locality of feature information [7, 25, 34]. There is no concept of feature interfaces or support for information hiding, so features refer to and directly override the details of other features. At the other extreme, the feature-interaction community often models features as distinct modules that have no knowledge of each other [23, 31]. There is no need for feature interfaces because each feature’s information is completely hidden. The downside of total information hiding is that it is hard to specify and manage intended feature interactions (e.g., enhanced or overriding behaviour). The aspect-oriented community has proposed aspect-aware interfaces that advertise a module’s pointcuts as well as its public data attributes and methods [1, 27], thereby providing limited means by which other modules can use, extend, or override the module’s services. However, such interfaces are not stable, as a module’s set of public pointcuts tends to change as new aspects are introduced or evolve [27]. In contrast, we propose a feature interface that provides limited information about a feature’s current execution state (information that is useful in the modelling of other features), such that the interface is stable and generic for all features. There is also work on deriving feature interfaces to support compositional verification [32], however such interfaces do not aid in the specification or evolution of feature models.

2.3 Existing Requirements Documents

Our industrial partner’s engineers have indicated that their product line includes over 500 features. Each feature tends to be fairly isolated from all other features, and rarely will a feature directly modify the behaviour of another feature. Instead, a feature will request information from other features and use that knowledge to determine its own behaviour.

In our industrial partner’s requirements documents, a feature’s required behaviour is primarily described in natural language accompanied by tables that provide information about environmental variables and a context diagram that lists the feature’s input and output signals. The signal names are, in most cases, not descriptive which makes it difficult for someone unfamiliar with the system to understand what the signals represent. Overall, we found that the context diagrams provided little assistance when trying to understand a feature’s behavioural requirements. The description of a complex feature is sometimes supplemented with a state-machine that sketches the feature’s high-level behaviour — especially if the feature’s enabling process progresses through multiple stages, or if the feature has multiple conditional behaviours once it is active. Such a state machine rarely includes details about a feature’s active behaviour, and its transitions tend to specify only a few of the feature’s enabling conditions and none of the feature’s actions.

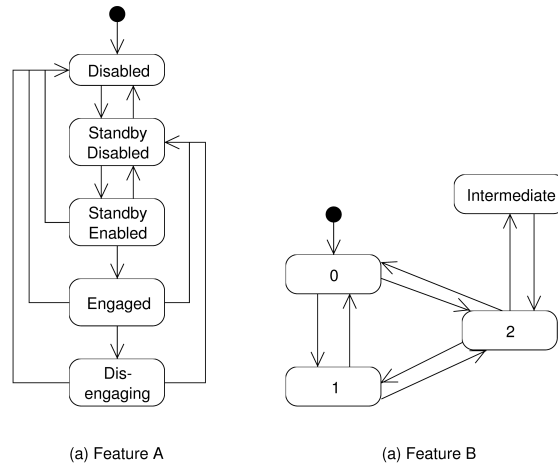


Figure 2.3: Two anonymized state-machine models from production-grade requirements documents.

To illustrate the kinds of state-machine models in our industrial partner’s requirements documents we have included two examples of features’ state machines as they exist in the requirements documents. Figure 2.3 shows anonymized versions of these state-machine models³. Although the state machines have been anonymized, the state names in the figure reflect the state names in the industrial documents. Feature **A**, depicted in Figure 2.3a, initializes in state Disabled and proceeds through its enabling process (states Standby Disabled and Standby Enabled) to the Engaged state, which represents all of the feature’s active behaviour. Details of how the feature behaves when it is Engaged were not included in the industrial partner’s state-machine model of feature **A**. Lastly, the Disengaging state is entered when feature **A** is becoming disabled, but is not yet fully disabled. Feature **B**, depicted in Figure 2.3b, initializes in state 0 (On)⁴, indicating that the feature immediately starts to affect the system’s behaviour. Again, the feature’s behaviour while On was not included in the industrial partner’s state-machine model of feature **B**. The feature transitions from 0 (On) to 1 (Partial On) if the user presses the disable button, at which point part of the feature’s functionality is disabled. The feature transitions from 0 (On) to 2 (Off) if the user presses and holds the disable button, at which point all of the feature’s

³All feature-identifying names and all transition labels have been removed, while retaining the original models’ structure and state names.

⁴In the following description we have introduced the terms On, Off, and Partial On to make the description easier to read – in our industrial partner’s requirements the states are only labelled as 0, 1, and 2.

behaviour is disabled. The feature transitions from 2 (Off) to 1 (Partial On) under certain environmental conditions, and transitions from 2 (Off) to 0 (On) when the user enables the feature. The Intermediate state provides the same behaviour as state 0 (On), and is used as a temporary override that activates the feature in the event of an emergency.

There are several interesting observations that can be made about our industrial partner's state-machine models for features **A** and **B**.

- It is common for documents to use slightly different terminology, or for the same term to be used differently, across multiple documents (e.g., an activated feature is in state 0 vs. On vs. Active vs. Engaged). This kind of inconsistent naming is unlikely to be a serious problem, but can lead to minor ambiguities and confusion. Domain knowledge gained from reading one feature's requirements does not ease the task of reading other features' requirements.
- State-machine models tend to be simple and non-hierarchical. Of the 21 feature requirements that we examined, 9 included state-machine models, and of these, 4 used hierarchy to describe some aspect of the active behaviour for the feature.
- The state machines often omit failures and recovery, although the textual requirements mention how (or at least that) the feature can fail. Of the 21 feature requirements, 11 requirements mention the possibility of the feature failing, 8 requirements specify a feature's failure and recovery conditions, and 4 of the 9 state-machine models include failure states. The feature-requirements documents are sometimes light on failure requirements because many features (and all safety-critical features) have a separate safety-requirements document that describes how the feature behaves in the presence of failures. We did not have access to any safety-requirements documents.
- Despite their many differences, the features have the same basic modes of operation: (1) active, in which the feature affects system behaviour; (2) becoming enabled; and (3) failed. These similarities in the features' behaviours prompted us to propose a pattern for structuring a feature's behaviour model in a way that explicates the similarities. Further study of the features' requirements suggested ways of decomposing and structuring the sub-behaviour of how a feature becomes enabled and how a feature affects the environment of the product.

Chapter 3

The Pattern

This chapter describes the pattern that we have devised and the accompanying interface for features. The pattern decomposes the model of a feature’s behavioural requirements, based on the three modes of operation that we have observed in our industrial partner’s feature requirements: Inactive, Active, and Failed. This chapter also describes several pattern extensions that provide guidance on how to structure the Inactive and Active behaviours of a feature.

The pattern is presented using the pattern template provided by the Third International Workshop on Requirements Patterns [10] for presenting patterns published in the workshop. We chose to use that template over more widely-known pattern templates because it is specifically designed for describing software requirements patterns.

The template has been slightly altered to suit our pattern. The template does not include a section on *Known Uses* because we are only now proposing the pattern and have not yet seen it used in any existing works. The template also does not include a *Cataloging* section because we are presenting only one pattern and so we do not need to place it in context with other patterns. We have added a *Resulting Context* section to describe consequences that result from applying the pattern.

The *Solution* section of the pattern uses state machines to illustrate the pattern concepts. The example that we provide is based the requirements of a specific production-grade automotive feature from our industrial partner. Details are removed or changed to avoid revealing proprietary information.

Following the pattern description, we describe how the high-level states of the pattern can serve as a general interface for referring to a feature’s high-level behaviour. At the end

of this chapter, we describe several Rational DOORS templates that we have created for specifying the requirements of features modelled using the pattern.

3.1 Pattern

Name: Mode-Based Behaviour Pattern

Context: *The Context lists the conditions under which the pattern is valid. The Requirements Engineering Activity lists the phase(s) of the requirements process in which the pattern should be used. The Stakeholders are those, from all of the product's stakeholders, who are involved in the use of the pattern.*

- Requirements Engineering Activity: The pattern is used during the Requirements Elicitation and Specification phases of development.
- Stakeholders: The stakeholders of models created using the pattern are the Requirements Engineers, Software Developers, and Product Managers. The Requirements Engineers specify the feature's requirements, the Software Developers implement the requirements, and the Product Managers manage the feature throughout its lifetime.

Problem: *The Problem describes an undesirable situation faced by the stakeholders who are listed in the Context section.*

Practitioners have difficulty writing state-machine models, for several possible reasons:

- It is difficult to determine how behaviour should be decomposed into states. This is especially true when a feature is first being specified. For example, modellers find it hard to choose the level of granularity of states (i.e., whether to decompose a state into two or more states), or to decide when to use states versus variables to record information about the machine's execution state.
- Different stakeholders are likely to make different decisions about how a state-machine model should be structured, this complicates the task of reading a model if the structure of the model does not match the reader's mental model of feature behaviour. In many cases, the different modelling decisions that stakeholders make will have no effect on the overall quality of the model, but there are cases in which a particular modelling decision may be more appropriate even though it correctly describes the intended behaviour. For example, although the same behaviour can be modelled in several ways it may be best to minimize the number of states used in the solution.

- A state machine that models very complex behaviour can be difficult to manage. If a model is complex, a reviewer or reader of the model may not take the time needed to understand all of the model’s details. On the other hand, if a state machine models complex behaviour too abstractly, the model is of no use when one needs a detailed understanding of the feature’s behaviour.

Although we do not have concrete evidence for these claims, we have observed these problems when working with student modellers, and these are problems that the engineers at our industrial partner mentioned to us during our discussions with them.

We believe that the Mode-Based Behaviour Pattern has the potential to mitigate many of these problems because it gives practitioners a pre-defined framework for reading and writing requirements.

Forces: *The Forces section describes the concerns that need to be balanced by the requirements process or product.*

- There is a trade-off between the usability of the modelling language and the percentage of a feature’s requirements that are modelled. A state-machine model might not be detailed if it is easier and faster to describe details using natural language.
- The pattern introduces several states. If a feature’s model does not need all of the pattern’s states and the modeller includes the unused states (e.g., for completeness), the unused states will clutter the model and may make the model more difficult to understand. In contrast, if the unused states are removed from the model then a reviewer of the requirements document may be unsure if they were omitted on purpose or as an oversight.

Solution: *The Solution presents the requirements pattern that solves the Problem presented earlier.*

The Mode-Based Behaviour Pattern decomposes and structures the feature’s state machine according to three high-level behaviour modes: Inactive, Active, and Failed. Figure 3.1 shows the high-level structure of the pattern. Each of the high-level states contains one or more sub-machines that model the detailed behaviour for the state.

The *Inactive* state describes a feature’s behaviour when it is not affecting the environment. This comprises a sub-machine that models the behaviour of the feature as it becomes enabled. As will be seen, we propose a sub-pattern for structuring the feature’s enabling

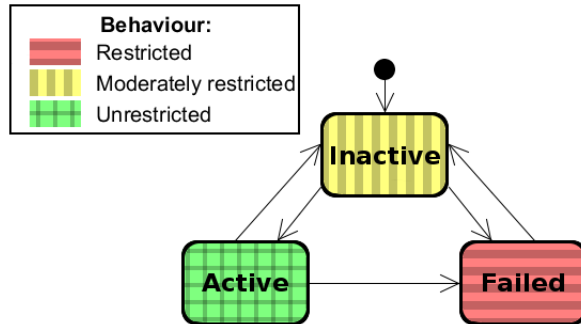


Figure 3.1: The Mode-Based Behaviour Pattern, where the modes are coloured with the degree to which the feature’s interactions with its environment are restricted.

process. Normally, a feature initializes in Inactive. When the feature is completely enabled, it transitions to the Active state, in which the feature performs its essential behaviour.

The *Active* state describes a feature’s behaviour when it is affecting the environment. This comprises a sub-state machine that models all of the ways in which a feature actively affects the behaviour of the system. The exact conditions that cause a feature to deactivate (i.e., that trigger the transitions from Active to Inactive) are feature specific, and thus are not part of the pattern. Transitions to Inactive can emanate from any sub-state within Active. Their respective destination states are normally the boundary of the Inactive superstate, meaning that the feature’s enabling process restarts from the initial state(s) of Inactive’s sub-machine(s).

The *Failed* state captures all aspects of how a feature behaves when it has failed. The exact conditions under which a feature transitions to or recovers from the Failed state are feature specific and are not part of the pattern. On recovery, a feature normally transitions to the boundary of the Inactive superstate. The pattern does not decompose the internal structure of the Failed state because the features we have examined typically do nothing more than monitor their environment while failed. Furthermore, the industrial feature-requirements documents to which we have had access are light on failure requirements because many features — and all safety-critical features — each have a separate safety-requirements document that describes how the feature behaves in the presence of a failure. We did not have access to any safety-requirements documents. As will be seen in future subsections, the Failed state is used only when a feature has completely failed and can no longer operate. If a feature provides some degraded service due to a failure, (i.e., the feature provides reduced functionality), then that behaviour is modelled as part of the

Active state. In subsection 3.1.2, we discuss how exactly to model degraded service within the Active state.

Consider the different ways in which a feature can interact with its environment:

- the feature monitors the environment (e.g., an Adaptive Headlights (AH) feature monitors the ambient light level)
- the feature acts on the environment (e.g., the AH feature automatically turns on the vehicle’s headlights)
- the environment monitors the feature (e.g., other features in the system monitoring the AH feature’s current execution state)
- the environment acts on the feature (e.g., the vehicle operator changes the sensitivity of the AH feature)

Each state reflects different types of interaction between the feature and its environment (as depicted in Figure 3.1 using colour). In restricted (red) states, the only allowable interaction is that the feature can monitor its environment, to determine if any of the state’s outgoing transitions are enabled. For example, a feature that has Failed can monitor its environment for signs that recovery conditions have been met. In moderately restricted (yellow) states, the feature can monitor its environment and the environment can act on the feature. For example, it may be possible for a user to manipulate feature settings when the feature is still Inactive (e.g., a user can set the cruising speed before the cruise-control feature becomes Active). In unrestricted (green) states, all four types of interactions are allowed. In this manner, the pattern’s high-level states partition the features’ behaviours into separate modes of operation.

3.1.1 The Inactive extension

In Section 2.3, we noted that most automotive features go through some enabling sequence before they can activate. The Inactive state extension provides advice on how to model a feature’s enabling process. A feature’s enabling process can range from simple (e.g., the user presses a button) to very complex (e.g., the user must perform multiple actions and several environmental properties must hold). There are three variants of the Inactive extension that provide guidance on how to model a feature’s enabling process based upon the degree to which the process is ordered.

1. There is a sequential ordering on the enabling conditions.
2. There are no ordering constraints on the enabling conditions.
3. The enabling conditions are partially ordered.

The enabling process differentiates between two types of enabling conditions: user actions and environmental conditions. A *user action* is an action that is performed directly by the user or human operator (e.g., the user turning on the feature). An *environmental condition* is a predicate over properties of the operating environment of the system (e.g., an automobile's speed) or properties of the execution state of the system (e.g., the state of another feature). We argue that the description of a feature's enabling process should prominently distinguish between user actions and environmental conditions because of the types of constraints each places on the feature design: each user action must have a corresponding user interface; whereas monitored properties must have corresponding sensors, and controlled properties must have corresponding actuators. Also, environmental conditions can often be aggregated on a single transition (i.e., a compound expression that combines conditions through conjunction and disjunction) in order to simplify the enabling process. In contrast, user actions are typically performed one at a time by a user and are therefore not amenable to aggregation on transitions.

Ordered Enabling

The Ordered Enabling variant applies when a feature becomes enabled in stages. The Inactive sub-machine is a sequence of user actions and environmental conditions that must be satisfied in the specified order. Each transition is triggered by either a combination (i.e., conjunctions, disjunctions, negations) of user actions or a combination of environmental conditions, but not both. If a transition is labelled with a user action, then the destination state of the transition is named User Enabled. Otherwise, the transition is labelled with an environmental condition and the destination state is named Environment Enabled. The Ordered Enabling variant can be generalized to any number of enabling stages. This handles cases where the enabling process is very long (i.e., when there are multiple user enabled or environment enabled stages) or very short (i.e., only one stage). When the final state in the sequence is reached, the feature transitions to the Active state.

The state-machine fragment in Figure 3.2 shows how the Ordered Enabling variant is modelled to specify an ordered enabling process. The name of each state is the type of the most recent combination of enabling conditions (user action or environmental conditions)

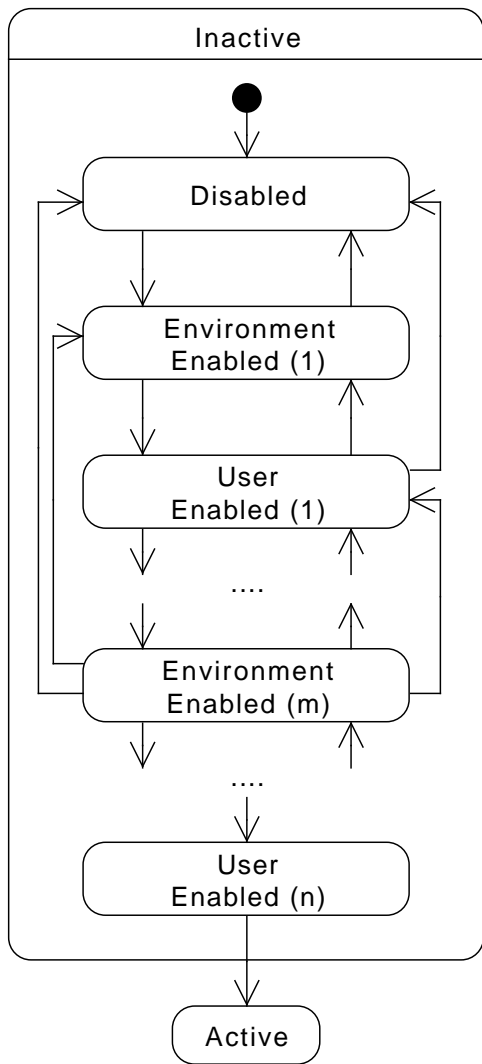


Figure 3.2: The Ordered Enabling variant of the Inactive state extension.

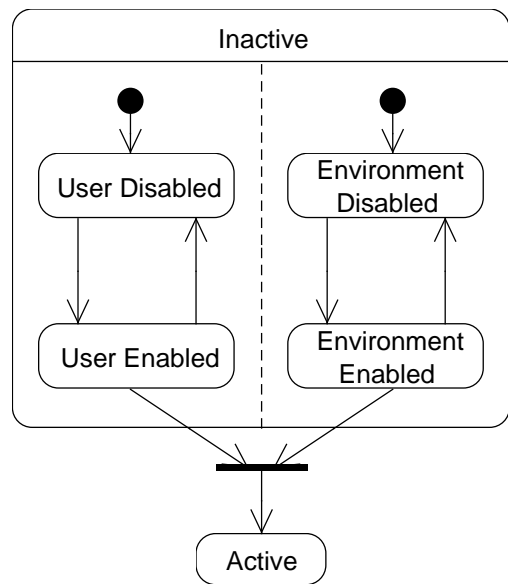


Figure 3.3: The Unordered Enabling variant of the Inactive state extension.

and the sequence number for that type of condition. The enabling sequence may include back transitions from later states in the sequence to earlier states, if enabling conditions became unsatisfied and cause the feature to revert to a less-enabled state. If the enabling process only has one User Enabled state and one Environment Enabled state then the states do not require a sequence number, the numbers are only needed for distinguishing multiple User Enabled stages or Environment Enabled stages.

Unordered Enabling

It often does not matter in what order a feature's enabling conditions become true: as soon as they all hold, the feature activates. The Unordered Enabling variant (shown in Figure 3.3) applies in these situations. The concurrent regions separate user actions (on the left) from the environmental conditions (on the right). A transition is triggered by either a combination of user actions or a combination of environmental conditions, but not both. A region's states are named according to whether the region is checking user actions or environmental conditions. When all of the regions are simultaneously in their most-enabled state, the feature transitions to the Active state. This behaviour is modelled using a join pseudo-state whose source states are User Enabled and Environment Enabled and whose destination state is Active.

The Unordered Enabling variant can be generalized to handle enabling processes comprising multiple sequences of user actions or environmental conditions, but where the order among the sequences does not matter. For example, consider a feature whose enabling process consists of two user actions but there is no specific order in which the actions must occur. Such an enabling process could be modelled using the Unordered Enabling variant with two concurrent regions, one for each user action.

Hybrid Enabling

In some cases, a feature's enabling process is partially ordered. That is, some stages of the enabling process cannot be satisfied until earlier stages have been completed; and in other stages, the order in which user actions and environmental conditions are satisfied does not matter. In the example in Figure 3.4, the enabling process is primarily ordered, but at one point in the sequence, there are enabling conditions whose orderings are not important. The transition from Environment Disabled to Environment Enabled can not take place until after the first user action has occurred. As well, the transition from User Enabled (2) to User Enabled (3) can not take place until the environmental conditions

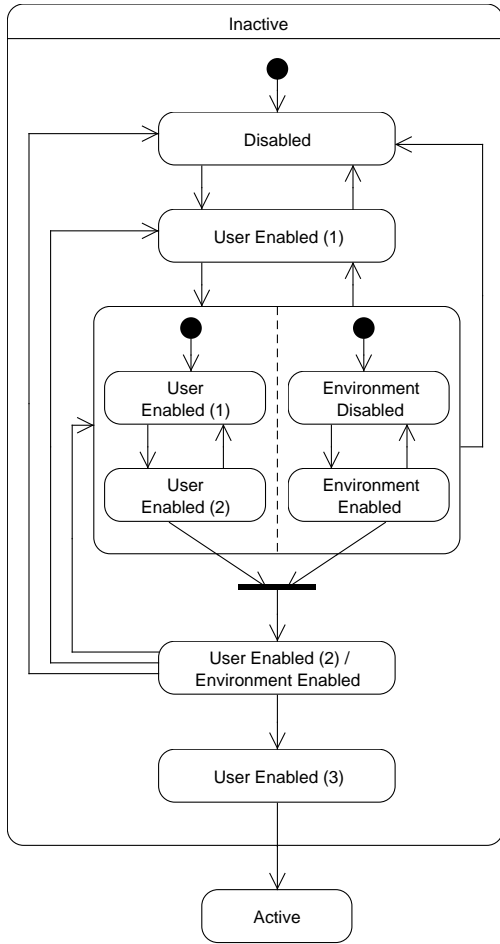


Figure 3.4: The Hybrid Enabling variant where some stages are ordered and other stages are not ordered.

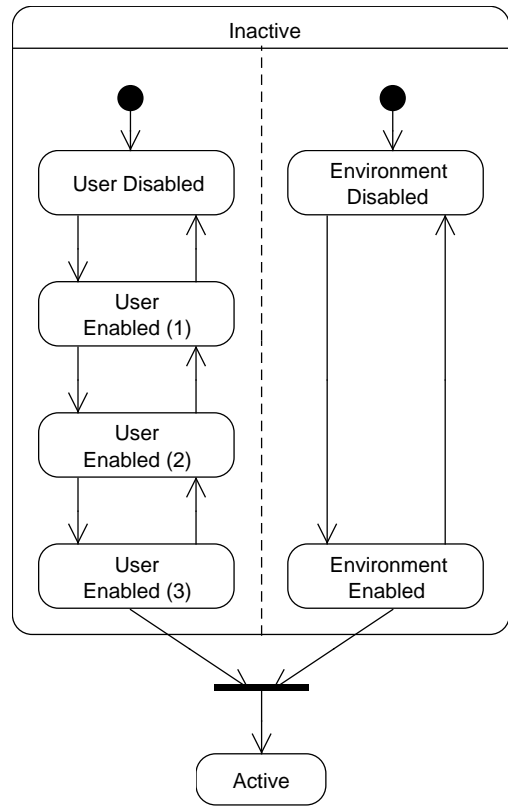


Figure 3.5: The Hybrid Enabling variant where the overall enabling process is unordered but substages are ordered.

are all valid. In a separate example shown in Figure 3.5, the user actions may be staged or the environmental conditions may be staged, but the order in which these substages complete is not important. The Hybrid Enabling variant of the Inactive state extension uses a mixture of concurrent regions and state sequencing to model these kinds of partially ordered behaviours.

The Hybrid Enabling variants is applicable to any enabling process that is not strictly ordered or unordered. That said, we have observed only one feature that utilizes this enabling variant.

3.1.2 The Active state extension

The Active state extension provides advice on how to structure the active behaviour of a feature. There are several kinds of behaviour that a feature performs while active. For example, consider a Cruise Control (CC) feature that maintains the vehicle's speed at a driver-specified value. When the CC feature is active, it alters the vehicle's speed only when the sensed speed is faster or slower than the driver-specified value. When the vehicle's speed matches the driver-specified value, CC maintains the vehicle's current throttle position and only monitors for changes to the vehicle's speed. Sometimes when a feature is active, it actively affects its environment; other times, the feature merely monitors its environment. In some cases, a feature will perform *clean-up* tasks before deactivating or transitioning to the Failed state. Figure 3.6 shows the Active state extension and how it relates to the Inactive and Failed states.

The Active state extension is composed of the *Primary Active Region* (which contains the *Controlling*, *Monitoring*, *Deactivating*, and *Failing* states), and any number of additional concurrent monitoring regions:

1. *Controlling*: This is the state in which a feature actively affects its environment. A feature's Controlling state typically contains one or more sub-machines that model the controlling behaviour of the feature. The Controlling state has unrestricted behaviour.
2. *Monitoring*: When in the monitoring state, the feature only monitors its environment; it does not actively affect its environment. In addition, the feature's environment can act on it (e.g., the user can modify feature settings).
3. *Deactivating*: In this state, a feature is in the process of deactivating, but needs to perform some actions before it becomes Inactive (e.g., the feature automates some

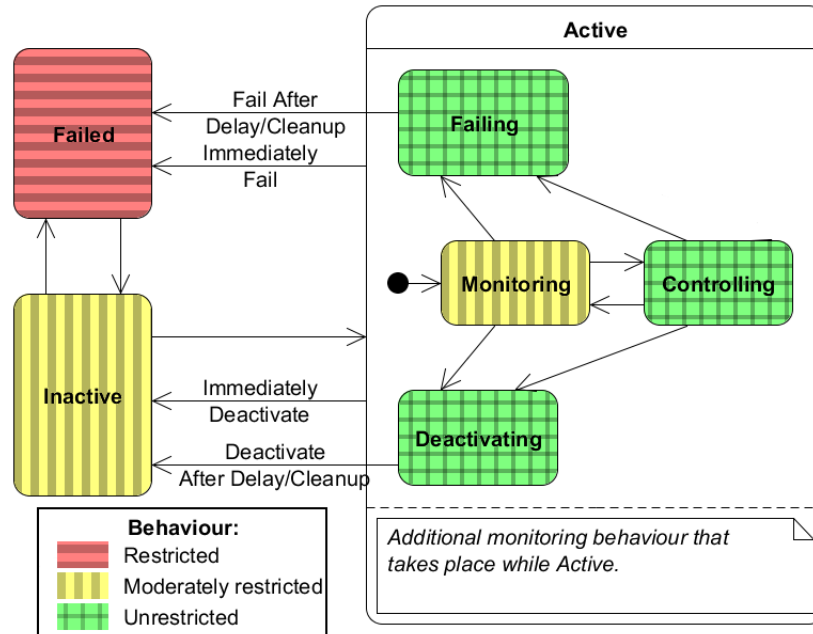


Figure 3.6: The framework of the Active state extension showing the kinds of behaviour that can occur in each state.

user’s task and notifies the user when it is deactivating — remaining operational for a period of time to allow the user to resume responsibility for that task). The Deactivating state has unrestricted behaviour.

4. *Failing*: The purpose of this state is similar to the purpose of the Deactivating state — when a feature is failing, it warns the user and attempts to remain operational temporarily to allow the user to resume responsibility over the feature’s task. The Failing state is also used in cases where a feature performs some degraded service after a failure (i.e., the degraded behaviour is modelled as part of the Failing state). The Failing state has unrestricted behaviour.
5. *Concurrent monitoring regions*: In addition to the Primary Active Region, there may be additional concurrent regions within Active. The additional regions are useful when the feature continuously monitors some environmental phenomena that is used in many of the Active sub-states. This continuous monitoring is different from the environmental monitoring that is performed only in the Monitoring state, for the purpose of determining whether the feature should begin to control its environment.

For example, a braking feature may continually monitor the brake hydraulic fluid levels to determine if it should deactivate (regardless of the current execution state). In contrast, the Monitoring state cares only whether or not the brake pedal has been pressed.

An activating feature normally initializes in the Monitoring sub-state and transitions between the Monitoring and Controlling sub-states. Whether the feature is in one sub-state or the other depends on whether the feature is currently performing some actions that control its environment or not. An Active feature may transition to Deactivating as an intermediate sub-state towards transitioning to Inactive, or to Failing as an intermediate sub-state towards transitioning to Failed. Alternatively, an Active feature may transition from the Monitoring or Controlling sub-states directly to Inactive or Failed (either because no intermediate state is necessary, or because no intermediate state is possible). Such transitions should originate from the border of the Active superstate and terminate at the Inactive or Failed states. The events, conditions, and actions with which the transitions in the Active state are labelled as feature specific and thus are not part of the pattern.

Variants:

There are several alternative implementations of the pattern:

1. The initial state of the feature may be Active. This is useful in cases where a feature has no Inactive behaviour. It is also useful when a feature is Active by default, but can be disabled after initialization, either by the user or by some environmental condition.
2. A feature may deactivate but still be partially enabled. Such cases are most common when the enabling process is Ordered. The Inactive sub-state after deactivation depends on the user actions or environmental conditions that cause the deactivation. In such a case, a transition from Active to Inactive crosses the Inactive superstate's border and terminates at some intermediate stage of the enabling process.
3. An enabling process can have transitions that skip one or more enabling stages. For example, a feature may have some kind of emergency override that automatically activates the feature. This is modelled as a transition that originates from the earlier enabling stage and terminates at the final enabling stage (which is immediately followed by the transition to Active). Likewise, an enabling process with many stages can have transitions from later stages to earlier stages.

4. The initial sub-state of Active can be Monitoring or Controlling. Sometimes, a feature needs only a Controlling sub-state, in which case, it should be the initial sub-state of Active.
5. Features in the Failing state can transition back to Controlling if the failure is fixed before the transition to Failed takes place. This addresses the case of partial or transient failures that recover quickly. We did not observe this variant in the industrial requirements documents that we examined, but it was a case raised by the engineers working at our industrial partner.
6. The Deactivating or Failing sub-states can be omitted if they are not useful. This reduces clutter in a model that does not fully utilize all of the pattern's constructs. However, there is a trade-off between clutter and comprehension because omitting pattern states from a model introduces ambiguity: the requirements reviewer may be unsure as to whether the states are missing because they have not yet been included (i.e., the model is incomplete) or because they are not being used.
7. There can be multiple instances of the Primary Active Region in the Active state. Figure 3.7 shows an example of an Active state with two instances of the Primary Active Region. This variant is used when a feature performs one more orthogonal behaviours while Active. However, the use of this variant may be a sign that the feature should be split into separate features, such that each feature's Active state contains only one instance of the Primary Active Region.

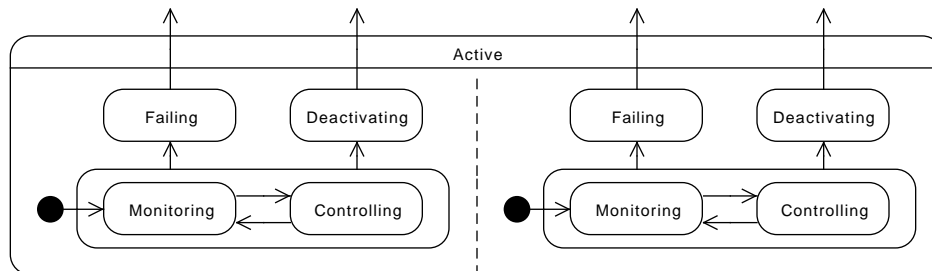


Figure 3.7: The Active state with multiple instances of the Primary Active Region.

Resulting Context: *The Resulting Context describes the consequences of using the pattern to model a feature's state machine.*

- The pattern simplifies the requirements engineer’s task when describing feature requirements and modelling state machines. We have not proven this, but we expect it to be true because of the way that the pattern pre-determines the high-level states of the feature. Furthermore, the elicitation and specification tasks can be decomposed by pattern mode, thus reducing them to smaller subtasks that can be tackled one by one.
- The more prevalent the pattern is used in feature requirements, the easier it is to read and review multiple requirements documents. This is for several reasons: (1) the pattern introduces a standardized vocabulary for all features that apply the pattern, and (2) the pattern structure organizes the behavioural requirements of a feature by mode, thus making it easier to locate information in an unfamiliar document.
- Any of the pattern’s states that are not applicable to a particular feature can either be (1) omitted from that feature’s model, or (2) included, but not used (e.g., have no entering or exiting transitions or have unlabelled exiting transitions, such that the feature spends no time in the state). The former may result in an ambiguous model: a reviewer may not know whether the omitted states have been left out intentionally or the model is incomplete. The latter may result in a cluttered model.

One possible resolution is to omit unused states from the feature’s model and use special syntax to indicate if the model is complete or incomplete. Previous work by Salay et al. [43] propose a special annotation in the top right corner of a model to indicate if the model is complete (**COMP**) or incomplete (**INC**).

Example:

We provide two examples of features that are modelled using the pattern. The first is a Cruise Control (CC) feature, and the second is a Lane Centring Control (LCC) feature. The CC example is not based on any production or academic feature description. Our model of the LCC feature is based on a requirements document that our industrial partner gave us. In these models, user actions are expressed in upper-case text and all other conditions are expressed in lower-case. To avoid revealing proprietary information, we have abstracted away several details of the features: specifically, in several transitions, the label has been omitted and replaced with the number of conditions that are checked on the transition. We do not claim that our models are either complete or correct, but are merely presented as pedagogical examples.

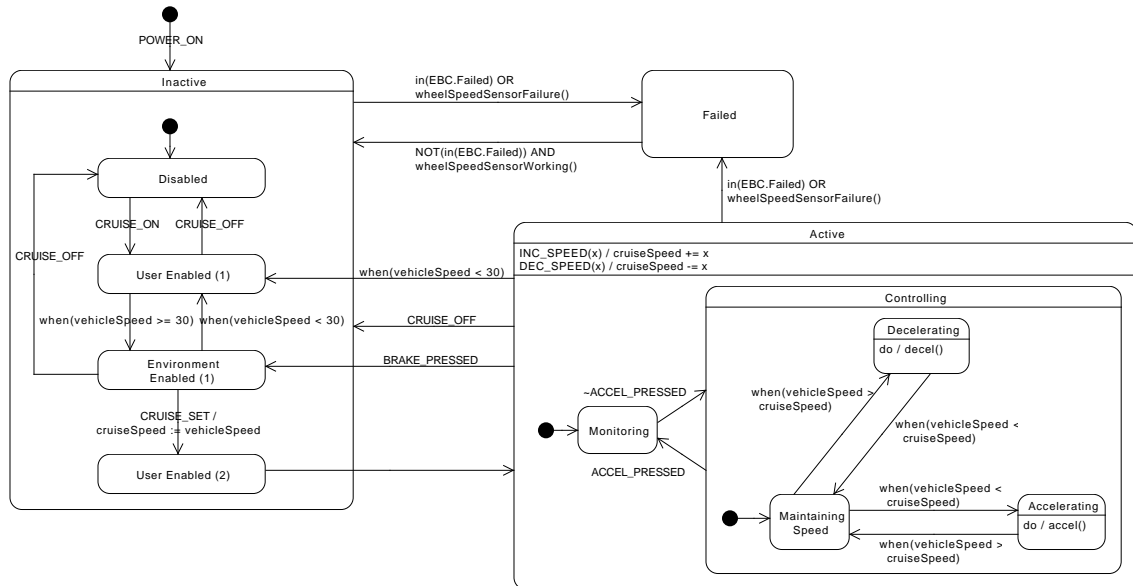


Figure 3.8: The Cruise Control (CC) feature.

Cruise Control

Consider a simplified version of the Cruise Control (CC) feature that, once activated, will maintain the speed of the vehicle at a driver-specified value. Figure 3.8 presents a behaviour model of the CC feature that uses the pattern.

The CC feature is modelled using the Ordered Enabling variant of the Inactive state extension. When the vehicle is powered on, the feature enters the Inactive state. The feature waits in sub-state Disabled until the user presses the CRUISE_ON button. The feature then waits in sub-state User Enabled (1) until the *vehicleSpeed* is greater than or equal to 30 km/h. The feature then waits in sub-state Environment Enabled (1) until the user presses the CRUISE_SET button, setting the *cruiseSpeed* of the feature (i.e., the speed that CC will maintain) to the current *vehicleSpeed*. At this point, the feature transitions to User Enabled (2) and immediately activates.

The Active sub-machine initializes in the Monitoring sub-state. If the driver is not pressing the accelerator pedal, the machine transitions to the Controlling sub-state. The Controlling sub-state initializes in the Maintaining Speed sub-state. If the vehicle speed exceeds or becomes less than the *cruiseSpeed*, the feature will transition to the Decelerating or Accelerating sub-states, respectively. If the driver presses the accelerator pedal

when the machine is in any of the Controlling sub-states, the machine will transition to the Monitoring sub-state. If the driver presses the brake or turns off the CC, or if the *vehicle-Speed* drops below 30 km/h, then the feature deactivates. Depending on the deactivating condition, the feature may transition back to a partially enabled state within the Inactive state.

There are two reasons why CC may fail: (1) if the Electronic Brake Control (EBC) feature fails, and (2) if the sensors detecting the vehicle's speed fail. The machine transitions to Inactive on recovery from the failure.

Lane Centring Control

Consider a simplified version of a Lane Centring Control (LCC) feature which, once activated, will maintain the vehicle's position in the centre of its current lane without any driver input. Figure 3.9 presents a model of LCC that employs the pattern.

The LCC feature is modelled using the Unordered Enabling variant of the Inactive state extension and all four states of the Primary Active Region. The enabling process checks that the user presses the LCC_ACTIVATE button for LCC and confirms that two environmental conditions are true. Once activated, LCC initializes in the Monitoring sub-state of Active and monitors eight conditions to determine if the feature should transition to the Controlling sub-state and start controlling the vehicle's steering. While inside the Controlling sub-state, LCC maintains the vehicle's position in the centre of the lane. There is an additional concurrent region in the Active state that keeps track of whether the vehicle is currently centred in the lane. The Controlling sub-machine checks whether the state machine is currently *in(Centred)* or *in(Not Centred)* to determine whether the feature should correct the vehicle's position.

Determining the conditions to check during the enabling process and the conditions to check in Monitoring is dependent on the feature's requirements. In LCC, the two environmental conditions that are checked during the enabling process are necessary in order to activate LCC, and while LCC is Active if either of these conditions becomes false then LCC deactivates. The eight conditions that are checked on the transitions between Monitoring and Controlling determine whether the centring behaviour should be temporarily overridden. For example, one of the overriding conditions checks if the vehicle driver is currently steering the vehicle; if they are, the LCC sub-machine transitions to Monitoring and does not attempt to control the vehicle until the driver is no longer steering.

When the LCC feature deactivates or starts to fail it goes through the Deactivating or Failing states, respectively. While in the Deactivating and Failing states, LCC slowly

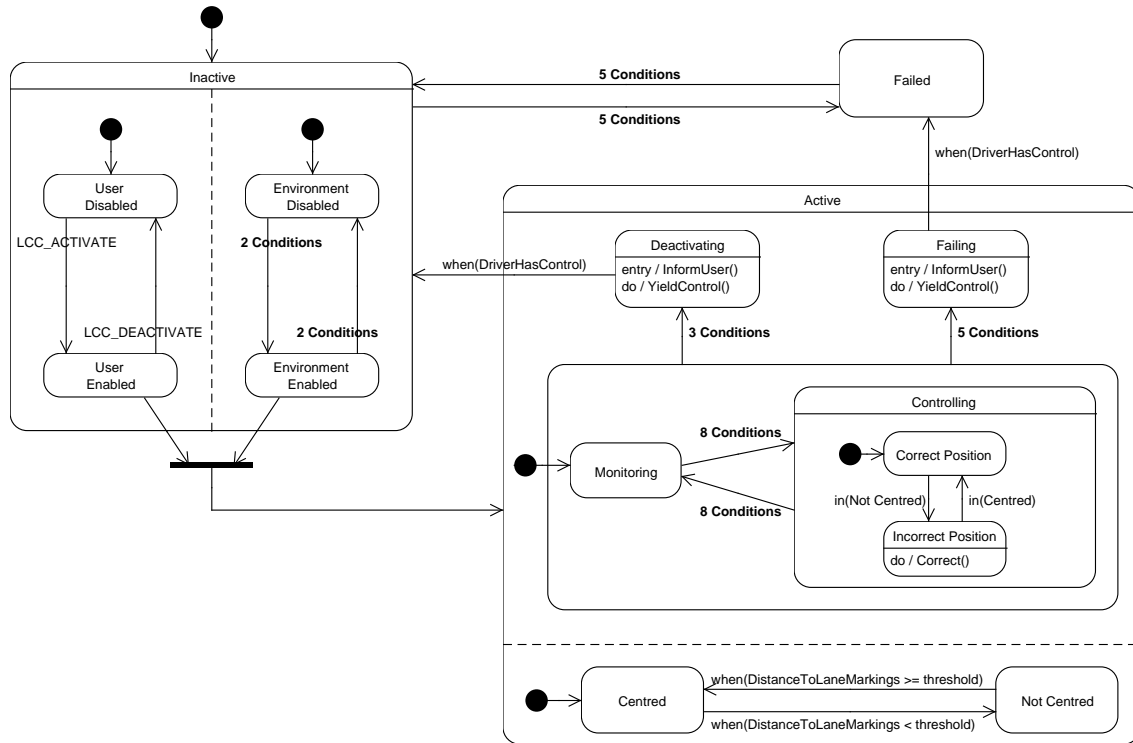


Figure 3.9: The Lane Centring Control (LCC) feature.

yields control of the vehicle to the driver (handled by the `YieldControl()` function) and transitions to Inactive or Failed only when the driver has control of the vehicle.

3.2 Interface

The primary purpose of the proposed pattern is to ease the elicitation, documentation, and review of behavioural requirements of individual features. However, an important side effect is that the pattern can be used to define a *generic behavioural interface* to any feature that applies the pattern.

There has been little research on interfaces for features. In feature-oriented software development, work on feature modularity has focused on features as a criterion for system decomposition and assembly, such as in product-line development [24]; and on the cohesion

of features [7, 25], including language or modelling support for coalescing all information related to a feature into a single module [34]. There is no information hiding among features, and one feature can directly refer to or override the details of other features. Alternatively, in the feature-interaction literature, feature modules are black boxes that have inputs and outputs, but otherwise share no information with each other [23, 31]. Such extreme information hiding facilitates parallel and third-party development of features, but makes it very difficult to specify intended interactions, such as when a new feature extends or overrides the behaviour of an existing feature, or when a feature ought to behave differently in the presence of other features.

We propose a compromise, in which features share a limited amount of information with each other by means of a feature interface. Our ideas are based on our initial analyses of the requirements documents that our industrial partner provided to us: in most instances where one feature’s requirements refer to another feature, the reference is an inquiry as to whether the other feature is active, has failed, or is even present in the system (since many features in a software product line are optional). Thus, we put forward our pattern’s high-level modes as a *generic behavioural interface* for features, whereby a feature reveals whether its current execution state lies within Inactive, Active, or Failed. The interface exports information that can be viewed by observing features; it does not provide hooks that modifying features can use to directly affect the feature’s behaviour. We hypothesize that such an interface reveals useful information about features, and would be sufficient in most cases where one feature needs to know the current state of another feature. Moreover, because the modes are common to all features applying the pattern, the interface does not reveal any details of a feature’s specification that is not already known to specifiers.

For example, consider again the behaviour model of our simple Cruise Control (CC) feature in Figure 3.8. CC fails if the Electronic Brake Control (EBC) feature fails, and we model this failure condition by labelling transitions to and from CC’s Failed mode with *in(EBC.Failed)* guard conditions that monitor whether EBC is in its Failed mode.

Not all feature cross-references are references to interfaces. If two features are tightly coupled, they may refer to one another’s internal details. For example, one feature’s behaviour may depend on a related feature’s current detailed state (e.g., there are limited autonomous driving features that depend on whether the driver is attentive). Alternatively, a new feature might override some detailed behaviour of an existing feature. We would expect such references to another feature’s detailed behaviour to be relatively rare, and limited to tightly coupled features that are part of the same sub-system, developed by the same team, and specified in the same requirements document. We would deem our generic feature interface to be useful if references to information in features’ interfaces were the norm.

3.3 Rational DOORS Templates

Our industrial partner uses the DOORS requirements management system to record and manage the requirements for their vehicle software. To ease the process of creating feature requirements that use the pattern, we have created three DOORS templates that provide a base structure for describing features using the pattern into which the feature-specific requirements can be inserted. One limitation of DOORS is that requirements need to be described textually, so the template is a textual representation of the structure of a state-machine model that follows the pattern. This does not appear to be a limitation, and only changes the medium in which the pattern is communicated.

Our templates are *modules* within a DOORS system and are stored in a central area where every requirements engineer has access. They can then be copied when needed. DOORS includes the Rational Doors Extension Language (DXL), which is used to write scripts for operating on requirements modules. However, DXL does not appear to be an ideal choice for implementing templates (in fact, when we asked for suggestions on DXL development forums, we were told that copying and pasting a base template was the best solution). Hence, although our solution seems somewhat primitive, it appears to be the best solution¹.

Figure 3.10 shows the primary template that is used as the basis for a feature's requirements. The template is divided into the following sections:

1. The feature's *Description* provides a brief high-level textual overview of the feature. The description does not correspond to anything in the pattern but we have included it in the template to serve as an introduction to the feature. We have also observed that the feature requirements documents provided by our industrial partner have a short introduction section.
2. The *List of Environmental Conditions and User Actions* section lists all of the user actions and environmental conditions that the feature uses. We noticed that our industrial partner's requirements always listed at the beginning of a feature's requirements document the signals used the feature (to serve as a reference for reviewers), so we have imitated that here by including in our template all of the user actions and environmental conditions.
3. Section 1.3 of the DOORS template includes a graphical state-machine model of the feature's behaviour. This should be provided before the start of the feature's

¹It is also the method that our industrial partner uses to implement their existing templates.

1 Feature Requirements	
1.1 Description	1
This section should provide a high-level textual description of the feature.	
1.2 List of Environmental Conditions and User Actions	
List all of the user actions and environmental conditions that the feature uses in this section. This may require information about signals and interfaces.	
1.2.1 User Actions	
User Action 1	
User Action 2	
etc...	
1.2.2 Environmental Conditions	2
Environmental Condition 1	
Environmental Condition 2	
etc...	
1.3 State-Machine Model of Feature	3
Place a diagram of the state-machine model here. It may be rather large, so you will probably want to describe the contents of the transitions on other pages.	
1.4 Inactive	
The inactive behaviour requirements.	
1.4.1 Enabling Type	4
Ordered OR Unordered OR Hybrid	
1.4.2	
<i>Link to several enabling steps here. These are found as separate modules in the same folder.</i>	
1.5 Active	
The active behaviour requirements.	
1.5.1 General Requirements	
Requirements that are supposed to be met when within any sub-state of the Active state.	
1.5.2 Controlling	5
Controlling requirements of the feature. This will probably contain many links to functions at lower levels of abstraction.	
1.5.3 Monitoring	
Monitoring requirements of the feature.	
1.5.4 Deactivating	
Deactivating requirements of the feature.	
1.5.5 Failing	
Failing requirements of the feature.	
1.6 Failed	6
The failed behaviour requirements.	

Figure 3.10: The primary pattern DOORS template.

requirements so that the model can be used as a guide when reading the feature’s detailed requirements. The transition labels may be too large to include in the model so a macro can be created to abbreviate long transition labels and the macro expansions can be listed at the bottom of the model.

4. The Inactive section describes the inactive behaviour of the feature. The Enabling Type of the feature is either Ordered, Unordered, or Hybrid. In section 1.4.2 of the DOORS template, the requirements engineer creates a textual representation of the enabling process using the two templates in Figures 3.11 and 3.12.

1 Step <X>: <Environment OR User Enabled>: <FINAL?>
1.1 State Requirements
These are any requirements for behaviour that takes place while the feature is in this state in the enabling process.
1.2 Proceed to NEXT STATE
The conditions that are checked, or user actions that are taken, to proceed to the next step.
1.3 Return to immediately PREVIOUS STATE
The conditions that are checked, or user actions that are taken, to proceed to the immediately previous step.
1.4 Skip forwards or backwards to STATE <Y>
The conditions that are checked, or user actions that are taken, to proceed to state <Y> within the enabling process. This is useful in cases where a backwards transition skips multiple previous stages of the enabling process, or where a forward transition skips several stages of the enabling process.

Figure 3.11: A single enabling step in the pattern.

Each enabling step, as shown in Figure 3.11, contains several pieces of information. An enabling step has an identifier, is an environment-enabled or user-enabled step, and may be the final state in the enabling sequence. The State Requirements section includes any requirements that are specific to that enabling stage (e.g., the user may be able to change some settings of the feature). Sections 1.2, 1.3, and 1.4 of the enabling step template list the conditions that are checked to determine if a transition to the next enabling state, the previous enabling state, or some other enabling state, respectively, should execute. Any actions that are performed when the transition executes are listed with the conditions. Note that there can be multiple instances of each section. The order of states in the enabling process is represented by the order that the enabling steps are listed in section 1.4.2 of the primary DOORS template.

1 User Actions
These are user actions in the enabling process. Insert the Enabling Step template here for each user action step.
2 Environmental Conditions
These are environmental conditions in the enabling process. Insert the Enabling Step template here for each environmental condition step.

Figure 3.12: A template for describing the Unordered Enabling process.

We have also created a template for describing the Unordered Enabling process (see Figure 3.12). The template merely separates the enabling process into user actions and environmental conditions. Enabling steps are included as subsections of User Actions and Environmental Conditions to describe the Unordered Enabling process.

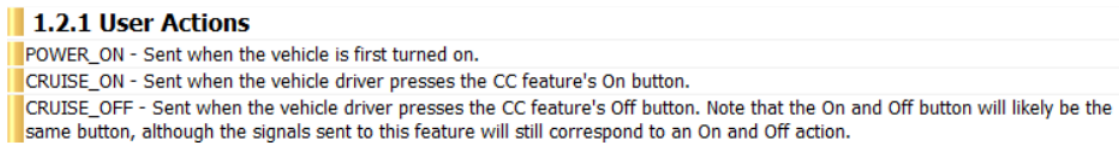
5. The Active section describes the active requirements of the feature. The General Requirements section describes any requirements of the feature that must hold while the feature is in the Active state. The remaining four subsections contain requirements specific to each of the four pattern states. The Controlling state of the feature may contain very complex behaviour so it may make sense to further split the Controlling state as necessary. We do not provide any templates for the Controlling sub-machine or the transitions within Active because that behaviour is feature specific and is difficult to anticipate. We also do not include a subsection for concurrent monitoring regions because they are unique to each feature. Additional subsections for the monitoring regions can be included at the end of the Active section.
6. The Failed section describes the requirements of the feature when it is in the Failed state. This section will likely be quite small as the Failed state's behaviour is very basic.

Features may have some kind of global requirements (e.g., safety requirements). The template does not include a section for these because we have not observed them in the requirements documents we have been given. If a section like this is going to be included, we believe it should be placed immediately after the state-machine model of the feature.

As with the pattern models, any sections that are unused can be removed from the requirements document. Also note that the templates consider only the behavioural requirements of features. Any non-behavioural requirements (e.g., quality or hardware requirements) can be added as additional sections in the requirements document.

3.3.1 Cruise Control Example

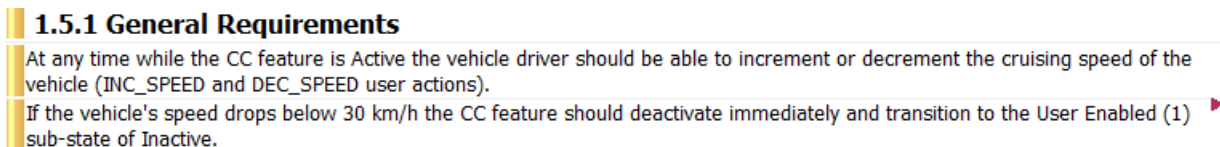
We have used the DOORS templates to create a set of requirements for the CC feature presented in this chapter. We describe parts of the example here, but the full set of requirements is included in Appendix B. Note that the requirements describe only the basic behaviour of the CC feature, in practice the requirements would likely be much more complex.



1.2.1 User Actions
POWER_ON - Sent when the vehicle is first turned on.
CRUISE_ON - Sent when the vehicle driver presses the CC feature's On button.
CRUISE_OFF - Sent when the vehicle driver presses the CC feature's Off button. Note that the On and Off button will likely be the same button, although the signals sent to this feature will still correspond to an On and Off action.

Figure 3.13: A fragment of the CC requirements showing several user action descriptions.

Figure 3.13 shows several examples of the various user actions to which the CC feature reacts. The entry for each user action describes the action's name and gives a short description about what the driver does to perform the action. We do not provide any hardware specific information (e.g., signal names) in our example, but in practice these will likely be present.



1.5.1 General Requirements
At any time while the CC feature is Active the vehicle driver should be able to increment or decrement the cruising speed of the vehicle (INC_SPEED and DEC_SPEED user actions).
If the vehicle's speed drops below 30 km/h the CC feature should deactivate immediately and transition to the User Enabled (1) sub-state of Inactive. ▶

Figure 3.14: A fragment of the CC requirements showing several of the Active state's requirements.

Figure 3.14 shows an example of the Active requirements of the CC feature. These requirements discuss how the driver can increment and decrement the cruising speed, and how CC behaves when the vehicle's speed drops below 30 km/h. The red triangle on the right side of the second requirement indicates a link, in this case, to the User Enabled (1) enabling step requirement (making it easy to follow the link and view the related requirements).

3.4 Pattern Summary

In this chapter we have described the Mode-Based Behaviour Pattern, which decomposes the behaviour of a feature according to the three modes of operation that we have observed in our industrial partner's feature requirements: Inactive, Active, and Failed. The Inactive state models the behaviour of the feature as it becomes enabled, the Active state models the behaviour of the feature as it is affecting its environment, and the Failed state models the failed behaviour of the feature. This chapter has also defined several pattern extensions for the Inactive and Active states that provide guidance on how to model a features' enabling process and active behaviour, respectively.

This chapter discussed two examples of features that are modelled using the pattern: the Cruise Control feature, which is modelled using the Ordered Enabling variant of the Inactive state and the Monitoring and Controlling states of the Primary Active Region. And the Lane Centring Control feature, which is modelled using the Unordered Enabling variant of the Inactive state, all four states of the Primary Active Region, and a concurrent monitoring region.

This chapter also described our notion of a behavioural interface for features that are modelled using the pattern. The public portion of the interface reveals if the feature is in the Inactive, Active, or Failed states and the private details of the feature is the internal details of the three high-level states. When the pattern is used by most (or all) of the features in a product line, the interface serves as a generic interface for referencing features.

Lastly, this chapter described the Rational DOORS requirement's templates that we have created for documenting the requirements of features that are modelled using the pattern and provided an example of the DOORS templates used to document the requirements of the Cruise Control feature.

Chapter 4

Case Study

This chapter presents a case study that we have performed in which we analyzed the requirements of 21 automotive features from 7 production-grade requirements documents. We created state-machine models of these features' behaviours using the pattern. In each case, we created a new state-machine model based on a feature's textual requirements rather than refactor any existing state-machine model, either because the feature's requirements document included no state-machine model or because the existing machine was missing many details. Five of the features were modelled while we refined the pattern, and the other 16 features were modelled after the pattern had stabilized. The purpose of the case study was to explore the impact of the pattern on two aspects of requirements modelling:

1. To what degree is the pattern applicable to real-world features in a product line?
2. Are the majority of inter-feature references simple queries that ask for information revealed by the proposed public interface (i.e., whether a feature is Inactive, Active, or Failed)?

4.1 Utility of the Pattern

We modelled a total of 21 features over the course of the case study and found that every feature could be modelled using the pattern. These features pertain to the vehicle's braking, heating, ventilation, and autonomous driving abilities. The features' behaviours range from very basic (e.g., a feature that applies the vehicle's brakes only when commanded) to very

complex (e.g., a feature that actively controls the position of the vehicle on the road). A short summary of all 21 features is shown in Table 4.1. The features highlighted in grey were modelled while refining the pattern. At the end of this section, we discuss in detail our models of the Adaptive Cruise Control and the Heating, Ventilation, and Air Conditioning features. The remainder of the features are discussed in Appendix A.

Table 4.2 lists the number of features that make use of the various pattern constructs. Nineteen of the features’ models include an Inactive state, 21 features’ models include an Active state, and 10 of the features’ models include a Failed state. In most cases, the reason that a feature has only two high-level states is that the feature has no specified failure requirements¹. In two cases, only an Active state is included in the feature’s state-machine model. Nine of the features’ enabling processes are modelled using the Ordered Enabling variant of Inactive, another 9 features are modelled using the Unordered Enabling variant, and 1 feature is modelled using the Hybrid Enabling variant. The remaining 2 features have no Inactive state. Most of the features that we examined have short enabling processes (i.e., one or two stages), but 3 of the features have enabling processes that have three or more stages. Our state-machine models for 19 of the features include a Controlling sub-state within the Primary Active Region; our models of the other 2 features have empty Active states because we did not have access to documents describing the features’ Active behaviours. Twelve features include a Monitoring sub-state, 5 features include a Failing sub-state, and 4 features include a Deactivating sub-state.

To give a sense of the sizes of the case-study models, we report the average numbers of states and transitions in the models (see Table 4.3). The average number of states in the case-study models is 12 and the average number of sub-states within the models’ Active state is 7. This suggests that a little more than half of the features’ requirements focus on the features’ Active behaviours. This correlates with our observations that most of the textual requirements focus on the features’ Active behaviour. As mentioned above, our state-machine models of two of the features have empty Active states because we did not have access to the features’ active requirements.

We also report the average number of transitions in the case-study models. When counting the number of transitions, we count each condition or conjunction of conditions as a separate transition, such that each transition is assumed to check only a single condition or conjunction of conditions. For example, a transition with five disjunctive clauses is counted as five separate transitions. On average, there are 27 transitions in a state machine with 17 of those being within the Active state. When counting the number of transitions

¹Many features have a separate safety-requirements document that describes how the feature behaves in the presence of failures. We did not have access to any safety-requirements documents.

Feature	Description
Adaptive Cruise Control	Maintains the vehicle's speed at a driver-set value or maintains a safe distance from a preceding vehicle
Lane Centring Control	Maintains the vehicle in the centre of its current lane
Lane Change Control	Automatically changes the vehicle's lane as directed by the driver
Forward Collision Alert	Alerts the driver if the vehicle is approaching a preceding object too quickly, such that a collision is imminent
Road Change Alert	Alerts the driver if any of the road's lanes are forking or merging
Automatic Braking	Applies the vehicle's brake when requested by other features (the driver does not directly interact with this feature)
Anti-Lock Braking System	Prevents the vehicle's tires from locking when the driver is braking
Active Trailer Stability Assist	Keeps the vehicle stable when towing a trailer
Air Quality System	Minimizes the pollution in the vehicle cabin by controlling the amount of recirculated versus fresh air
Air Recirculation Control	Regulates the blending of recirculated air and fresh air in the vehicle's cabin – the requirement's documents we have examined do not discuss how the Air Recirculation Control and Air Quality System features interact
Brake Assist	Applies the brakes with 100% force when the driver performs an emergency stop
Brake Cleaning	Cleans buildup from the vehicle's brakes by scraping the brake pads along the brake disc while the vehicle is in motion
Electric Park Brake	Controls the vehicle's parking brake using a dashboard button
Enhanced Traction System	Prevents the vehicle from losing traction with the road by modifying the engine torque
Heating, Ventilation, and Air Conditioning	Maintains the vehicle's temperature at a driver-set value
Hill Hold	Prevents the vehicle from rolling backwards while on a hill
Lane Keep Assist	Keeps the vehicle from leaving its current lane
Manual Park Brake	Controls the vehicle's parking brake
Recirculation Control Run	Ensures that the vehicle's windows do not fog
Traction Control System with Electronic Stability Assist	Prevents the vehicle from losing traction with the road by controlling the engine torque and applying the vehicle's brakes
Competitive Traction Control System with Electronic Stability Assist	Same basic behaviour as the Traction Control System with Electronic Stability Assist; provides several settings for how and when to activate the traction control

Table 4.1: The 21 features that we modelled over the course of the case study.

Pattern construct	Number of Features using each construct /21
Inactive state	19
Active state	21
Failed state	10
Ordered Enabling process	9
Unordered Enabling process	9
Hybrid Enabling process	1
No enabling process	2
Controlling state	19
Monitoring state	12
Failing state	5
Deactivating state	4

Table 4.2: The number of features that use the various pattern constructs.

State or Transition count	High	Average	Low
Number of states	40	12	5
Number of sub-states within Active	37	7	0
Number of transitions	155	27	5
Number of transitions within Active	134	17	0

Table 4.3: The average number of states and transitions in each feature.

in the Active state, we include transitions that originate within the Active state but whose destination is outside of that state. These results also suggest that a majority of the features' requirements focus on the features' Active behaviours. Two case-study models have no Active behaviour, thus the lowest number of transitions within Active is zero.

Table 4.4 lists the number of features that are modelled using each pattern variant. Six features initialize in the Active state rather than in the Inactive state. Two features, when they deactivate, transition to a partially enabled sub-state within Inactive rather to the initial sub-state in Inactive. Three features skip one or more enabling stages (1 feature has an override that skips forward several stages and 2 features have disabling conditions that cause the feature to skip backwards several stages in the enabling process). On entry to the Active state, 9 features initialize in the Controlling sub-state of Active and 10 features initialize in the Monitoring sub-state. No feature transitions from the Failing sub-state back to the Controlling sub-state. One feature contains multiple instances of the Primary

Pattern variant	Number of Features using each variant /21
Initializes in the Active state	6
Deactivates to a partially enabled state	2
Skips one or more enabling stages	3
Initializes in the Controlling sub-state	9
Transition to Controlling from Failing	0
Multiple instances of the Primary Active Region	1

Table 4.4: The number of features that are modelled using a pattern variant.

Active Region.

We highlight two of the case-study features (the remainder of the features are discussed in Appendix A). To avoid revealing proprietary information, we have abstracted away many details of each feature. For example, many details of the Active state are omitted, and sets of transitions between states are represented by singleton transitions whose labels document the combined number of transition conditions. In the models, events are expressed in upper-case text, the number of conditions are in bold font, and all other conditions are in lower-case text (the models in Appendix A also follow this convention).

4.1.1 Example — Adaptive Cruise Control

The feature in the case study with the largest number of enabling conditions is Adaptive Cruise Control (ACC). Once activated, the ACC feature will maintain the vehicle’s speed at a driver-set value and will maintain a safe distance from the preceding vehicle. Enabling the ACC feature is a multi-stage process: the user turns on the feature, after which several environmental conditions are checked, and finally the user must perform one of two possible actions to complete the enabling process.

The ACC feature’s enabling process is modelled using the Ordered Enabling variant of the Inactive extension (see Figure 4.1). There are three transitions from the Active state back to the Inactive state because, depending on the deactivation conditions, the ACC feature will either deactivate completely and the enabling process will begin anew, or it will transition to a partially enabled state.

The active behaviour of the ACC feature is modelled using all four of the Primary Active Region’s sub-states. The primary region of the Active sub-machine initializes in the Monitoring sub-state and transitions to the Controlling sub-state when the feature

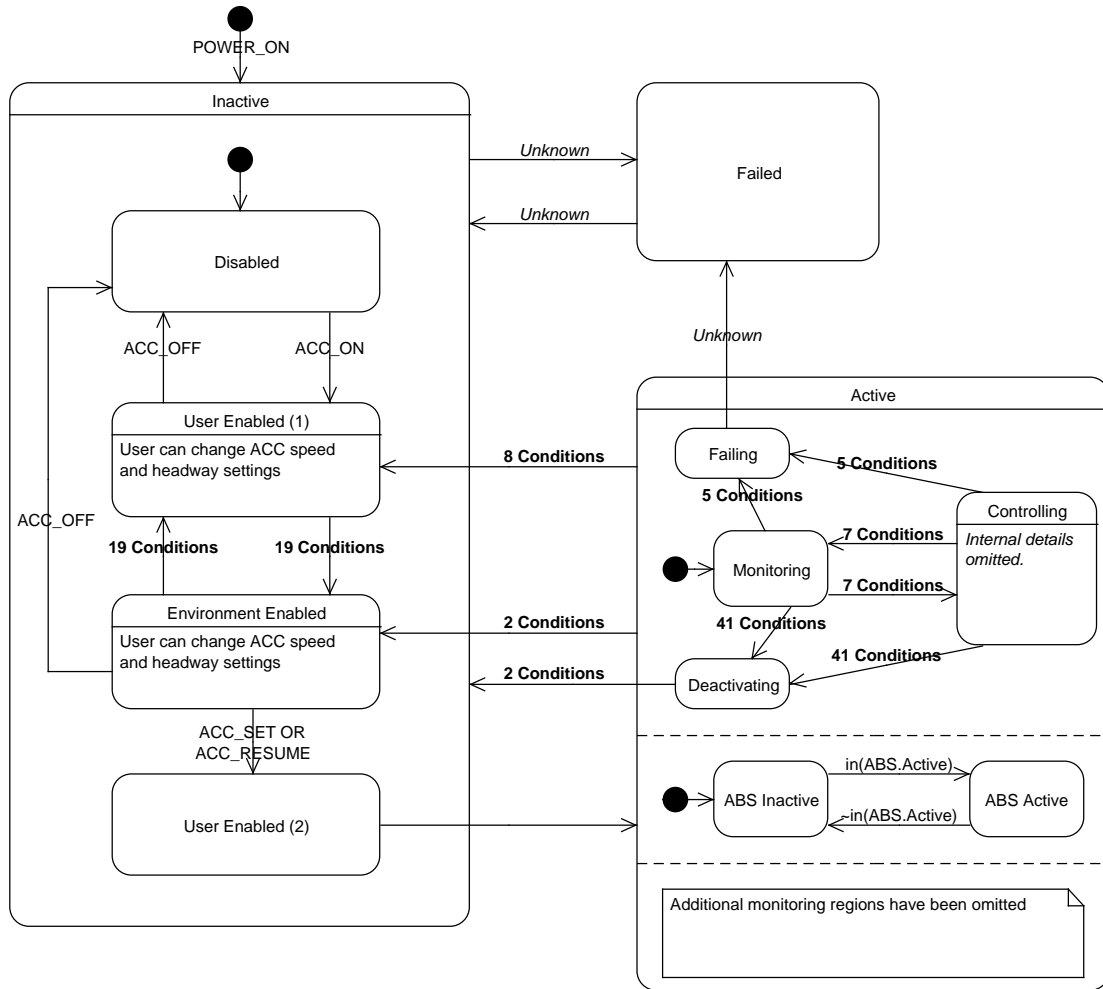


Figure 4.1: Adaptive Cruise Control (ACC) feature.

should actively control the vehicle’s speed (we have omitted the internal details of the Controlling sub-state for confidentiality reasons). One of the conditions checked in the Monitoring sub-state is the distance to the preceding vehicle. If the preceding vehicle is too close, then the ACC feature reduces the vehicle’s speed.

Concurrent monitoring regions in the Active state are used to monitor information that the entire Active state requires (e.g., whether the ABS feature is currently active: if the ABS feature is active, then the ACC feature deactivates regardless of its current sub-state within Active). Our case-study model of the ACC feature includes eight concurrent monitoring regions within the Active state, most of which are not shown. The omitted concurrent regions all have structures that are similar to the one monitoring region shown in the model, but they check different conditions. Within the Deactivating sub-state, the ACC feature has no ability to apply the throttle and instead controls the vehicle’s speed using only the brakes. After a delay, or if the vehicle driver disables the ACC feature, the state machine transitions to the Inactive state.

The feature’s requirements document mentions several possible failure conditions that we have modelled as transitions to the Failing sub-state (e.g., the ACC feature should fail if the brake system fails). The feature’s behaviour within the Failing sub-state is assumed to be similar to its behaviour within the Deactivating sub-state, but these details are not described in the feature’s requirements document. Also, the requirements document does not specify the feature’s behaviour when it has Failed², so we have not attempted to label the transitions to and from the Failed state.

4.1.2 Example — Heating, Ventilation, and Air Conditioning

The case-study feature with the most complex Active behaviour is Heating, Ventilation, and Air Conditioning (HVAC) (see Figures 4.2 and 4.3). The HVAC feature maintains the vehicle’s cabin temperature at a user-set value, by using the heater (to increase the cabin temperature) and the air-conditioning system (to decrease the cabin temperature). The HVAC feature also minimizes the amount of pollution in the vehicle’s cabin by controlling the blend of fresh air and recirculated air.

Our state-machine model of the HVAC feature contains 40 states (the highest number of states of the features that we examined) and has 53 transition conditions. The HVAC requirements documents do not describe the feature’s enabling process in any detail so

²Many features have a separate safety-requirements document that describes how the feature behaves in the presence of failures. We did not have access to any safety-requirements documents.

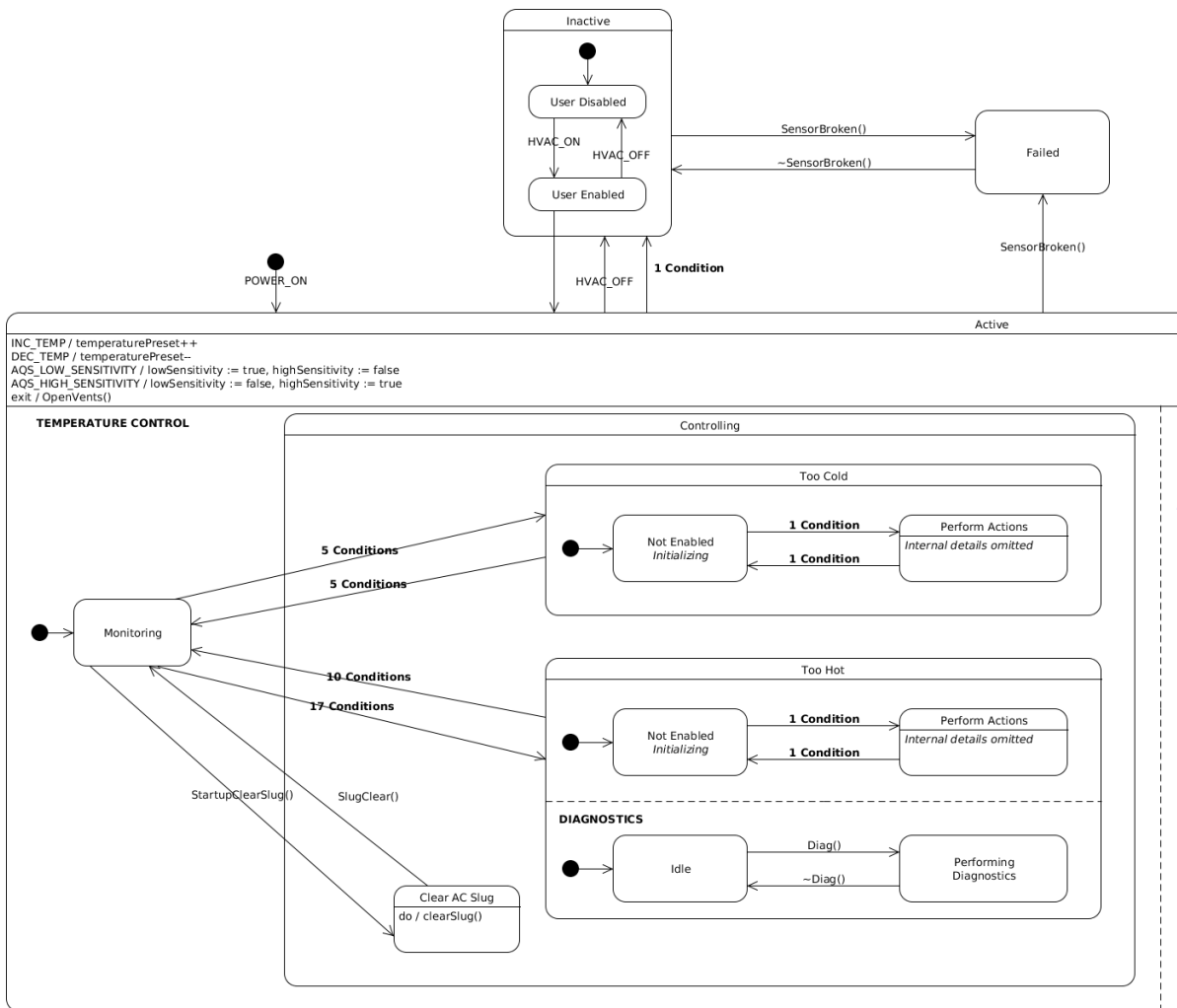


Figure 4.2: The Heating, Ventilation, and Air Conditioning (HVAC) feature.

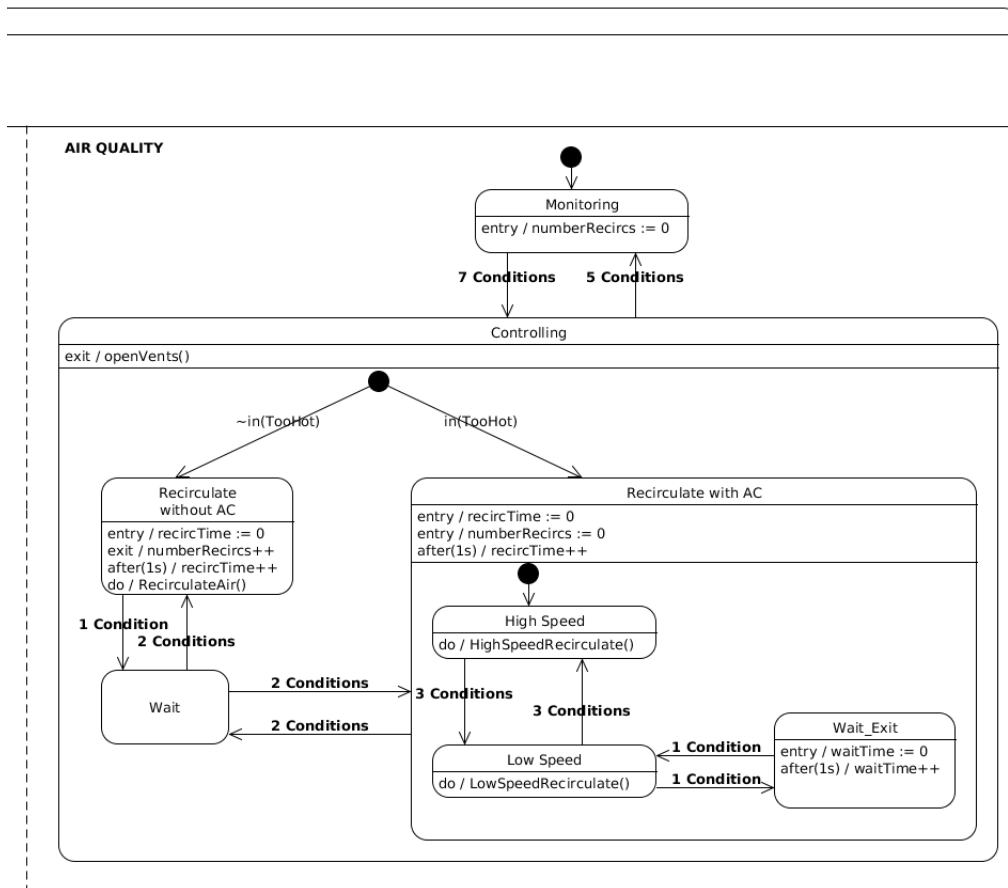


Figure 4.3: The Heating, Ventilation, and Air Conditioning (HVAC) feature.

we have modelled only a single user action that activates the feature (i.e., pressing the HVAC_ON button). This simple enabling process is modelled using the Ordered Enabling variant of the Inactive extension. Within the Active state, two conditions are checked to determine if the HVAC feature should deactivate (one of these is the user action of pressing the HVAC_OFF button). While the HVAC feature is active, the driver or a passenger can set the desired cabin temperature and the sensitivity of the pollution sensors by pressing buttons on the dashboard.

The state-machine model of the HVAC feature uses two instances of the Primary Active Region: one that implements *Temperature Control* and one that controls *Air Quality*. The Temperature Control sub-machine initializes in the Monitoring sub-state. If there exists any buildup from previous executions (referred to as the *AC Slug*), the machine transitions to the Controlling sub-state, removes the buildup, and returns to the Monitoring sub-state. During normal operation, the Temperature Control sub-machine transitions from the Monitoring sub-state to the Controlling sub-state if the cabin temperature becomes either *Too Cold* or *Too Hot*. Depending on the sub-state of Controlling that the machine is in, it will utilize the vehicle’s heater or air conditioner to raise or lower the vehicle temperature, respectively. Both the Too Cold and Too Hot sub-machines initialize in a Not Enabled sub-state in which the respective sub-machine initializes the relevant systems. Each sub-machine transitions to its Perform Actions sub-state, which implements the heating or cooling logic, when the initialization process has completed.

The Air Quality sub-machine initializes in the Monitoring sub-state and transitions to the Controlling sub-state when the pollution level becomes too high. When in the Controlling sub-state, the HVAC feature’s behaviour depends on whether the vehicle’s air conditioner is active. The HVAC feature limits the amount of time that the cabin air can be recirculated before the fresh air vents are opened; the Recirculate with AC and Recirculate without AC sub-states increment a timer and the sub-machine transitions to either the Wait or Monitoring sub-states when fresh air needs to be let into the vehicle.

Our state-machine model for the HVAC feature includes 17 additional concurrent monitoring regions, but we have not included these because of confidentiality reasons and because the size of the model would be very large. The HVAC requirements do not discuss failures so our model does not include a Failed state.

The HVAC feature was the last set of requirements that we were given by our industrial partner. The engineers gave us these requirements as somewhat of a stress test after we presented our initial findings from the case study. The HVAC requirements are very different from the other feature requirements that we examined. It was relatively simple to model the other features’ behaviours using the pattern because each of those features had

some high-level description of the feature’s behaviour followed by detailed descriptions of the functionality that the feature provides. In contrast, the HVAC feature’s requirements did not include any kind of high-level descriptions of behaviour. This made it relatively difficult to model the HVAC requirements because it was hard to determine how the functions that the HVAC feature provides relate to one another. In addition, we expected the HVAC feature’s requirements to comprise certain behaviours based on our own knowledge of what functionality an HVAC feature typically provides (e.g., the user being able to control fan speed or completely turn off the air conditioner). Because the HVAC feature presented in the requirements document did not match our mental model of the feature, it was harder to understand the requirements on first examination.

We believe that there may be three reasons that explain the difference between the HVAC feature and the other case-study features:

- The requirements of the HVAC feature that we were given were a work-in-progress and the high-level descriptions of the feature’s behaviour and enabling process may not have been finished.
- The HVAC feature is owned by a team different from the other features that we examined, so it may be the case that this team has a different method for structuring and defining their requirements.
- The requirements for the HVAC feature are being documented in Rational DOORS. Within DOORS, it is very easy to describe functional behaviour using text, which may have influenced the way in which the HVAC’s requirements were specified.

Looking again at the use of two instances of the Primary Active Region to model HVAC’s active behaviour (one instance for controlling the temperature and one instance for controlling the cabin’s air quality), we see that there is very little overlap between the behaviours modelled in each region. In fact, the only overlap is that the Air Quality region checks if the air conditioner is currently being used, by checking whether the Temperature Control region is in its Too Hot state. The benefit of modelling these orthogonal behaviours as a single feature is that there is no need for an inter-feature reference. The downside is that the state machine is performing two orthogonal tasks. Because of the minimal overlap, we recommend modelling the HVAC feature as two distinct features: one (called HVAC) that controls only the cabin temperature (see Figure 4.4) and one (called Air Quality System (AQS)) that regulates the pollution in the cabin (see Figure 4.5). Note that in this thesis, when we provide metrics regarding the case-study models (e.g., the average number

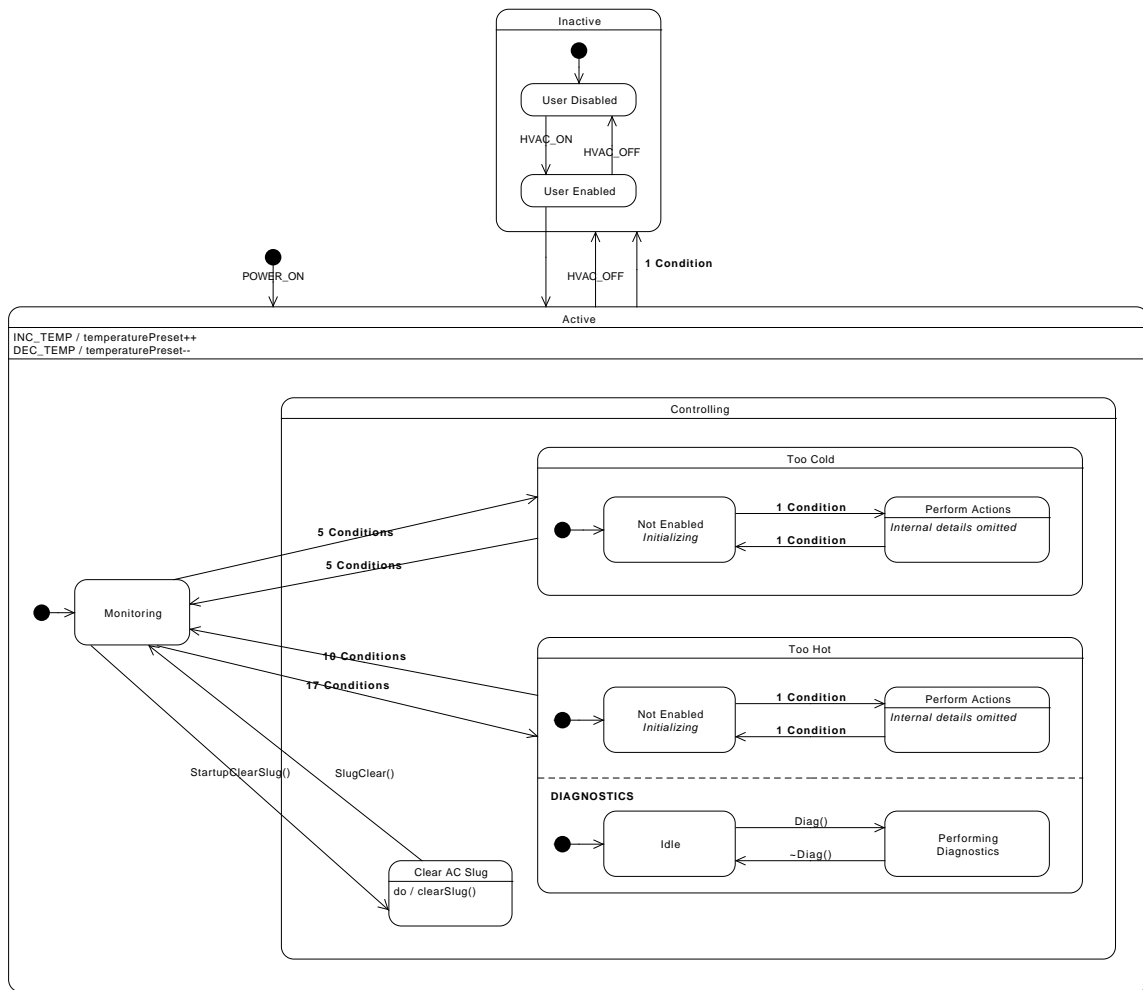


Figure 4.4: The Heating, Ventilation, and Air Conditioning (HVAC) feature modelled without air quality control.

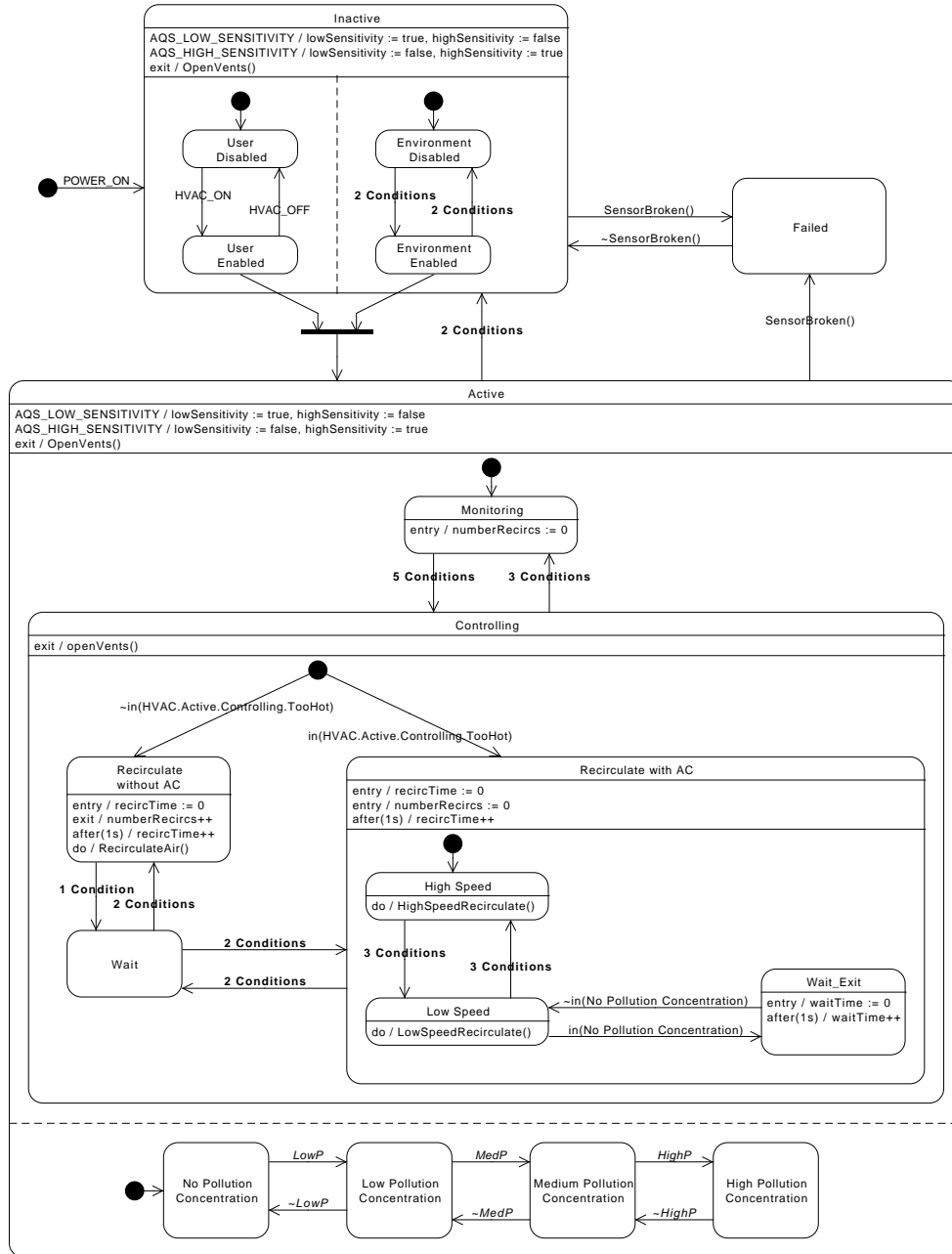


Figure 4.5: The Air Quality region of HVAC modelled as the Air Quality System (AQS) feature.

of states in the features' state-machine models), we count the HVAC and AQS features as two separate features.

Our new state-machine model of the HVAC feature in Figure 4.4 models the same requirements as the Temperature Control region of our original model of the HVAC feature. The new HVAC state-machine has identical Inactive and Active states as the original HVAC machine. The only difference is that the Failed state is not present in the new HVAC model because the failure requirements are unique to AQS.

Our state-machine model of the AQS feature in Figure 4.5 captures the same requirements as the Air Quality region of our original model of the HVAC feature, but there are several differences between the AQS machine and our original model: (1) Our original model included an in-state reference that checks if the air conditioner is active; in our model of the AQS feature, this check is modelled as an inter-feature reference to a state in the new HVAC feature. (2) The enabling process of the AQS feature is modelled as an unordered process that checks whether the HVAC_ON action has occurred and whether two environmental conditions³ are true. In the original model, those environmental conditions were modelled on the transitions between the Monitoring and Controlling sub-states of the Air Quality region because they are unique to the Air Quality requirements.

The benefit of modelling HVAC and AQS as separate features is that each feature has only one essential responsibility (indicated by having only one instance of the Primary Active Region). We believe that having a single Primary Active Region in a feature is preferable because it simplifies the resulting state-machine model.

4.2 Generality of the Public Interface

We have examined our case-study models to evaluate whether the pattern is viable as a public interface. Specifically, we examined: (1) how many features reference other features and (2) how often the references are to elements of the public interface.

In our industrial partner's requirements, every feature belongs to a *sub-system* which categorizes a set of related features. For example, the ACC feature presented earlier is part of the Freeway Limited Ability Autonomous Driving Features (FLAAD) sub-system. We list the features according to their sub-systems in Appendix A. Two features are loosely coupled if they reference one another's public-interface information, and that they are

³One condition checks whether the AQS feature has been calibrated to be available, and the other condition checks whether the outside air temperature is above some lower limit.

tightly coupled if they reference one another's internal details not revealed by their public interfaces. We have a notion of preferred versus unpreferred references between features. A preferred reference is one between loosely-coupled features or between tightly-coupled features from the same sub-system. An unpreferred reference is one between tightly-coupled features from different sub-systems. We say that certain references are unpreferred because they indicate that features with little in common (e.g., features that interact with entirely different parts of the system and are developed by different teams or organizations) are closely related. If one of the features is changed without the team working on the other feature being aware of the change, then the requirements of the unchanged feature could become incorrect without any warning.

Each of the five features that were modelled while the pattern was being refined has between 1 and 14 references to other features. These features are primarily interested in whether other features are Inactive, Active or have Failed. However, three of the features have a small number of references to other features' detailed states (discussed in more detail below).

Of the sixteen features that we modelled after the pattern stabilized, only eight of the features have references to other features and all but one reference are to high-level states in the other features' public interface. The eight features each have one to six references to other features. We hypothesize that the reason the latter features have fewer inter-feature references is that they are low-level features, whereas the initial five features are high-level features that build on the behaviour of low-level features. We need more data to confirm this hypothesis.

We mentioned that three features in the case study make inter-feature references to information not made available in other features' interfaces. In total, there are eight such references: five that read information and three that override some behaviour. Of the references that read information, four are to features in the same sub-system as the referencing feature. The remaining reference is to a feature outside of the referencing feature's sub-system, to obtain information that is required to perform an algorithmic calculation. The references that override behaviour are all to features within the referencing feature's sub-system and occur in cases where the referencing feature is also reading information from the referenced features.

In summary, there are 58 inter-feature references, of which 50 are references to features' interface data. Seven references refer to private details of features in the same sub-system. Only one reference violates our convention of how features ought to behave and accesses private information from a feature outside of the referencing feature's sub-system.

4.3 Threats to Validity

The major threat to the validity of the case studies' conclusions is the narrow domain from which we have gathered our feature requirements. As we are exploring features from only one company in one domain, it is possible that there is some underlying reason, that we are unaware of, for why the features can all be modelled using the pattern. However, there are several differences between the various requirements we have examined that seem to indicate that the pattern is generally applicable to automotive features:

- The requirements documents that we examined are from at least two different teams within our industrial partner.
- All seven requirements documents that we have been provided are authored by different people.
- The features that we examined come from three different sub-systems.
- The pattern is applicable to both high-level and low-level features.
- The HVAC feature could be modelled using the pattern despite all of its differences from other features that we examined.

We have also had discussions with researchers working with other automotive companies and they have indicated that they believe that the pattern is applicable to the features they have examined. Because we have examined only automotive features during the case study, we do not claim that the pattern is applicable beyond the automotive domain, but we believe there is nothing automotive specific about the pattern.

The other threat is that the pattern's author was the one who performed the case study. This introduces the possibility of bias because the pattern's author may have ignored some aspects of feature behaviour that did not nicely fit into the pattern. However, engineers at our industrial partner have seen presentations of some of our case-study models and have fully detailed versions of all of our models. They have not suggested that our models are oversimplifications of their features' requirements.

4.4 Case Study Summary

This chapter’s introduction presented the two research questions to be addressed by the case study:

1. Question: To what degree is the pattern applicable to real-world features in a product line?
 - The results of the case study with respect to this question are favourable. Although we modelled only 21 features, the pattern was applicable to all 21. These features come from at least two different teams within our industrial partner, and the features belong to three different sub-systems within the vehicle.
2. Question: Are the majority of inter-feature references simple queries that ask for information revealed by the proposed public interface (i.e., whether a feature is Inactive, Active, or Failed)?
 - Fifty out of 58 references in the case study are only interested in whether a feature is Inactive, Active, or has Failed. This indicates that the interface reveals useful information without revealing a feature’s detailed behaviour. Given that the pattern’s wide applicability to features seems promising, the high-level behaviour modes have the potential to serve as a generic interface for all of the features that adhere to the pattern.

For future work, we would like to perform a case study involving features from a different domain (e.g., embedded aerospace systems) to assess whether the pattern is applicable beyond the automotive domain.

Chapter 5

User Study

This chapter presents a user study that was performed to evaluate the benefits that the pattern provides to the reviewers and specifiers of feature requirements. The user study was designed to answer three questions that we have about expected benefits:

1. Does the pattern and interface aid a requirements reviewer in interpreting state-machine models?
2. Does the pattern and interface aid a modeller in writing correct and readable state-machine models?
3. Does the pattern and interface improve the confidence of the requirements reviewer and specifier?

Appendix C includes all of the user-study materials.

5.1 Performing the Study

The study was conducted in two phases. Each phase performed a nearly identical study but was performed at two different times. In this thesis, we aggregate the data from both study phases into one set of results and will discuss the minor differences between the two studies when they arise.

The first phase's participants were drawn from computer science graduate students at the University of Waterloo. These participants were recruited through an e-mail campaign

targeting students who would likely be familiar with state-machine modelling. The second phase’s participants were drawn from upper-year computer science undergraduate students and graduate students at the University of Waterloo. As in the first phase of the study, we used an email campaign to recruit participants who would likely be familiar with state-machine modelling. The reason we recruited participants who had state-machine modelling experience is that we did not want the study to be a participant’s first exposure to state-machine modelling. We did not expect participants to have any experience with automotive features but we did not disqualify those who did.

Participants were randomly placed into three groups: *Control (C)*, *Pattern (P)*, and *Pattern+Interface (PI)*. Members of the C group knew nothing about the pattern and interface. Members of the P group knew about the pattern but not the interface. Members of the PI group knew about both the pattern and the interface.

Eighteen participants took part in the user study, such that each group had six participants. Fourteen of the participants were graduate students and four were undergraduate students. Each participant was given a tutorial and after completing the tutorial, was given the main study.

5.2 Tutorial

Each participant was given a one-hour tutorial to complete on his or her own time. The tutorial introduced the automotive domain and the vehicle environment (using a domain model), and provided participants a review of many of the advanced aspects of state-machine modelling (nearly identical to what we presented in Section 2.1). The version of the tutorial given to members of the P group also described the pattern, and the version of the tutorial given to members of the PI group described both the pattern and the interface. All three versions of the tutorial include an example model of the Road Change Alert (RCA) feature (all three models are included in Appendix C). The C group’s state-machine model of the RCA feature is adapted from a state-machine model provided by our industrial partner. The state-machine models for the PI and P groups capture the same requirements as the C group’s model and are modelled using the pattern. The enabling process of the RCA feature is Unordered, and the Active state uses the Monitoring and Controlling sub-states of the Primary Active Region. The RCA feature’s enabling process checks that the MAP Information feature is active before it can activate; this reference is modelled in the PI group’s state-machine model using the public interface of the MAP Information feature. The last part of the tutorial asked participants to model the behaviour of an Adaptive Headlights (AH) feature that turns the vehicle’s headlights on or off depending

on the ambient light level. The exercise was exactly the same in all three tutorials. In particular, the exercise did not ask the participant to apply the pattern or the interface.

The purpose of the tutorial exercise was to give participants practice with the modelling notation, pattern, and interface. This way, the study would not be a participant's first working experience in applying these technologies. The tutorial exercise also gave participants an extra example to refer to during the main study. Before giving the participants the main study, we confirmed that he or she completed the tutorial exercise but we did not provide any feedback to avoid giving some participants more individual information (about state-machine modelling, the pattern, or the interface) than we gave others.

The differences between each study were confined to the descriptions of the pattern and the interface. Therefore, we argue that the difference in the tutorials given to each study group are not a threat to the validity of the study because all of the tutorials provide the same examples and exercises to the participant.

5.3 Main Study

The main study is a written exercise that consisted of three sections: Section 1 asked the participant to provide background information about his or her experience with state-machine modelling and automotive modelling, Section 2 asks the participant to review a state-machine model and answer several questions about it, and Section 3 provides a textual description of a feature's behaviour and asks the participant to create a state machine that models that behaviour. The study is identical for all three groups of participants, except that the model that each participant was asked to review varied depending on his or her participant group: the C group's state-machine model was not structured using the pattern, the P group's model was structured using the pattern, and the PI group's model was structured using the pattern and there are two references to other features that refer to that feature's public interface. We asked participants to spend a maximum of 90 minutes on the study (with a maximum of 60 minutes on Section 3).

Throughout this chapter, we show the correlation between study variables using Pearson correlation [42]. In this work, we say that two variables are sufficiently correlated if their Pearson correlation value is greater than 0.6.

5.3.1 Participant Background

The first section of the study asked the participants three questions to determine his or her knowledge of state-machine modelling and automotive software. We gathered this information because we wanted to determine whether use of the pattern and interface improves readability and writability of state-machine models, and we wanted to rule out modelling experience as the primary determinant of better performance. The three questions that we asked of each user study participant were:

1. Do you have previous experience with requirements modelling or state-machine modelling? If so, briefly state the types and levels of experience (include notations, methods, and tools used, length of time used, and whether your experience is from coursework or industrial experience).
2. What is your experience with modelling automotive features (if any)?
3. On a scale of 1 to 5, express your level of comfort with UML State Machines or statecharts (1=never heard of them, 2=heard of them and have looked at some models, 3=used the notation in the past but do not recall a lot of details, 4=can probably sketch a model, and 5=have a good knowledge of them).

The first question asked each participant to describe his or her experience with state-machine modelling. We asked this question so that we could preclude the possibility that a group's performance was due to the increased experience of that group's participants. Figure 5.1 shows the experience of the participants in each study group. In each group, the majority of participants have some experience with state-machine modelling. The PI and P groups each had one participant with no state-machine modelling experience, and the C group had two participants with no experience. All of the participants who had no experience with state machines did have experience with deterministic and non-deterministic finite-state automata. One participant in the PI group and two participants in the C group had industrial experience with state-machine modelling. One of the C group's participants stated that his or her industrial experience was with automotive software.

All of the participants who had experience with state-machine modelling stated that they are familiar with the UML State Machine language. Other modelling notations that participants mentioned experience with are: Simulink¹ and SysML [36].

¹<http://www.mathworks.com/products/stateflow/>

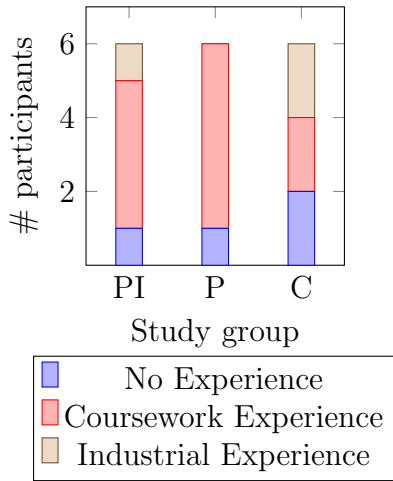


Figure 5.1: The previous experience with state-machine modelling of all of the study participants in each group.

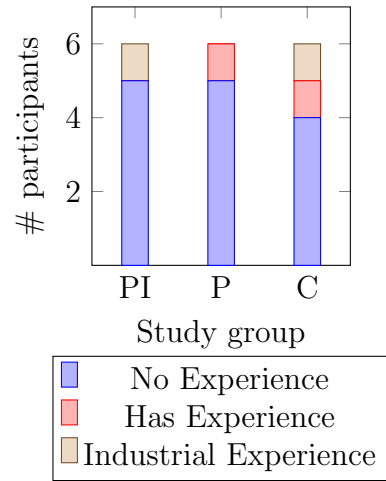


Figure 5.2: The previous experience with automotive modelling of all of the study participants in each group.

We found no correlation between a participant’s experience and any aspect of his or her performance in the state-machine comprehension or state-machine modelling portions of the study.

The second question asked each participant to describe his or her experiences with automotive modelling. Four of the 18 participants stated that they have experience modelling automotive software (see Figure 5.2): one participant in the PI group, one participant in the P group, and two participants in the C group. Figure 5.3 shows the relationships between each participant’s experience with state-machine modelling, automotive modelling, and industrial modelling. The vast majority of participants only had experience with state-machine modelling in an academic setting. One of the PI group participants and one of the C group participants had industrial experience with state-machine modelling, and experience with automotive modelling; the C group participant explicitly stated that his or her automotive state-machine modelling experience was in an industrial setting. One of the P group participants had experience with automotive state-machine modelling in a coursework setting. Lastly, the C group participant with state-machine modelling experience and automotive modelling experience stated that his or her experience was with an automotive-specific regular-expression parser and text editor (the participant did not elaborate any further).

We found no correlation between a participant’s experience with automotive modelling and any aspect of his or her performance in the remainder of the study.

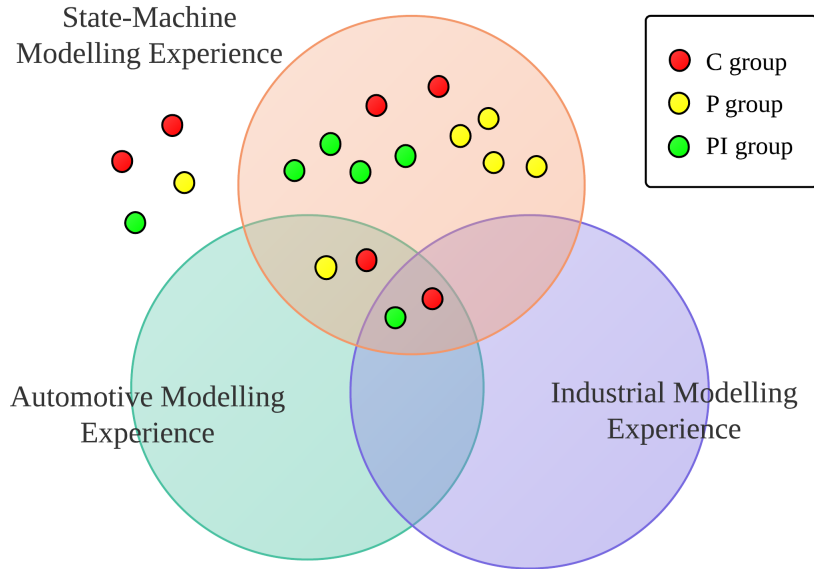


Figure 5.3: A Venn diagram showing the relationships between each study participant’s experience with state-machine modelling, automotive modelling, and industrial modelling.

The final question asked participants to rank how comfortable he or she is with UML State Machines or statecharts. We provided five rankings: (1) never heard of them, (2) heard of them and have looked at some models, (3) used the notation in the past but do not recall a lot of details, (4) can probably sketch a model, and (5) have a good knowledge of them. The average comfort level for the PI group participants was 3.16, the average comfort level for the P group participants was 2.83, and the average comfort level for the C group participants was 3.16. The distribution in each group is shown in Figure 5.4. The distribution of comfort is fairly even across all participant groups with the majority of the participants in each group stating that he or she is familiar with state-machine modelling but does not know many details.

We found no correlation between a participant’s level of comfort with state machines and the correctness of his or her solution to the state-machine comprehension or state-machine modelling portions of the study. However, we did find that those participants who ranked themselves as having a high comfort tended to have greater confidence in their solutions (Pearson correlation = 0.61, we will discuss this again in Section 5.3.4).

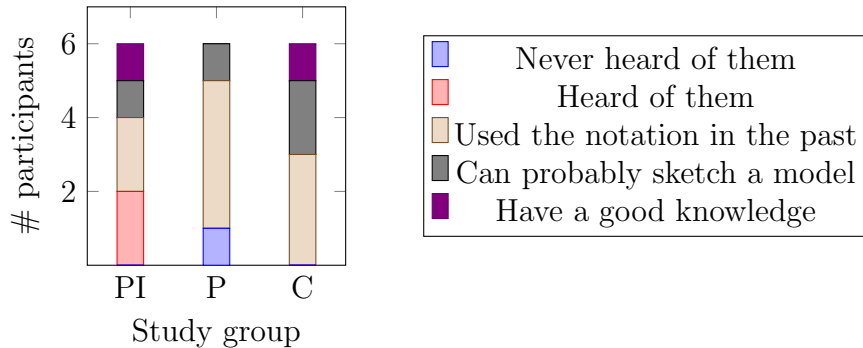


Figure 5.4: Each group’s participant’s comfort with state-machine models.

Overall, the participants with greater experience with state-machine modelling or automotive modelling and those participants with high levels of comfort with state-machine modelling were evenly distributed among the three study groups. Therefore, we do not believe that any of these factors affected the validity of the study results.

5.3.2 State-Machine Comprehension

The second study section tested a participant’s ability to read and understand the behaviour of a state-machine model. We tested this by providing participants with a state-machine model of an Adaptive Cruise Control (ACC) feature and asking them to answer several questions about its behaviour. The requirements for this feature come from a course project used in a past offering of an undergraduate computer science course on software requirements² and was not based on the requirements provided to us by our industrial partner. The structure of the provided state-machine model was different for each participant group: the state-machine model provided to the PI group is shown in Figure 5.5, the state-machine model provided to the P group is shown in Figure 5.6, and the state-machine model provided to the C group is shown in Figure 5.7. The PI and P group’s models use the Ordered Enabling variant of Inactive and use all four sub-states of the Primary Active Region. The PI group model references the public interface of two other features (we explain this in more detail below). The state-machine model provided to the C group’s participants (see Figure 5.7) was created by the students as part of their course project, so they had lots of time to work on the layout and presentation of their model. We used

²The course is *CS445: Software Requirements and Specification*, taught at the University of Waterloo in the Winter 2012 semester.

the ACC state-machine from the highest-ranked group. The only changes that we made were to the transition conditions to standardize them across all three models and we also fixed any syntax errors.

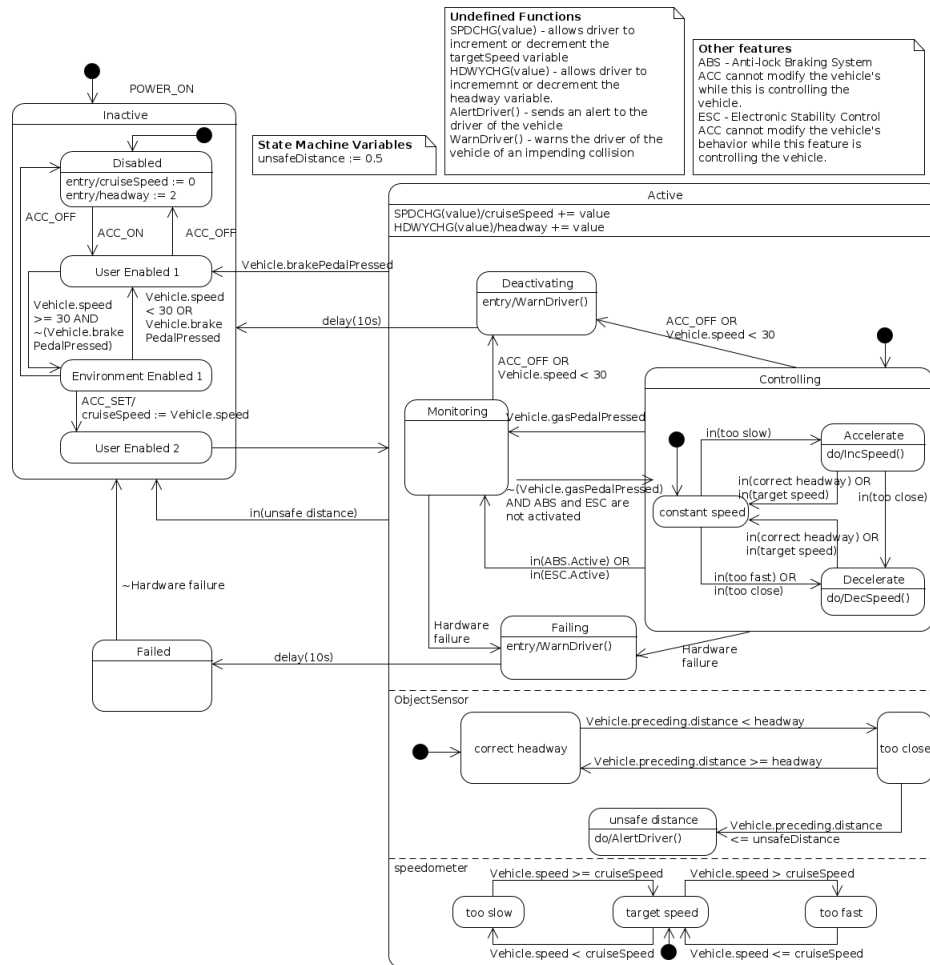


Figure 5.5: The Adaptive Cruise Control feature’s state-machine model provided to the PI group’s participants.

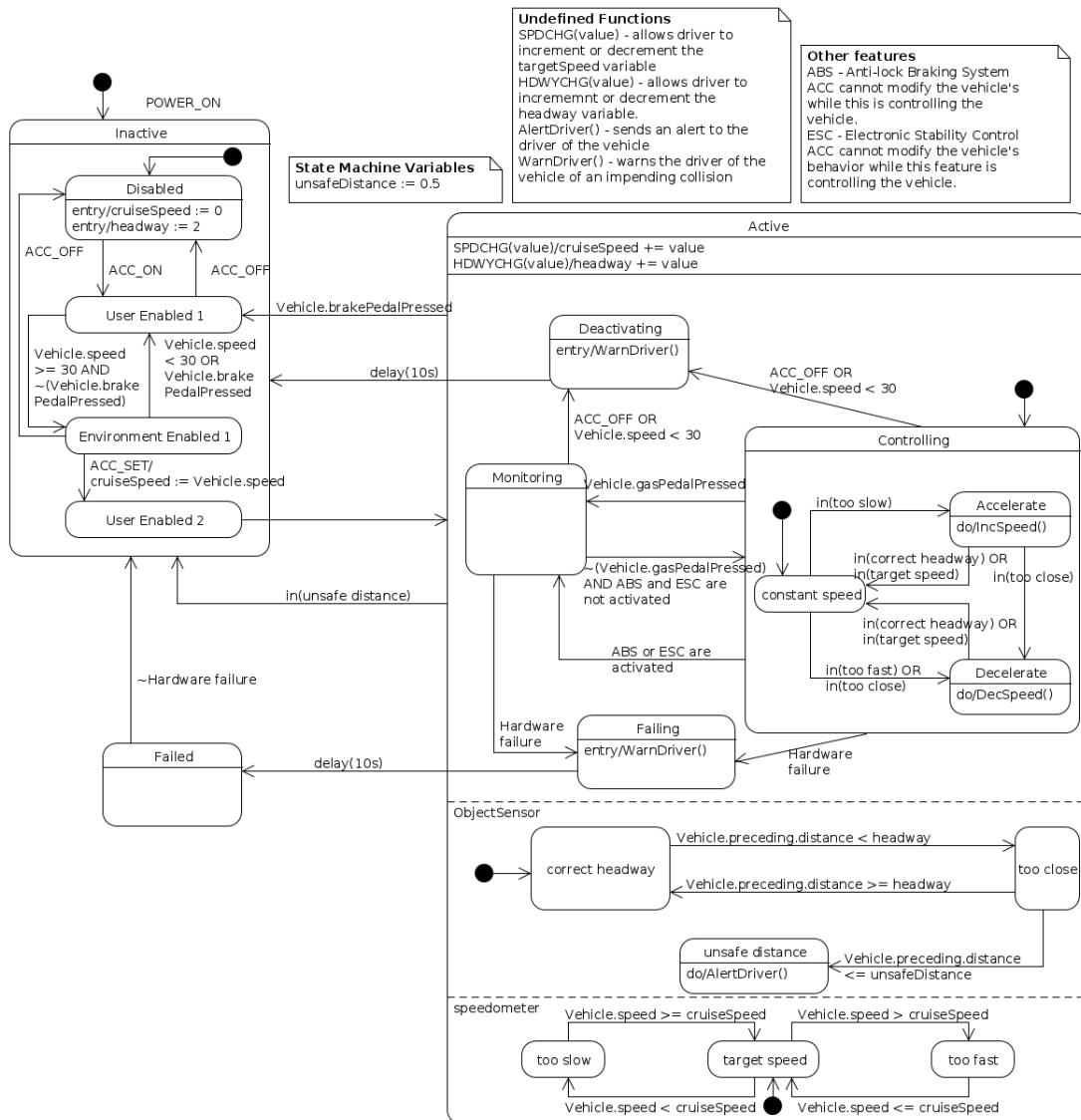


Figure 5.6: The Adaptive Cruise Control feature's state-machine model provided to the P group's participants.

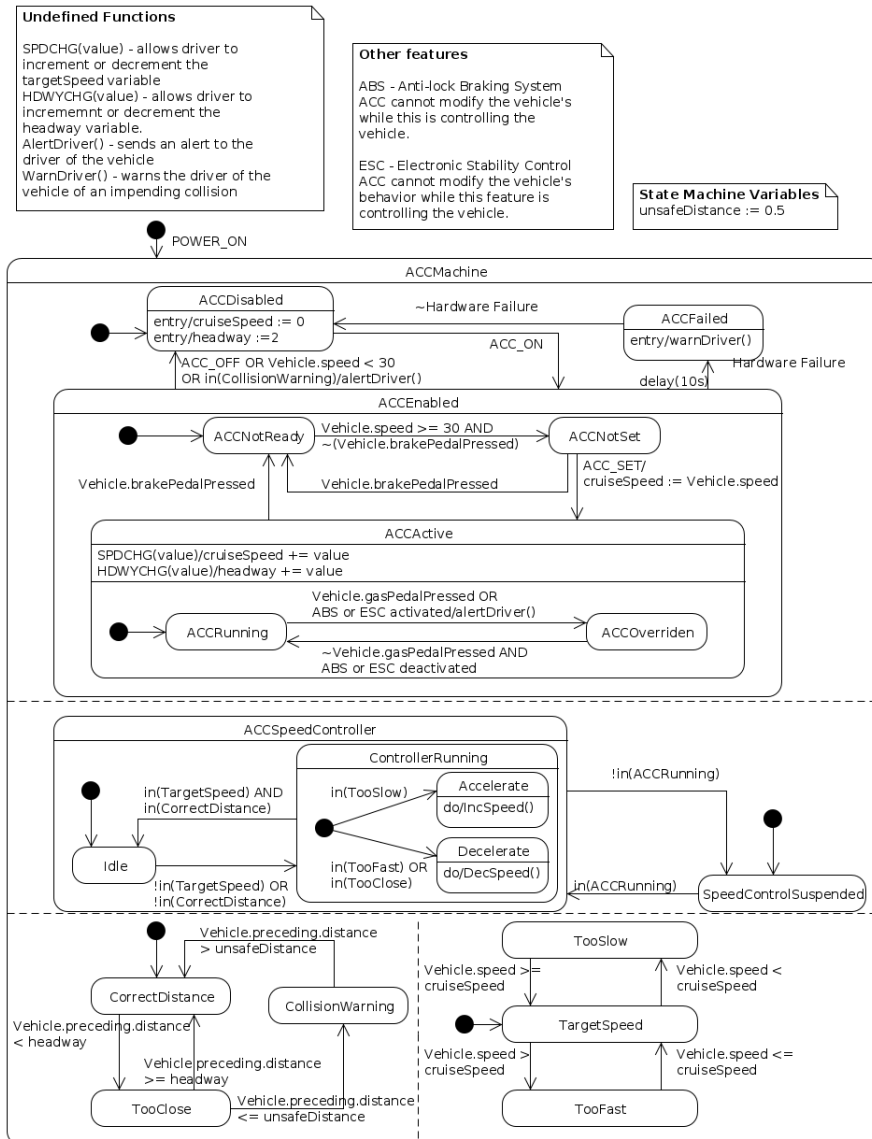


Figure 5.7: The Adaptive Cruise Control feature's state-machine model provided to the C group's participants.

Table 5.1 lists the six questions that the study asked each participant about the provided model of the ACC feature.

The first question asks the participant to list all of the environmental conditions that need to hold for the ACC feature to remain active. Table 5.1 lists the four conditions. In order for the participant's response to be correct, he or she must list all four conditions and not list any extra conditions.

The second question asks the participant to list the user actions that need to be performed to activate the ACC feature. There are two user actions: the driver pressing the ACC_ON button and the driver pressing the ACC_SET button. Although the enabling process is ordered, we did not consider the order in which participants listed user actions. In order for the participant's response to be deemed correct, it must include the two actions mentioned above and must not list any additional actions.

The third question asks the participant to list all of the states in which the feature can directly affect the vehicle's behaviour. The number of states and their names depends on the study group to which the participant belongs. The states in which the PI and P group's model affects the vehicle's behaviour are: Accelerate, Decelerate, Deactivating, Failing, and Unsafe Distance. The states in which the C group's model affect the vehicle's behaviour are: Accelerate, Decelerate, and ACCFailed. In order for a participant's response to be deemed correct, it must include exactly the states appropriate to the participant's group. We also accepted any responses that listed also those states' superstates because the superstates contain sub-states that affect the vehicle's behaviour.

The fourth question asks the participant to describe the behaviour of the ACC feature the first time that it detects a failure. A correct response is one that mentions that the *driver is warned* and then, *after a 10 second delay*, the ACC feature fails. This behaviour is the same in all three groups' state-machine models. We considered a participant's response to be correct as long as it listed the two behaviours described above and did not list any additional.

The fifth question asks the participants to list the initial state(s) of the ACC feature's state machine. The name of the initial state depends on the study group to which the participant belongs. In the PI and P groups' models, the Inactive state is the initial state (we also accepted Disabled as a correct response because it is the initial state of Inactive's sub-machine). In the C group's model, the initial state is called ACCMachine.

The final question asks participants to indicate the features to which the ACC feature refers and to describe the information that the ACC feature requires about those features. There are two features to which the ACC feature refers: the Anti-Lock Braking System (ABS) feature and the Electronic Stability Control (ESC) feature. These references check

Question
<p>1. List all of the environmental conditions that must hold for the feature to remain active once the feature is active and controlling the vehicle's speed.</p> <p><i>There are four environmental conditions that must hold: the vehicle's speed must be greater than 30 km/h, the brake pedal must not be pressed, there must not be a hardware failure, and the vehicle must be a safe distance from a preceding vehicle. A participants response must list all four conditions to be correct. If the participant's response does not list all four conditions or lists additional conditions, then it is incorrect.</i></p>
<p>2. List all of the user actions that the user performs as part of the process to activate the feature.</p> <p><i>There are two user actions that are performed in order to activate the feature: ACC_ON and ACC_SET. A participant's response must list these two actions to be correct. If the participant's response is missing either of these actions or lists additional actions, then it is incorrect.</i></p>
<p>3. List all of the states in which the feature can directly affect the behaviour of the vehicle.</p> <p><i>The correct answer depends on which group the participant is in. All of the states that directly affect the vehicle's behaviour must be listed for the participant's response to be correct. If a response is missing any state that directly affects the vehicle's behaviour or if the response includes additional states, then it is incorrect.</i></p>
<p>4. Describe how the feature behaves when it first detects a failure.</p> <p><i>In this question, we want the participant to give a short textual description of the feature's behaviour when it detects a failure and transitions to Failed. Specifically, we want the participant's response to state that the driver is warned that a failure has occurred and then the feature waits 10 seconds before transitioning to Failed.</i></p>
<p>5. List the name(s) of the initial state(s) of the ACC feature (when the vehicle is first powered on)?</p> <p><i>The correct answer depends on which group the participant is in. There is only one initial state in each state-machine. Only that state should be listed for the participant's response to be correct. If a response includes additional states, then it is incorrect.</i></p>
<p>6. Circle the features that the ACC feature refers to, and list all of the information that ACC obtains from these features.</p> <p><i>This question provides a list of six features: ACC, Lane Centring Control (LCC), Adaptive Headlights (AH), Road Change Alert (RCA), Electronic Stability Control (ESC), Anti-Lock Braking System (ABS), and Map Information (MAP). The features that the ACC feature refers to are ABS and ESC. The information that ACC obtains is whether ABS and ESC are Active. Both features should be listed and the information should be correct for a participant's response to be correct. If a response is missing either feature, if the response includes additional features, or if the response gives the wrong information being referenced from ABS or ESC, then it is incorrect.</i></p>

Table 5.1: The six questions that we asked participants about the provided state-machine model of the ACC feature. The criteria we used to evaluate participant's responses are given in italics.

Question	Number of participants who answered questions correctly		
	Control (C)	Pattern (P)	Pattern+Interface (PI)
List environmental cond.	0/6	2/6	0/6
List user actions	2/6	4/6	4/6
List behaviour states	3/6	5/6	5/6
Describe failure behaviour	3/6	5/6	6/6
Initial state name	5/6	5/6	6/6
Describe references	2/6	4/6	4/6
Average	2.5	4.2	4.2

Table 5.2: The total number of participants who answered each question correctly.

(1) whether the ABS feature is Active and (2) whether the ESC feature is Active. The question includes a list of six automotive features that were previously explained to the participant (Table 5.1 lists the six features), including the ABS and ESC features. The participants were to mark the ABS and ESC features and list the information that the ACC feature requires about them (in the first phase of the study, the question did not provide the participants with a list of possible features from which to choose).

Our hypothesis is that a model that employs the pattern and interface is easier to understand than a model that does not employ the pattern or interface. Questions 1, 2, 3, and 4 ask about aspects of the ACC feature’s model that are directly affected by the pattern, so we would expect that these questions are more likely to be answered correctly by the participants in the PI and P groups than by participants in the C group. Question 5 is relatively simple and we expected that it would be answered well by the participants in all study groups. Question 6 should be answered well by the PI group’s participants because they have been told how the interface can be used to reference information about other features’ current execution state.

State Machine Comprehension Results

The number of correct responses to each question for each study group is shown in Table 5.2. On average, the participants in the PI and P groups answered 4.2 questions correctly, whereas the participants in the C group answered only 2.5 questions correctly on average.

The question that asked participants to list all of the environmental conditions that should hold for the ACC feature to remain active was particularly difficult for the participants to answer correctly. Most participants listed at least two of the four environmental conditions, but very few managed to list all four conditions. It was also common for

participants to list six or seven conditions (some of which were not even present in the state-machine model). These results refute our hypothesis that this question would be answered well by the members of the PI and P groups.

The majority of participants in the PI and P groups correctly listed the user actions performed during the feature's enabling process, but only a third of the C group's participants correctly listed the actions. We believe that this is because the Inactive state extension separates user actions and environmental conditions into distinct groups, making them easier to locate in the model.

The states in which the ACC feature directly affects the vehicle's behaviour were listed correctly by a majority of the participants in the PI and P groups, but only half of the C group's participants listed the correct set of states. This is likely because the pattern places constraints on the states in which the feature can directly alter its environment, thus making it obvious where to look for that kind of behaviour. Those participants who answered incorrectly usually listed the correct set of states but also included several other states that do not directly affect the vehicle's behaviour.

The participants in the PI and P groups had no trouble correctly describing the behaviour of the ACC feature when it first detects a failure, but many of the C group's participants were unable to correctly describe that behaviour. This may be explained by the fact that the pattern makes the failure behaviour easier to locate than in the C group's model. The most common mistake that participants made was listing additional behaviours (some of which were not even present in the ACC feature's model).

The initial state of the ACC feature was correctly listed by most of the participants in all three study groups. This behaviour was fairly obvious so we are unsurprised by this result.

In the final question, two thirds of the participants in each of the PI and P groups correctly stated that the ACC feature refers to the ABS and ESC features, but only a third of the C group's participants correctly stated these features. Those participants who correctly listed the ABS and ESC features always correctly described the information that the ACC feature requires. In the first phase of the study, most of the participants who answered this question incorrectly seemed to misunderstand the question and listed several additional feature references that are not present in the state-machine model. In the second phase of the study, the question listed six possible features that the ACC feature may refer to. Unfortunately, many of the participants still seemed to misunderstand the question and tried to list all of the ways in which the ACC feature may refer to each of the six features that we listed.

We performed a one-way ANOVA test with post-hoc TukeyHSD analysis [49] to determine statistical significance of the study results. The ANOVA test evaluates statistical significance by determining if the variation between the scores of participants from different groups is large or small compared to the variation between scores of participants within the same group. The post-hoc TukeyHSD analysis decomposes the ANOVA results to examine the variation between pairs of study groups. For each pair of groups, if the variation (the p value) is less than 0.05 then the results are considered statistically significant.

The independent variable is the participant's group and the dependent variable is his or her results to the questions. The participant's results follow a normal distribution and the standard deviation of each participant group's results is within tolerance, thus the ANOVA test can be used to determine statistical significance.

Unfortunately, the results of the state-machine comprehension portion of the study are not statistically significant for any pair of groups. The p value for the P and C groups is 0.25 and the p value for the PI and C groups is 0.25. The p value for the PI and P groups is 1 (indicating that there is no noticeable difference between the groups). If the PI and P groups are combined into one group and that is compared to the C group's results using the ANOVA test then the resulting p -value is 0.053, which is still not statistically significant. Therefore, although the average number of questions answered correctly is higher for the PI and P groups, we can not prove our hypothesis that the pattern improves state-machine comprehension.

5.3.3 State-Machine Modelling

In the last section of the study, we tested the participant's ability to create a state-machine model of a feature. The participants were provided with textual requirements for a Lane Centring Control (LCC) feature and were asked to create a state-machine model of the requirements. The requirements for this feature come from the same course project as the ACC feature from the state-machine comprehension portion of the study. We asked the participants to spend no more than one hour on the state-machine modelling portion of the study.

The textual requirements for the LCC feature describe its enabling process, active behaviour, and failure requirements. The LCC feature's enabling process is a sequential, multi-stage process that first determines if the ACC feature is active then waits for the driver to press the LCC_ON button. The active behaviour of the LCC feature monitors the lane markings on either side of the vehicle and keeps the vehicle centred in its lane. The failure requirements of the LCC feature state that if a hardware failure occurs, then

Evaluation Criteria
<p>1. Does the state machine include all of the enabling conditions. <i>There are two enabling conditions that must be satisfied before the LCC feature can activate: the vehicle determines if the ACC feature is Active and the driver must press the LCC_ON button. For a participant's model to satisfy this criterion it must model both conditions. If the participant's model does not include both conditions, or includes additional conditions, then it does not satisfy this criterion.</i></p>
<p>2. Does the state machine model the correct enabling sequence. <i>The requirements describe the LCC feature as having an sequential, multi-stage enabling process. The enabling process should first determine if the ACC feature is Active, then wait for the driver to press the LCC_ON button before activating. For a participant's model to satisfy this criterion it must use an Ordered Enabling process with the correct order of enabling stages.</i></p>
<p>3. Does the state machine model the correct deactivation conditions. <i>There are five deactivation conditions that need to be considered: (1) the ACC feature deactivates, (2) the driver removes his or her hands from the steering wheel for a time greater than some threshold, (3) the vehicle's speed drops below 60 km/h, (4) the ACC feature detects a possible collision, and (5) the driver presses the LCC_OFF button. For the participant's model to satisfy this criterion it must include all five conditions. If his or her model does not include all five conditions, or includes additional conditions, then it does not satisfy this criterion.</i></p>

Table 5.3: The first three of seven requirements details that we used as evaluated criteria to assess the correctness of a participant's state-machine model. The criteria we used to evaluate participant's responses is given in italics.

the driver should be warned to take control of the vehicle; and once the driver has control, the feature ceases operation. The textual requirements list five environmental conditions and one user action that the participant should use to model the LCC feature's behaviour.

The study included a use-case description [11] to supplement the textual requirements. The use-case description provides the same information as the textual requirements but makes the ordering of actions explicit.

We evaluated correctness by checking that the model conformed to the textual requirements. Specifically, we checked whether the model captured seven representative details from the LCC feature's textual requirements (see Tables 5.3 and 5.4). We chose these seven requirements details as evaluation criteria because they cover the range of the feature's enabling process, active behaviour, and failure requirements. The first evaluated criterion is

Evaluation Criteria
<p>4. Does the state machine have the correct deactivation behaviour. <i>There are five deactivation conditions. One of the deactivation conditions (the driver pressing LCC_OFF) results in an immediate deactivation. If one of the other four conditions occur, the driver is warned that the LCC feature is deactivating before the feature deactivates. For the participant's model to satisfy this criterion it must correctly model the LCC_OFF button press resulting in an immediate deactivation and the other four conditions resulting in a warning to the driver before deactivation. If his or her model does not include all of the deactivation conditions or does not correctly model which conditions result in a delay, then it does not satisfy this criterion.</i></p>
<p>5. Does the state machine model the correct controlling and monitoring behaviour. <i>The LCC feature should actively control the vehicle to keep it centred in its host lane. If the driver turns the steering wheel past some threshold, then the LCC feature should transition to the Monitoring sub-state because the driver has assumed responsibility for changing lanes. For a participant's model to satisfy this criterion both types of behaviours need to be correctly modelled in the LCC feature's state machine. If the participant's model does not correctly model both behaviours or models additional behaviours, then it does not satisfy this criterion.</i></p>
<p>6. Does the state machine model the correct failure behaviour. <i>When the LCC feature fails, the driver should be warned that the feature is failing. For a participant's model to satisfy this criterion it needs to warn the driver about the failure before the feature fails. If the failure conditions are not correctly modelled or the driver is not warned about the failure, then the participant's model does not satisfy this criterion.</i></p>
<p>7. Does the LCC feature reference the ACC feature correctly. <i>The ACC feature must be active before the LCC feature can activate and if the ACC feature deactivates while the LCC feature is active then the LCC feature must deactivate. For a participant's model to satisfy this criterion it must correctly reference the ACC feature to determine if it is Active or Inactive. If the participant's model does not correctly reference both activation and deactivation, or does so incorrectly, then the participant's model does not satisfy this criterion.</i></p>

Table 5.4: The final four of seven requirements details that we used as evaluated criteria to assess the correctness of a participant's state-machine model. The criteria we used to evaluate participant's responses is given in italics.

coverage of all of the enabling conditions, and the second criterion is the correct sequence of enabling conditions. The third criterion is coverage of all of the deactivation conditions, and the fourth criterion covers the requirement that the driver should be warned before the LCC feature completely deactivates. The fifth evaluation criterion covers the requirement that the LCC feature should actively keep the vehicle centred in its lane and the driver should be able to temporarily override the centring behaviour in order to change lanes. The sixth criterion covers the requirement that the driver be warned when the LCC feature fails and the final criterion covers the LCC feature’s references to the ACC feature.

Note that Criterion 1 and Criterion 2 overlap. For example, if the participant’s model does not include all of the enabling conditions (Criterion 1) then their sequence of enabling conditions (Criterion 2) will also be incorrect. We did not take into account the fact that these criteria have overlapping concerns while evaluating the participant’s models, so a participant’s model may not satisfy multiple criteria due to the same omission. We found this to be rare because if a participant’s model did not satisfy one criterion it usually had multiple errors that caused it to not satisfy the overlapping criterion.

Our hypothesis is that the pattern and interface will improve a participant’s ability to model the LCC feature’s requirements: the pattern provides advice on how to organize the many behaviours of the LCC feature, and the interface provides a standard method for referring to other features in the environment.

State-Machine Modelling Results

Table 5.5 reports how accurately the models created by participants from the three study groups captured the requirements details that we established as evaluation criteria. For all criteria, a model had to capture the corresponding requirements detail exactly to be deemed correct. We did not consider any kind of partially correct criteria. Overall, our findings indicate that the state-machine models created by participants in the PI and P groups were more correct than the state-machine models created by participants in the C group.

All of the participants in the PI and P groups correctly included the LCC feature’s enabling conditions, whereas only half of the C group’s participants correctly included them (each of the incorrect models omitted some enabling condition). Likewise, all of the participants in the PI and P groups correctly modelled the LCC feature’s enabling process, whereas only half of the C group’s participants did so correctly. We believe that the reason the models created by the PI and P participants had more complete enabling processes is due to the emphasis that the pattern places on how to model a feature’s enabling process. The pattern specifies how to model different types of enabling conditions (i.e., user actions vs. environmental conditions), and it specifies how to model a sequential,

Evaluation Criteria	Number of participants who modelled criteria correctly		
	Control (C)	Pattern (P)	Pattern+Interface (PI)
Includes all enabling conditions	3/6	6/6	6/6
Correct enabling sequence	3/6	6/6	6/6
Correct deactivation conditions	1/6	2/6	5/6
Correct deactivation behaviour	3/6	5/6	6/6
Correct controlling and monitoring behaviour	3/6	4/6	5/6
Correct failure behaviour	2/6	5/6	5/6
References ACC correctly	1/6	1/6	5/6
Average	2.2	4.3	5.4

Table 5.5: The total number of participants who modelled each evaluation criterion correctly.

multi-stage enabling process (using the Ordered Enabling variant of the Inactive extension); the C group’s participants did not have any such guidance.

Five of the PI group’s participants, two of the P group’s participants, and one of the C group’s participants correctly modelled all of the LCC feature’s deactivation conditions. That the members of the PI group performed better than the other study’s participants with respect to this evaluation criterion makes sense because one of the LCC feature’s deactivation conditions is that the ACC feature is no longer active. Many of the participants in the P and C groups tried to model this condition as ACC_OFF, however that is not correct because there are multiple ways that the ACC feature can deactivate. Because the PI group’s participants had received advice on how to reference another feature’s state-machine model, they were better able to model this particular deactivation condition correctly.

The majority of the participants in the PI and P groups correctly modelled the deactivation behaviour of the LCC feature, but only half of the participants in the C group correctly modelled that behaviour. We believe that the reason the models created by participants in the PI and P groups modelled the feature’s deactivating behaviour more

accurately is due to the pattern's Deactivating state; under four (of the five) deactivation conditions, the driver should be warned before the LCC feature deactivates. The Deactivating state in the pattern explicitly describes how to model an intermediate step in a feature's deactivating process. The participants who incorrectly modelled the feature's deactivation almost always omitted the warning to the driver.

Five of the PI group's participants, four of the P group's participants, and three of the C group's participants correctly modelled the monitoring and controlling behaviour of the LCC feature. Although a higher percentage of the PI group's participants modelled this behaviour correctly, the differences between the groups' performances are not very large. The active behaviour of the LCC feature (as it is described in the study) is relatively basic and so it is possible to model the monitoring and controlling behaviour using very few states. All of the PI and P group's participants separated the monitoring behaviour from the controlling behaviour using the Monitoring and Controlling sub-states of Active. Most of the models created by the C group's participants did not separate the feature's monitoring and controlling behaviour into two states, but three of the C group's models still captured the correct behaviour.

The failure behaviour of the LCC feature was modelled correctly by five participants in each of the PI and P groups, but was modelled correctly by only two of the C group participants. Modelling the failure behaviour of the LCC feature is nearly identical to modelling its deactivating behaviour. As with the deactivation behaviour, when the LCC feature fails, it should warn the driver about the failure before it completely stops functioning. The Failing state in the pattern explicitly describes how to model an intermediate step in a feature's failing process. As with the deactivation behaviour, the participants who modelled the failing behaviour incorrectly usually omitted the warning to the driver.

The final behaviour that we check is if the LCC state machine correctly references the ACC feature. Five of the PI group's participants referenced ACC correctly, whereas only one P group participant and one C group participant did so. This is understandable because the interface explicitly states how to model one feature referring to another. All three tutorials introduced how to use the *in()* transition condition to include references to other features on a transition, but very few participants in the P and C groups attempted to use *in()*.

We performed a one-way ANOVA test with post-hoc TukeyHSD analysis to determine the statistical significance of the state-machine modelling results. For each pair of groups, if the *p* value is less than 0.05 then the results are considered statistically significant. The independent variable is the participant's group and the dependent variable is the correctness of his or her model according to the evaluation criteria. The participant's results

Confidence	Control	Pattern	Pattern+Interface
State-Machine Comprehension	64.2%	62.5%	72.5%
State-Machine Modelling	60%	55%	66%

Table 5.6: The average confidence of each participant group in his or her solution to the comprehension and modelling tasks.

follow a normal distribution and the standard deviation of each participant group’s results is within tolerance, thus the ANOVA test can be used to determine statistical significance. We found that the difference between the PI and C groups was statistically significant (p value = 0.0006) and difference between the PI and P groups was statistically significant (p value = 0.03). The difference between the PI and P groups was not statistically significant (p value = 0.17). These results seem to confirm our hypothesis that the pattern improves the ability of participants to create correct state machines, although because the difference between the PI and P groups is not statistically significant we can not claim that the interface alone improves the correctness of state machines.

5.3.4 Participant Confidence

We expected that one of the tertiary benefits of using the pattern and interface would be that the modeller would have greater confidence in his or her understanding of state-machine models and greater confidence in the quality and completeness of his or her own models. To measure confidence, the study asked each participant to record his or her confidence in their solutions to each question. Each question in the state-machine comprehension portion of the study and the one question in the state-machine modelling portion of the study listed five confidence values: (1) Guessed, (2) At least 50% correct, (3) 50% - 75% correct, (4) 75% - 90% correct, and (5) 90% - 100% correct. For each question in the state-machine comprehension portion of the study and for the model that the participant created in the state-machine modelling portion of the study, the participant was asked to select the confidence value that best matched the participant’s level of confidence in his or her answer.

The average confidence level of each participant group in the state-machine comprehension and state-machine modelling tasks is shown in Table 5.6. The averages are calculated for each group by using the higher value for the interval that was selected (e.g., if the participant chose 50% to 75% confidence then the value that we used to calculate the average was 75%). As can be seen, all of the groups’ confidence averages fall within the range of being 50% - 75% correct. Participants from the PI group tend to have slightly

higher confidence in their solutions, but the difference is not significant enough for us to be able to draw any conclusions. Therefore, we can not claim that the use of the pattern leads reviewers to have greater confidence in their understanding of state-machine models or gives modellers greater confidence in the quality and completeness of their own models.

We noticed a correlation between a participant's confidence in his or her solutions and the participant's level of comfort with state-machine modelling (as reported in question 3 of the study's background section). The Pearson correlation is 0.61. However, we found no correlation between a participant's confidence and the correctness of his or her solutions.

5.3.5 Timing Results

We also expected that one of the tertiary benefits of using the pattern would be a reduction in the amount of time needed to perform the state-machine comprehension and modelling tasks. Therefore, as part of the study, we asked the participants to record the amount of time that he or she spent on the state-machine comprehension and state-machine modelling tasks. In the state-machine comprehension portion of the study, the amount of time spent includes the time spent reading the questions and the time spent reviewing the state-machine model. In the state-machine modelling task, the amount of time spent does not include the time to perform an initial reading of the requirements of the LCC feature; it only measures the time spent creating the state-machine model and the time spent referring back to the (already read) textual requirements.

We have summarized the timing results of the state-machine comprehension task in Figure 5.8. The median time spent on the state-machine comprehension task by each group is represented by the vertical bar. The grey box's bounds represent the first and third quartiles of the timing values, and the error bar's model the shortest and longest time spent on the task. On average, the participants in the PI and P groups spent slightly less time answering the model-comprehension questions than the C group's participants spent. However, the timing variability of the C group was very large with one participant taking only 10 minutes to answer all of the questions. Because of the large timing variability of each group and the similar averages we do not feel that we can claim that the pattern reduces the amount of time needed to understand a state-machine model.

The timing results of the state-machine modelling task are summarized in Figure 5.9. Strangely, the results indicate that, on average, the C group's participants took nearly 15 minutes less than the participants of the PI or P groups took to create a state-machine model of the LCC feature's requirements. Again, the variability in the timing results for the modelling task are very large, so we can not draw any definitive conclusions. Still, the lower

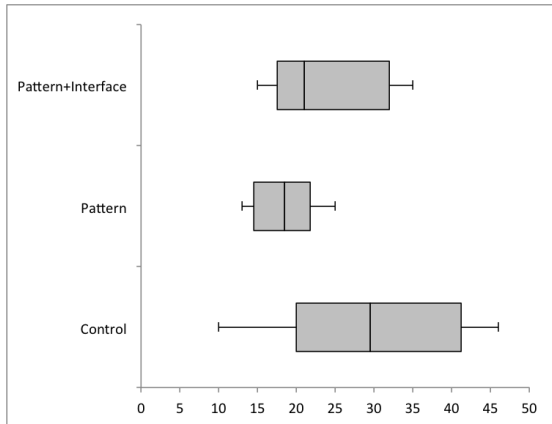


Figure 5.8: The average amount of time that the participants of each group spent on the state-machine comprehension task.

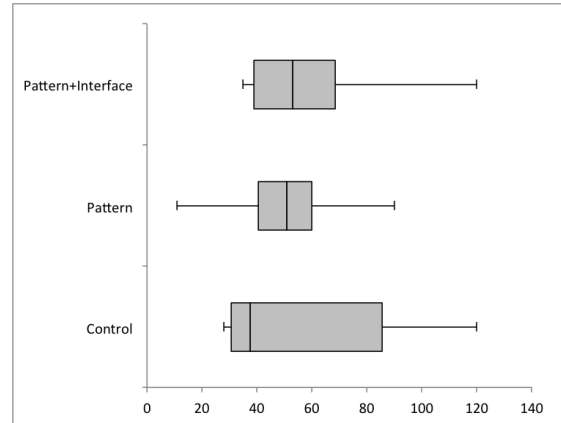


Figure 5.9: The average amount of time that the participants of each group spent on the state-machine modelling task.

average of the C group’s participants was unexpected. After examining the solutions, we have two possible explanations for why this occurred: (1) two of the C group’s participants spent very little time crafting their solutions and, due to the small number of participants per group, their timing results significantly pulled down the group’s average, and (2) the C group’s models in general tended to model less behaviour than the PI and P group’s models, which could explain why the C group’s participants spent less time on the modelling task.

Despite the 60 minute time limit for the state-machine modelling task, many of the participants seemed to spend as much time as they needed to complete the modelling task. We asked some of the participants who spent more than 60 minutes creating their models why they took so long, and they said that a significant amount of time was spent making their model visually appealing.

5.4 Threats to Validity

We have identified several threats to the validity of the user study. We do not feel that any of these are severe enough to invalidate the study results but they are important to list:

- The study had only 18 participants which makes the results hard to generalize. Despite our advertising campaign, including email directed at students who had the appropriate background, we did not receive many responses from the undergraduate community. To make strong claims about the pattern’s effect on the readability and writability of state-machine models, we need to perform the study on more participants. In the future, it may be better to make a shorter study that does not explore as many aspects of the pattern and interface but can be completed in less time.
- The study was created and the participants’ solutions were evaluated by people familiar with the pattern and interface. Thus the study could have been unintentionally favourable to the participants in the PI and P groups. This threat should be minimized because the study features’ requirements come from an external source and the ACC model for the C group was the best model created by that pool. This means that the C group participants were working with a well-designed state-machine model for the state-machine comprehension portion of the study.
- We determined the evaluation criteria for assessing the correctness of the participants’ models of the LCC feature hence we could have chosen evaluation criteria that favoured the models created by the PI and P groups. This was mitigated by the fact that we determined the correctness criteria before the participants’ models were examined.
- The ACC and LCC features exist in production vehicles and have fairly well-known behaviours. Therefore, the participants may have had pre-conceived notions about a feature’s behaviour that could have affected his or her solution to the study tasks. In fact, one of the study participants stated that they had previous experience modelling the requirements for an ACC or LCC feature. This could have resulted in some participants having experience that resulted in better performance in the comprehension and modelling portions independent of the use of the pattern and interface. However, this threat is likely mitigated because very few participants stated having any knowledge of automotive features.

5.5 User Study Summary

We have shown that use of the pattern improves the writability of state-machine models. We tried to show that use of the pattern improves the readability of the resulting state-machine models, however the results of the state-machine comprehension portion of the

study are not statistically significant. We also tried to show that use of the pattern improves the confidence of reviewers and specifiers of state-machine models but our results were inconclusive. Likewise, we were unable to determine if the pattern reduces the amount of time needed to review or specify state-machine models. Both of these aspects of the pattern require further study before we can draw any definitive conclusions. In the introduction of this chapter, we introduced three questions that we hope to answer with the user study and we include our responses below.

1. Question: Does the pattern and interface aid a requirements reviewer in interpreting state-machine models?
 - The study results seem to indicate that the pattern slightly improves the readability of state-machine models. Unfortunately, the results were not statistically significant.
2. Question: Does the pattern and interface aid the modeller in writing correct and readable state-machine models?
 - The study results indicate that the pattern and interface improve the ability of a modeller to create correct state-machine models of behaviour.
3. Question: Does the pattern and interface improve the confidence of the requirements reviewer and specifier?
 - The study results are inconclusive. There is no significant difference in the reported confidences of the participants who used the pattern versus the participants in the control group. As well, use of the pattern did not lead to a significant reduction in the time needed to review or specify the state-machine models of any one group. We believe that the pattern improves the requirements engineer's experience because it improves the readability and writability of state-machine models, but this point can not be proved.

A final conclusion from the user study is that the pattern can be taught to people rather quickly. We were able to teach the pattern to participants in one hour. This is a fairly short amount of time and may improve the chances that companies will adopt the pattern to structure their requirements.

Chapter 6

Conclusion

This thesis presents the Mode-Based Behaviour Pattern for structuring a state-machine model of a feature's behaviour. The pattern provides several extensions for modelling a feature's enabling process and its active behaviour. We performed a case study on 21 automotive features to assess the applicability of the pattern and found that the pattern is applicable to all 21 features. The thesis also presents a generic interface to features that limits the information that a feature reveals to other features to be just the feature's current mode of operation (i.e., Inactive, Active, or Failed). The case study examines the applicability of the interface and found that the vast majority of inter-feature references (50 out of 58 references) are simply queries of public-interface data.

We performed a user study to evaluate the usability benefits of the pattern. The study was performed on 18 participants and we found that the pattern improves the ability of participants to create correct state-machine models that apply the pattern. We asked each participant to record his or her confidence in the correctness of their solutions and record the amount of time they spent answering the questions, but we were unable to draw any conclusions from that data.

Overall, we believe that the pattern presents a useful pre-defined structure for a state-machine model of a feature's behavioural requirements.

As future work, the first step would be to evaluate the pattern on features from other domains or on features from other automotive companies. We also plan to perform further user studies to assess more definitively the usability benefits of the pattern. Finally, we believe it would be beneficial to provide a graphical state-machine modelling tool that supports the pattern's constructs.

References

- [1] J. Aldrich. Open modules: modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 144–168, 2005.
- [2] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model superimposition in software product lines. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT'09)*, pages 4–19, 2009.
- [3] S. Apel and C. Kastner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.
- [4] G. Arango and P. Freeman. Modeling knowledge for software development. In *Proceedings of the 3rd International Workshop on Software Specification and Design (WSSD'85)*, pages 63–66, 1985.
- [5] S. Arora, P. Sampath, and S. Ramesh. Resolving uncertainty in automotive feature interactions. In *Proceedings of IEEE International Requirements Engineering Conference (RE)*, pages 21–30, 2012.
- [6] L. Bass and B. E. John. Linking usability to software architecture patterns through general scenarios. *Journal of Systems and Software*, 66(3):187–197, 2003.
- [7] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
- [8] L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 266–277, 1997.

- [9] B. H. C. Cheng, S. Konrad, L. A. Campbell, and R. Wassermann. Using security patterns to model and analyze security. In *Proceedings of the Workshop on Requirements for High Assurance Systems, at the IEEE International Requirements Engineering Conference*, pages 13–22, 2003.
- [10] L. Chung, B. Paech, L. Zhao, L. Liu, and S. Supakkul. RePa requirements pattern template. Technical report, 2013.
- [11] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.
- [12] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
- [13] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering ESEC/FSE'01*, pages 109–120, 2001.
- [14] B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [15] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering ICSE'99*, pages 411–420, 1999.
- [16] M. Fowler. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [17] X. Franch, C. Palomares, C. Quer, S. Renault, and F. D. Lazzer. A metamodel for software requirement patterns. In *Proceedings of Requirements Engineering: Foundation for Software Quality (REFSQ'10)*, pages 85–90. Springer, LNCS 6182, 2010.
- [18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [19] A. Granlund, D. Lafrenire, and D. A. Carr. A pattern-supported approach to the user interface design process. In *Proceedings of HCI International 2001, 9th International Conference on Human-Computer Interaction*, New Orleans, 2001.

- [20] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [21] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
- [22] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [23] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, Oct. 1998.
- [24] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.
- [25] C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference (SPLC), Volume 2*, pages 5:1–5:8, 2011.
- [26] R. K. Keller, J. Tessier, and G. von Bochmann. A pattern system for network management interfaces. *Communications of the ACM*, 41(9):86–93, Sept. 1998.
- [27] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE'05)*, pages 49–58, 2005.
- [28] S. Konrad, L. A. Campbell, and B. H. C. Cheng. A requirements patterns-driven approach to specify systems and check properties. In *Model Checking Software, LNCS 2648*, pages 18–33. Springer Verlag, 2003.
- [29] S. Konrad and B. H. C. Cheng. Requirements patterns for embedded systems. In *Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE'02)*, pages 127–136, 2002.
- [30] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 372–381, 2005.

- [31] R. C. Laney, T. T. Tun, M. Jackson, and B. Nuseibeh. Composing features by managing inconsistent requirements. In *International Conference on Feature Interactions (ICFI)*, pages 129–144, 2007.
- [32] H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for modular feature verification. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 195–204, 2002.
- [33] Y. Li, C. Pelties, M. Kaser, and N. Nararan. Requirements patterns for seismology software applications. In *Proceedings of the IEEE International Workshop on Requirements Patterns (RePa)*, pages 12–16, 2012.
- [34] R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 169–194, 2005.
- [35] Object Management Group. UML Specification: Superstructure, version 2.2, 2009.
- [36] OMG. *OMG Systems Modeling Language*, 1.2 edition, 2010.
- [37] K. Ostermann, P. G. Giarrusso, C. Kästner, and T. Rendel. Revisiting information hiding: reflections on classical and nonclassical modularity. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11)*, pages 155–178, 2011.
- [38] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [39] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Berlin/Heidelberg, 1997. Springer-Verlag.
- [40] S. Prochnow and R. V. Hanxleden. Statechart development beyond wysiwyg. In *In Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS07)*, 2007.
- [41] H. Reubenstein and R. Waters. The requirements apprentice: automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, March 1991.

- [42] J. L. Rodgers and W. A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42:59–66, 1988.
- [43] R. Salay, M. Chechik, and J. Horkoff. Managing requirements uncertainty with partial models. In *Proceedings of the 2012 IEEE 20th International Requirements Engineering Conference (RE'12)*, pages 1–10, 2012.
- [44] P. Shaker, J. M. Atlee, and S. Wang. A feature-oriented requirements modelling language. In *Proceedings of the International Requirements Engineering Conference (RE'12)*, pages 151–160, 2012.
- [45] A. Sutcliffe and N. Maiden. The domain theory for requirements engineering. *IEEE Transactions on Software Engineering*, 24(3):174–196, March 1998.
- [46] D. Svetinovic, D. M. Berry, N. A. Day, and M. W. Godfrey. Unified use case state-charts: Case studies. *Requirements Engineering Journal*, 12(4):245–264, 2007.
- [47] L. Tahvildari. Assessing the impact of using design-pattern-based systems. Master's thesis, David R. Cheriton School of Computer Science, 1999.
- [48] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, 1999.
- [49] T. C. Urdan. *Statistics in plain English; 3rd ed.* Taylor & Francis, Hoboken, 2010.
- [50] S. Withall. *Software Requirement Patterns*. Microsoft Press, Redmond, WA, USA, 2007.
- [51] M. K. Zimmerman, K. Lundqvist, and N. Leveson. Investigating the readability of state-based formal requirements specification languages. In *Proceedings of the 24th International Conference on Software Engineering, ICSE '02*, pages 33–43, New York, NY, USA, 2002. ACM.

Appendix A

Catalogue of Case Study Models

In this section we include a complete catalogue of state-machine models created during the case study. To avoid revealing proprietary information we have abstracted away several details of the state machines; several transition labels have been omitted and replaced with the number of conditions that are checked by the transition. We have also omitted many feature-specific details within the Controlling state and removed many of the additional concurrent regions within the Active state.

Each feature belongs to one of three sub-systems:

1. **Electronic Braking Features** - These are features that assist the driver while braking the vehicle. For example, Brake Assist, which increases braking force during an emergency stop.
2. **Freeway Limited Ability Autonomous Driving Features** - These are features that perform some limited automatic driving while the vehicle is travelling on a freeway. These features tend to expect the vehicle to be travelling above a certain speed, and they expect the road to have certain characteristics. This set of features includes things like Adaptive Cruise Control, which maintains the vehicle at a certain speed.
3. **Heating, Ventilating, and Air Conditioning Features** - This set of features are used to control the environment inside the vehicle's cabin. For example, the Air Quality System feature controls the pollution levels within the vehicle.

A.1 Electronic Braking Features

A.1.1 Automatic Braking (AB)

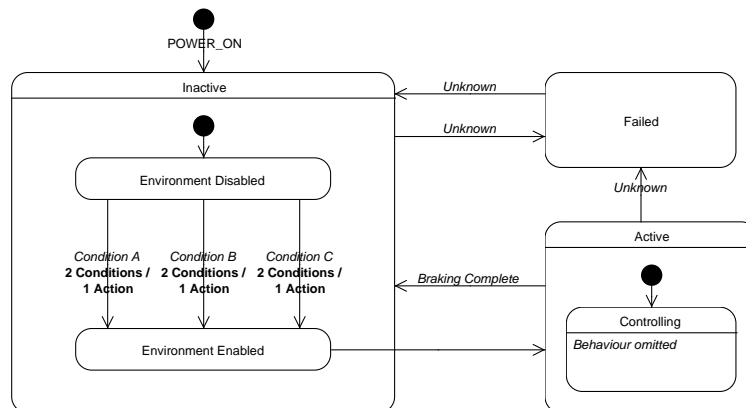


Figure A.1: The Automatic Braking (AB) feature.

AB does not interact with the vehicle’s driver but is sent requests by other features to apply the vehicle’s brakes. It seems uncommon for a feature to perform this kind of a “subservient” role and of the features we have examined, Adaptive Cruise Control is the only feature that uses AB. In fact, the one behaviour modification that we discussed in Chapter 4 was the case where the Adaptive Cruise Control feature applies the Automatic Braking feature. Two other features use AB, but we do not have access to their requirements.

AB is modelled using the Ordered Enabling extension of Inactive. The three transitions from Environment Disabled to Environment Enabled (labelled *Condition A*, *Condition B*, and *Condition C*) correspond to the three different features that use AB. AB monitors for the signals that correspond to each feature that utilizes it. The action on each transition sets the braking force depending on the feature that requires braking. The requirements for AB do not discuss the case where multiple features may try to use it at the same time.

AB is modelled using the Controlling sub-state of Active. While in Controlling, AB listens for a signal from the calling feature to determine when it should stop braking and transition back to Inactive. The failure requirements of AB were not discussed.

A.1.2 Anti-lock Braking System (ABS)

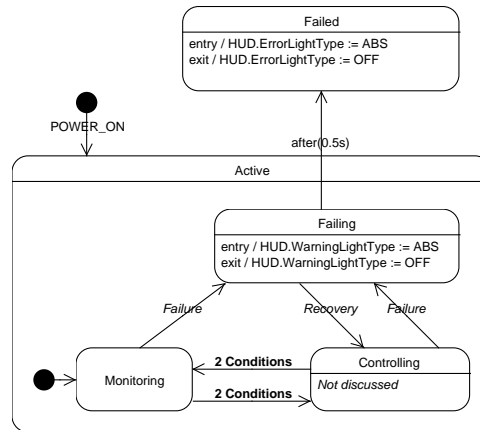


Figure A.2: The Anti-lock Braking (ABS) feature.

ABS reduces the stopping distance of the vehicle by preventing the wheels from losing traction with the road (our requirements documents do not describe how exactly this occurs). One interesting aspect of ABS is that it can not be turned off by the user, hence there is no Inactive state because there are no enabling or disabling requirements. This also makes ABS one of the few features that initializes in the Active state.

ABS is modelled using the Monitoring, Controlling, and Failing sub-states of Active. ABS transitions from the Monitoring to the Controlling sub-state when it determines that the vehicle's tires are losing traction. The requirements documents did not provide enough information to model any behaviour within the Controlling sub-state. ABS states that it can fail, but the requirements do not discuss the exact failure conditions.

A.1.3 Active Trailer Stability Assist (ATSA)

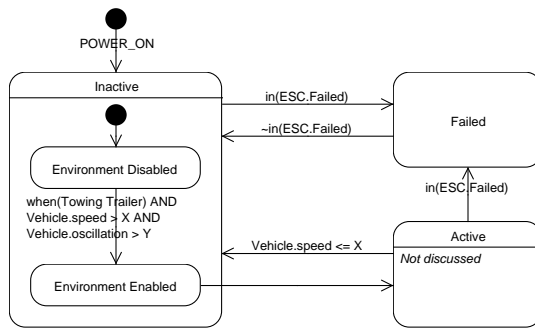


Figure A.3: The Active Trailer Stability Assist (ATSA) feature.

Trailers that are being towed by a vehicle travelling at a high speed may oscillate from side to side. The oscillation can affect the vehicle and cause the driver to lose control. ATSA assists the driver maintain control of the vehicle when it is towing a trailer by minimizing the oscillation when it is detected. The requirements documents indicate that ATSA uses the Electronic Stability Control (ESC) feature to slow down and reduce the oscillation of the vehicle (more detail is not provided in the documents to which we have access). We have not omitted the transition conditions on ATSA because there are few of them and they are relatively simple.

ATSA is modelled using the Ordered Enabling extension of Inactive, but only checks environmental conditions during the enabling process. ATSA activates when the vehicle is towing a trailer, and the vehicle speed and oscillation are greater than some calibrated values. We did not have enough information about ATSA to model its active behaviour.

A.1.4 Brake Assist (BA)

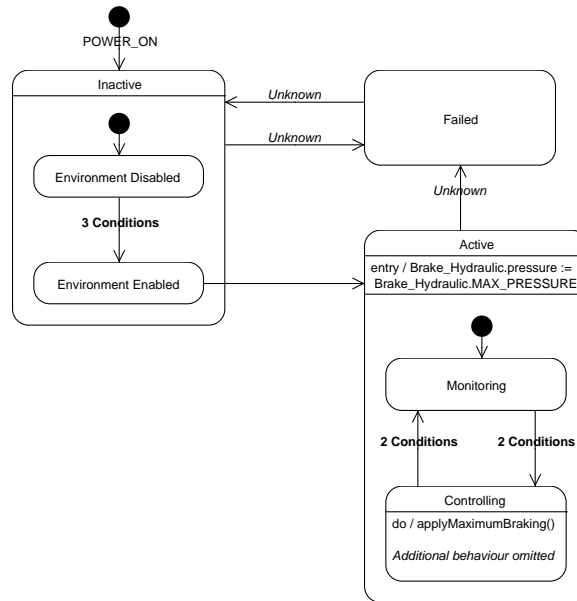


Figure A.4: The Brake Assist (BA) feature.

BA reduces the vehicle's stopping distance by applying maximum braking force when a panic stop is initiated by the user. A panic stop is defined as pressing down quickly on the brake pedal with force greater than some threshold.

BA is modelled using the Ordered Enabling extension of Inactive and the Monitoring and Controlling sub-states of Active. The enabling process checks only environmental conditions to determine if ATSA can activate. The entry action of the Active state commands the brake hydraulics to maintain maximum pressure so that emergency braking can be performed. BA waits in the Monitoring sub-state until emergency braking is required, at which point the brakes are applied with maximum pressure. The failure requirements of BA are not discussed.

A.1.5 Brake Cleaning (BC)

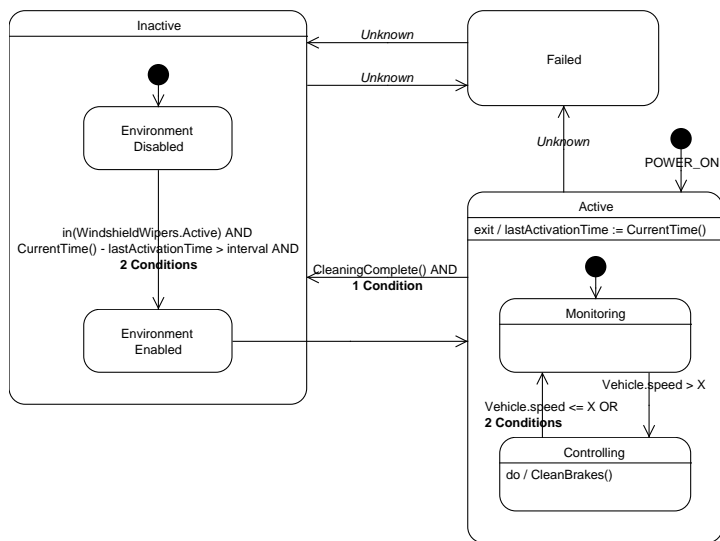


Figure A.5: The Brake Cleaning (BC) feature.

BC removes buildup on the vehicle's brakes by scraping the brake pads along the brake discs while the vehicle is moving at a high speed. BC is activated when the driver activates the windshield wipers. There is a minimum amount of time that must elapse before subsequent activations (the requirements do not explain why this is needed).

BC is modelled using the Ordered Enabling extension of Inactive. The environmental conditions ensure that the windshield wipers are active and that the minimum activation time interval has elapsed before BC can activate. BC is modelled using the Monitoring and Controlling sub-states of Active. The active sub-machine initializes in the Monitoring sub-state and transitions to the Controlling sub-state when the vehicle speed is above some minimum value. If the vehicle speed drops below the minimum value and the brakes are not clean then BC will wait in the Monitoring sub-state until the vehicle speed becomes greater than the minimum value again. When the brake cleaning is complete, the feature deactivates. When the state machine leaves the Active state the activation interval resets.

The requirements for BC state that it should clean the brakes as soon as the vehicle is turned on and vehicle speed becomes greater than the minimum value. We have modelled this by making Active the initial state. BC does not discuss any failure requirements.

A.1.6 Electric Park Brake (EPB)

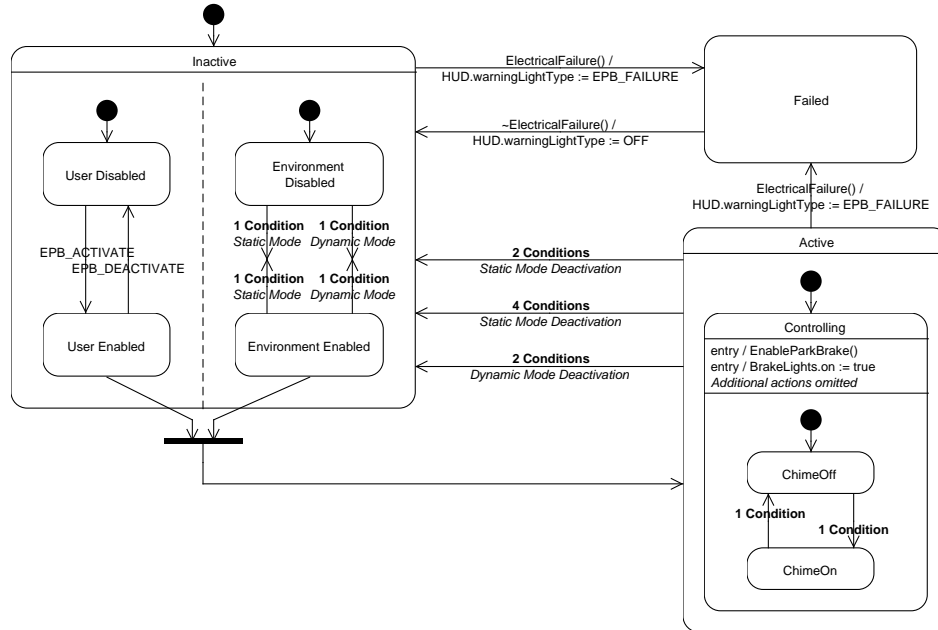


Figure A.6: The Electric Park Brake (EPB) feature.

The EPB feature controls the vehicle’s parking brake. The EPB is controlled using buttons on the vehicle dashboard. The EPB has a *static* operating mode and a *dynamic* operating mode that affect how the EPB responds to changing environmental conditions while active. In static mode, the park brake is always on until it is deactivated. In dynamic mode, the park brake considers the vehicle’s speed and modulates the amount of braking pressure appropriately. The EPB can be activated or deactivated when the vehicle is turned off. If the vehicle is turned on and the EPB is activated then it applies the parking brake. If the vehicle is not turned on and the EPB is activated then the vehicle briefly enters a low-power mode in order to enable the park brake and display an alert on the dashboard.

The EPB is modelled using the Unordered Enabling extension of Inactive and the Controlling sub-state of Active. When the state machine enters the Controlling sub-state it enables the parking brake and turns on the brake lights. The sub-states within Controlling implement behaviour that warns the driver if he or she moves the vehicle while the EPB is operating in static mode. The EPB does not discuss any failure requirements.

A.1.7 Enhanced Traction System (ETS)

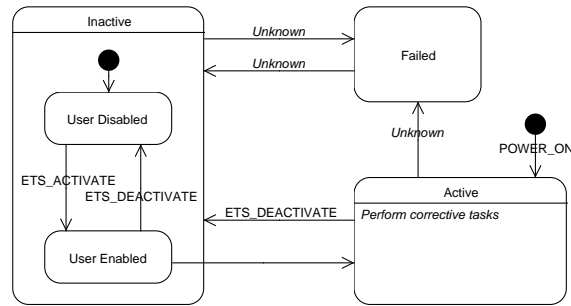


Figure A.7: The Enhanced Traction System (ETS) feature.

The ETS feature assists the driver maintain control of the vehicle when it starts to lose traction with the road surface. The ETS maintains traction by limiting engine torque but how exactly it does this is not described in the feature requirements. By default, the ETS is active when the vehicle is turned on.

The ETS feature is modelled using the Ordered Enabling extension of Inactive and only checks that the driver presses the *ETS_Activate* button before activating. The requirements for the ETS did not provide enough detail to model any active behaviour. The ETS initializes in the Active sub-state and can be disabled by the driver after the vehicle has been turned on. The ETS requirements do not discuss any failure conditions.

A.1.8 Hill Hold (HH)

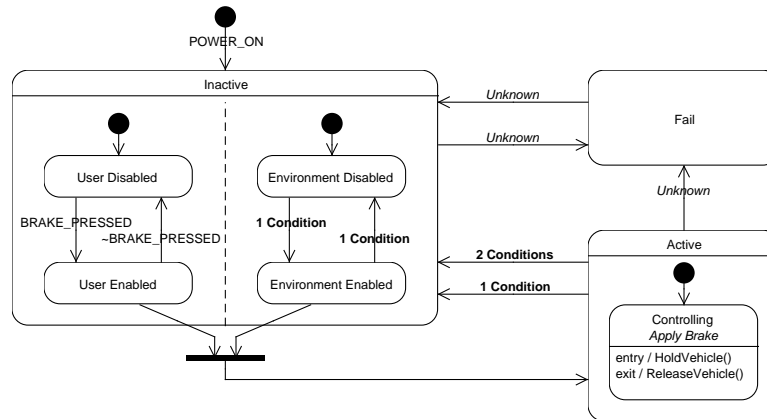


Figure A.8: The Hill Hold (HH) feature.

The HH feature delays the rate at which the vehicle rolls backwards down an incline if the brake pedal is released. To determine if HH should be used, the incline of the hill is checked using vehicle sensors. HH can not be permanently deactivated by the vehicle's driver.

HH is modelled using the Unordered Enabling extension of Inactive and the Controlling sub-state of Active. It activates when the brake pedal has been released and the vehicle is on an incline. When entering the Controlling sub-state, HH slows the vehicle from rolling downhill. HH deactivates when the driver presses on the brake or the vehicle starts to move forward. The requirements for HH do not discuss failures.

A.1.9 Traction Control System with Electronic Stability Control (TCS_ESC)

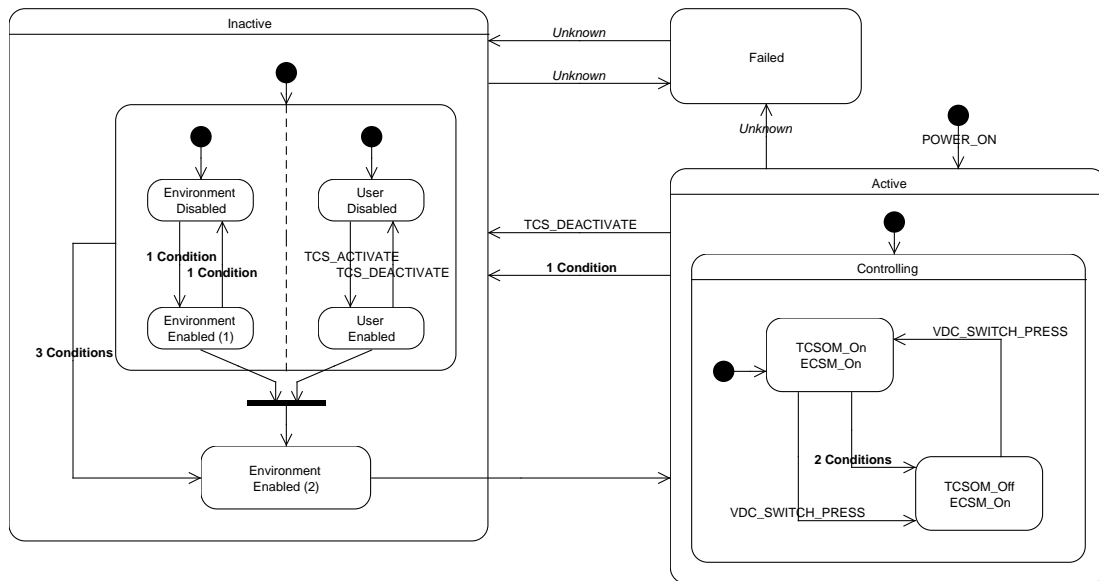


Figure A.9: The Traction Control System with Electronic Stability Control (TCS_ESC) feature.

The TCS_ESC feature assists the driver maintain control of the vehicle when it starts to lose traction with the road surface by reducing engine torque and applying the vehicle’s brakes. This is a refinement of the ETS feature that was previously presented in this section. Once active, the TCS_ESC feature has two operating modes that provide varying levels of control over the vehicle. The operating mode is toggled by pressing a button on the dashboard (modelled as the *VDC_SWITCH_PRESS* user action). The details of what each operating mode provides are not discussed in the requirements to which we have access.

The TCS_ESC is the only feature that is modelled using the Hybrid Enabling extension of Inactive. In the basic case, the enabling process for the TCS_ESC waits for the vehicle driver to explicitly enable the feature and waits for an environmental condition to be true. However, the TCS_ESC can be activated in emergency situations without any driver input – we use the hybrid enabling extension to model this behaviour.

The active behaviour for TCS_ESC is modelled using the Controlling sub-state of Active. Within the Controlling sub-state, the behaviour is separated into 2 states that depend on the mode that the driver has chosen. We have not attempted to model any behaviour within those two states.

The failure requirements for the TCS_ESC are not discussed.

A.1.10 Competitive Traction Control System with Electronic Stability Control (Competitive_TCS_ESC)

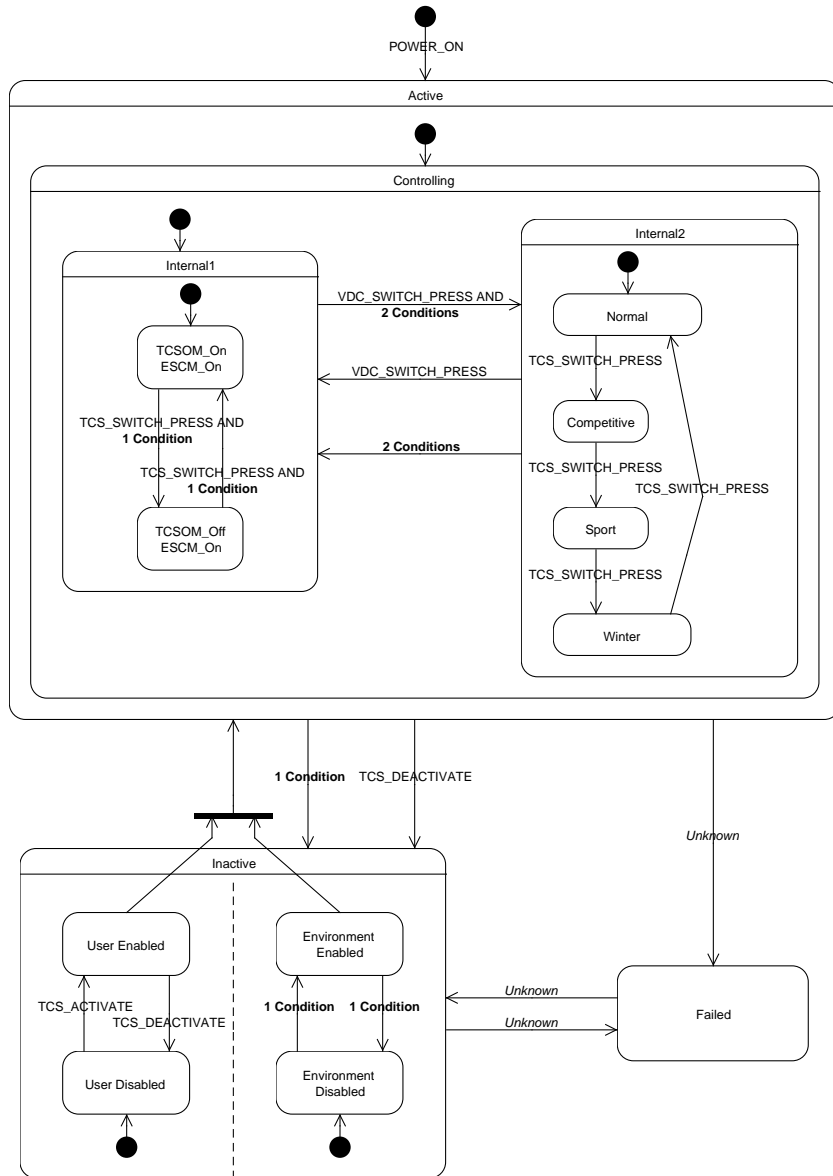


Figure A.10: The Competitive Traction Control System with Electronic Stability Control (Competitive_TCS_ESC) feature.

The `Competitive_TCS_ESC` feature is the most advanced refinement of the ETS. It offers several operating modes that are useful for different road conditions and driver requirements. The requirements documents do not provide enough information to model any mode-specific behaviour. Interestingly, `Competitive_TCS_ESC`'s active behaviour is a superset of `TCS_ESC`'s but `Competitive_TCS_ESC` does not offer an emergency override.

`Competitive_TCS_ESC` is modelled using the Unordered Enabling extension of Inactive and the Controlling sub-state of Active. There are two switches that the driver uses to switch between the various behaviour modes for `Competitive_TCS_ESC`. The failure requirements of `Competitive_TCS_ESC` are not discussed.

A.1.11 Manual Park Brake (MPB)

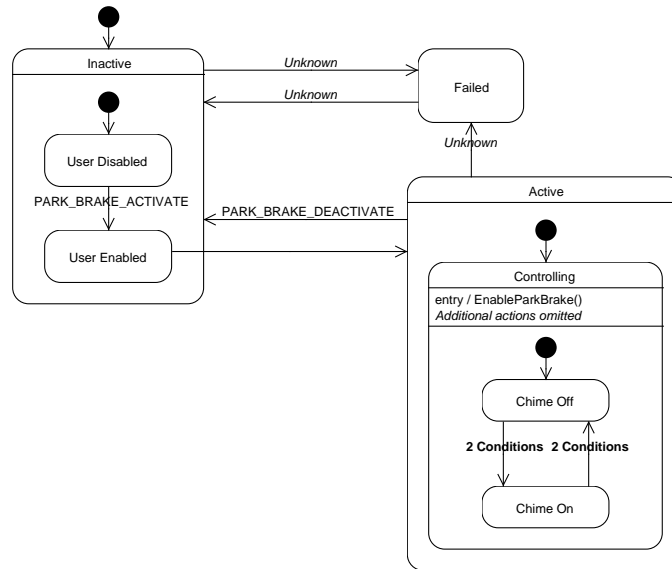


Figure A.11: The Manual Park Brake (MPB) feature.

The MPB feature implements behaviour for the vehicle’s parking brake. Unlike the EPB feature presented earlier in this section, the MPB is activated by pushing or pulling on a lever. If the vehicle is not turned on and the MPB is activated then the vehicle briefly enters a low-power mode in order to enable the park brake and display an alert on the dashboard. The driver is alerted if they try to move the vehicle the MPB is active.

The MPB is modelled using the Ordered Enabling extension of Inactive and the Controlling sub-state of Active. The MPB requirements do not discuss failures.

A.2 Freeway Limited Ability Autonomous Driving Features

The Freeway Limited Ability Autonomous Driving (FLAAD) features tend to have more complex behaviour and greater interaction with the driver than the Electronic Braking features. This is partially because of the completeness of the FLAAD feature requirements that we have been provided. The requirements for the six FLAAD features in this sub-section are contained in 3 separate requirements documents, totaling over 500 pages. Whereas every one of the Electronic Braking features is described in a single document that has only 178 pages. However, the FLAAD features do tend to implement more varied behaviour than the Electronic Braking features.

A.2.1 Forward Collision Alert (FCA)

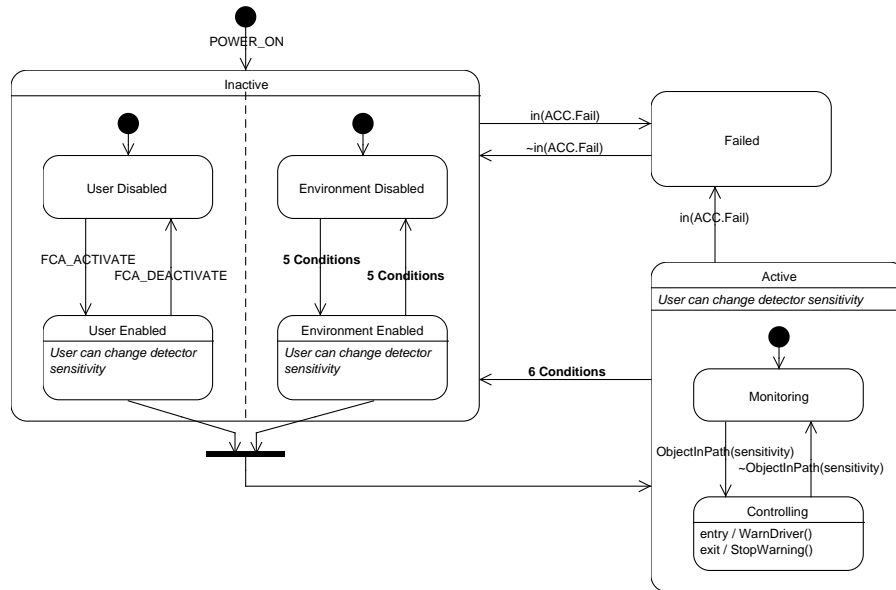


Figure A.12: The Forward Collision Alert (FCA) feature.

FCA detects if there is an object in the path of the vehicle and uses a visual, audible, and/or haptic alert to inform the driver. For example, if a preceding vehicle severely decreases their speed then FCA will detect that and inform the driver to slow down the vehicle. The driver can adjust the sensitivity of FCA using buttons on the vehicle's dashboard.

FCA is modelled using the Unordered Enabling extension of Inactive and the Monitoring and Controlling sub-states of Active. When FCA is at least partially enabled, the driver can adjust the detection sensitivity. When FCA is active the state machine waits in Monitoring until a preceding object is detected approaching too quickly, at which point the machine transitions to Controlling and alerts the driver. When the object is no longer in the vehicle's path, the state machine transitions back to Monitoring. FCA is dependent on ACC and will fail when ACC fails. Other failure requirements are not discussed.

A.2.2 Lane Centring Control (LCC)

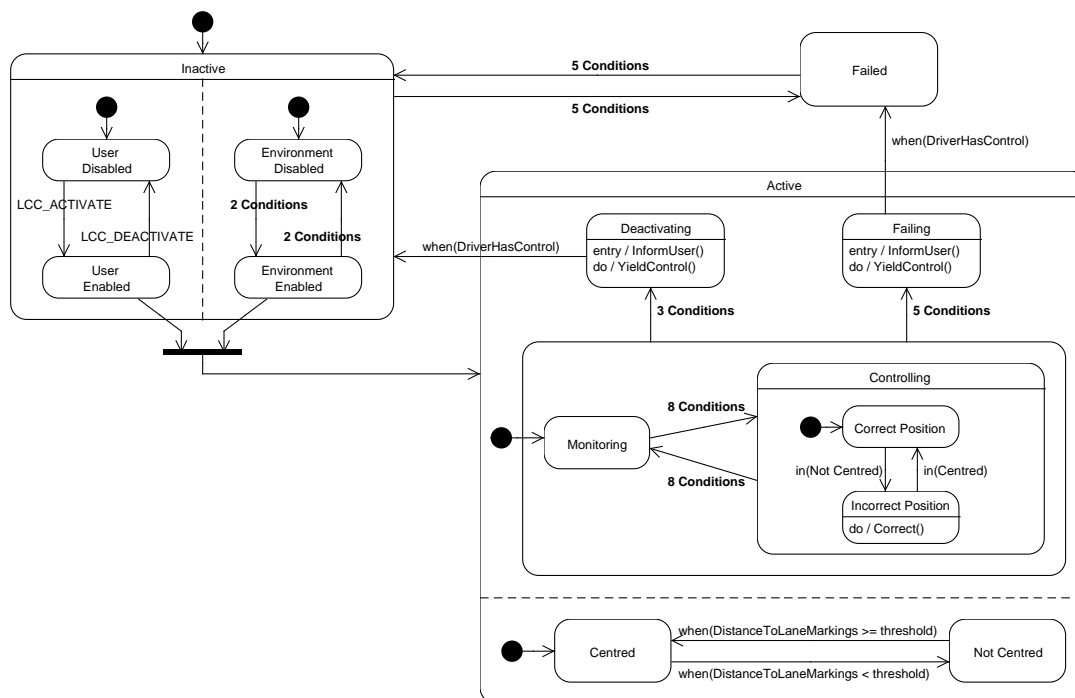


Figure A.13: The Lane Centring Control (LCC) feature.

LCC uses the lane markings on either side of the vehicle to keep it centred in its current lane. The driver can adjust the offset from the centre of the lane using buttons located on the dashboard. If the lane markings are not visible then LCC can not activate (and if active, will deactivate). Whenever LCC deactivates or a failure occurs, it informs the driver to take control of the vehicle and maintains control as best as it can until the driver has control.

LCC is modelled using the Unordered Enabling extension of Inactive, and all 4 sub-states of Active. We have modelled an additional concurrent region within Active that determines if the vehicle is centred in the lane (thus simplifying the transitions within Controlling). Note that the lane centring should take the offset into account, but we have omitted it in the model to simplify the transition label. Detailed failure requirements have not been included with the requirements we have been given for LCC.

A.2.3 Lane Keep Assist (LKA)

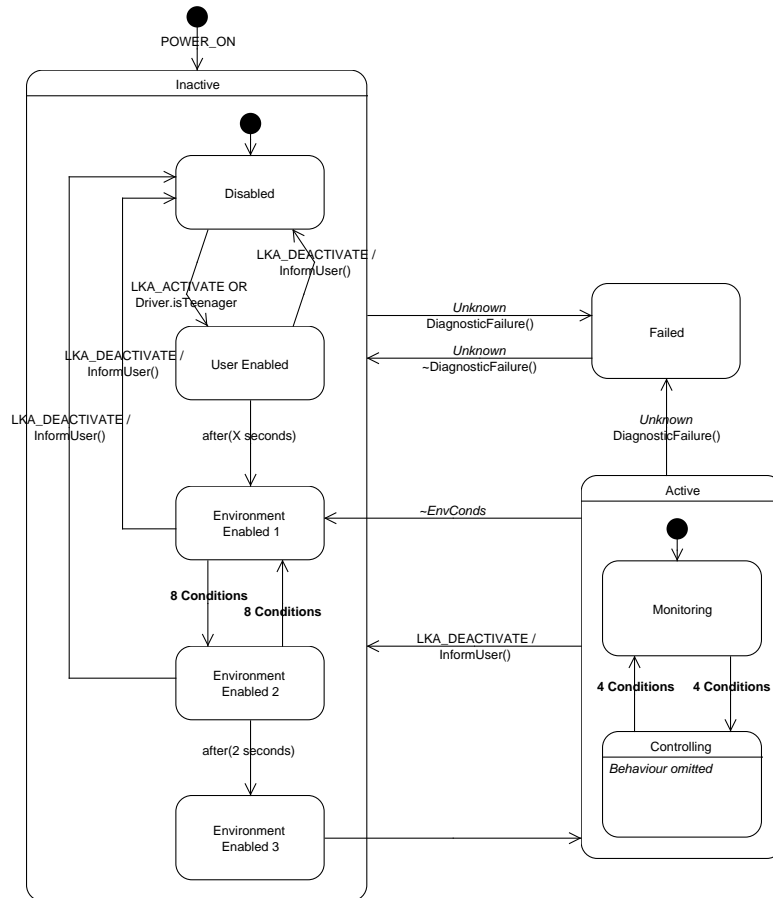


Figure A.14: The Lane Keep Assist (LKA) feature.

LKA ensures the vehicle does not accidentally leave its current lane. This behaviour is similar to LCC but instead of keeping the vehicle in the centre of the lane, LKA ensures that the vehicle does not drift out of its current lane. LKA allows the driver to change lanes when he or she has turned the steering wheel past a certain threshold.

LKA is modelled using the Ordered Enabling extension of Inactive. Of all the features we have examined, LKA has the greatest number of stages in its enabling process. There are multiple Environmental Enabling steps in succession because the requirements state

that certain conditions should hold for some period of time before the feature can activate. LKA's active behaviour is modelled using the Monitoring and Controlling states. The details of the controlling behaviour for LKA has been omitted. The failure conditions for LKA are not described.

A.2.4 Lane Change Control (LXC)

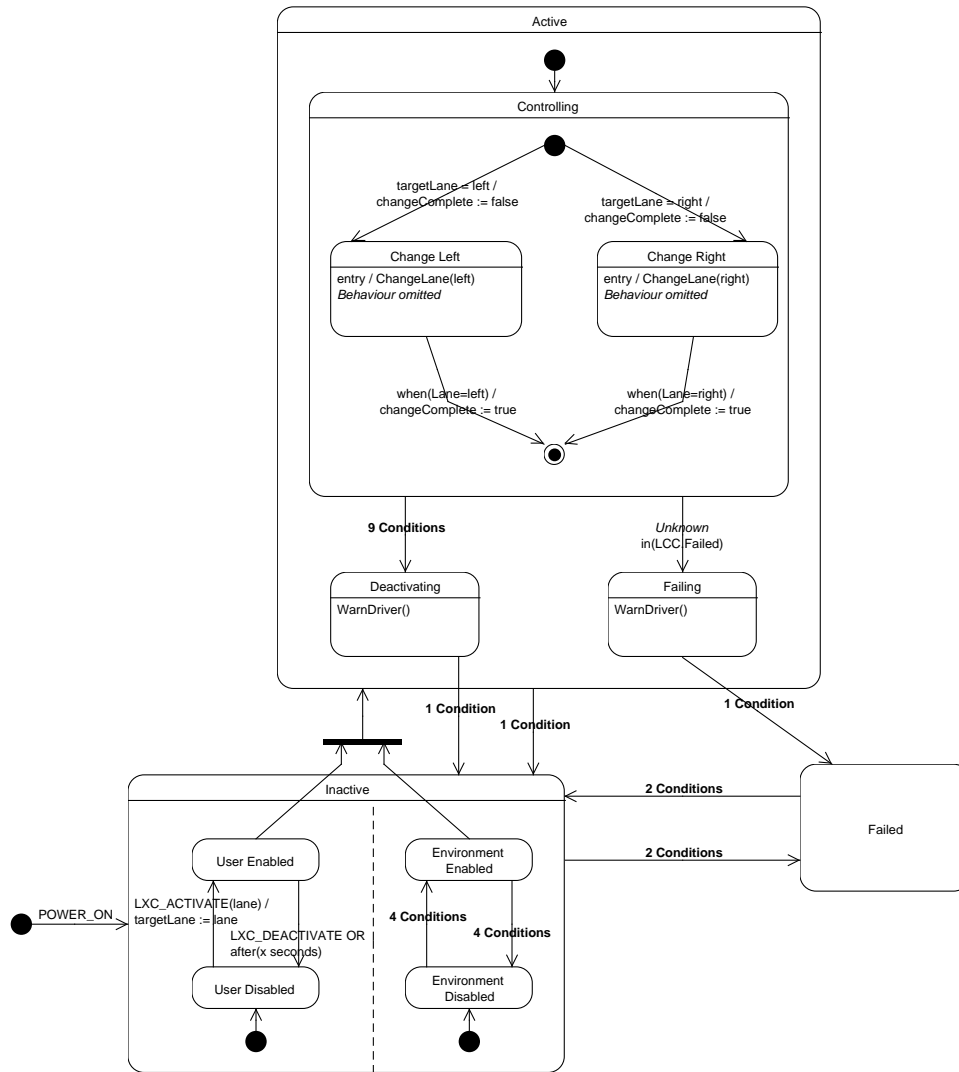


Figure A.15: The Lane Change Control (LXC) feature.

LXC automatically changes the vehicle's current lane to one requested by the driver. The driver indicates the destination lane and, when it is safe, LXC moves the vehicle to that lane.

In the GM requirements documents LXC and LCC were modelled as one feature. We have chosen to model them as two separate features in the case study because, in terms of their behaviour, they are not very tightly coupled. If they were modelled as one feature then the entire behaviour for LXC would be included in the Active state of LCC. By treating LXC as a separate feature a dependency is added that requires LCC is active before LXC can activate.

LXC is modelled using the Unordered Enabling extension of Inactive and the Controlling, Deactivating, and Failing states of Active. The enabling process checks that it is safe to change lanes before transitioning to Active. LXC does not use the Monitoring state because it is always controlling the vehicle while Active. In the event that the driver cancels an in-progress lane change or a failure occurs then LXC will transition to Deactivating or Failing, respectively, and ensure the driver has control of the vehicle before relinquishing control.

A.2.5 Road Change Alert (RCA)

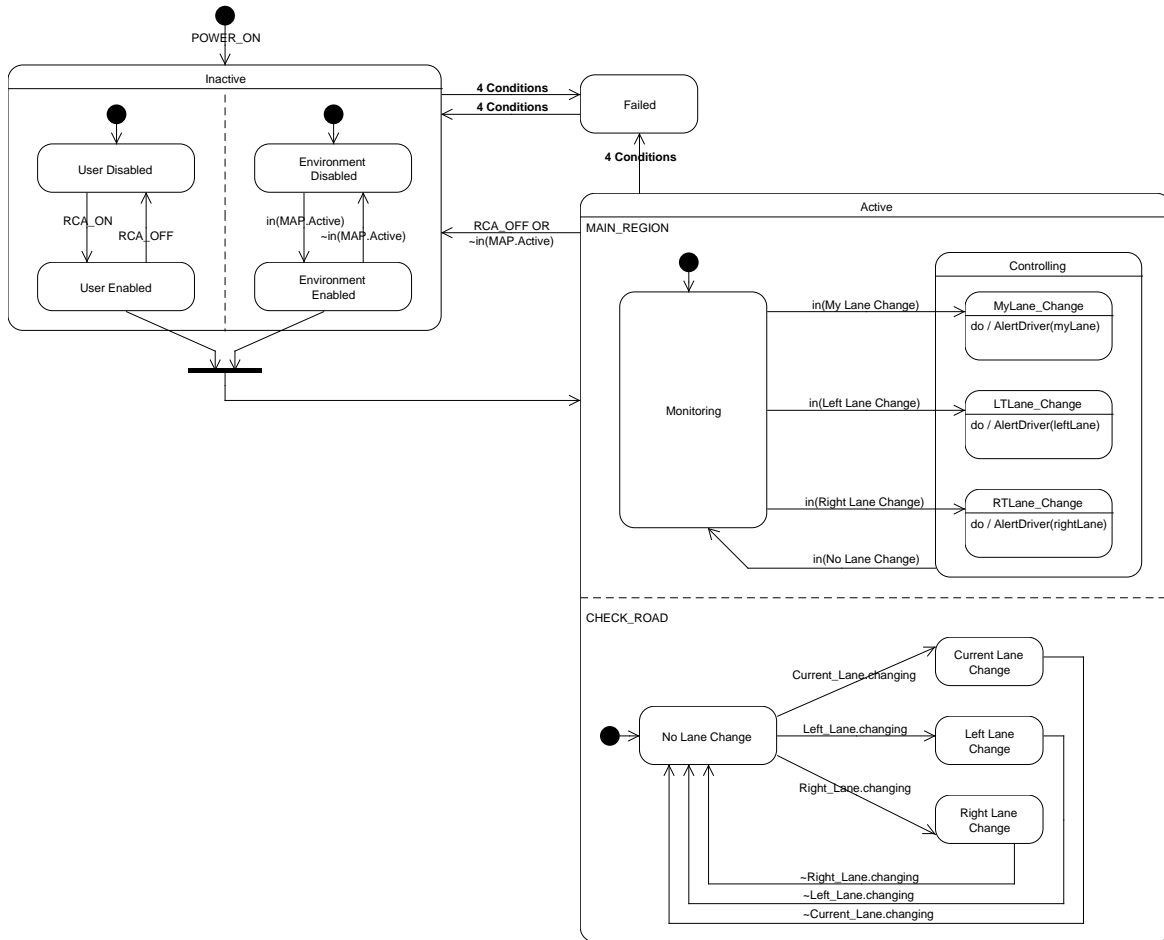


Figure A.16: The Road Change Alert (RCA) feature.

RCA alerts the driver (using visual, audible, and/or haptic feedback) when the road ahead of the vehicle is changing. A change to the road is defined as either multiple lanes merging into one or one lane forking into multiple lanes.

RCA is modelled using the Unordered Enabling extension of Inactive. The enabling process only waits for the driver to enable RCA and checks that the MAP feature (which provides Global Positioning data) is active. The Active state has two concurrent regions:

a main region that includes the Monitoring and Controlling states, and a separate region that continually checks for changes to the road. When the road is not changing, the RCA state machine waits in Monitoring. When a road change is detected, the state machine transitions to Controlling. By using the concurrent region, the transitions between the Monitoring and the Controlling sub-states are simplified because they do not directly refer to the domain model. The requirements for RCA do not discuss the case where multiple lanes are changing at one time. We have therefore modelled the state machine so that the first change detected has priority.

The requirements for RCA provide several reasons why it may fail. These involve sensor errors, communication failures, or the MAP feature failing.

A.3 Heating, Ventilation, and Air Conditioning Features

A.3.1 Air Recirculation Control (ARC)

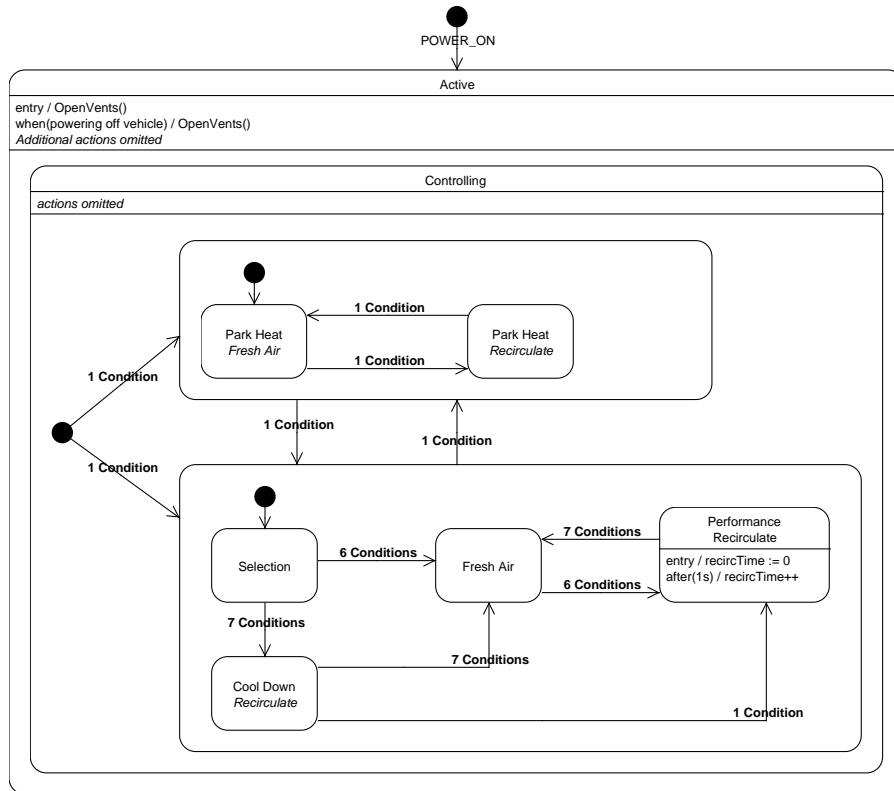


Figure A.17: The Air Recirculation Control (ARC) feature.

The ARC feature regulates the blending of outside air with recirculated air in the vehicle cabin. This is done by opening and closing the vehicle's outside air vents to let in fresh air. ARC takes into account things like current cabin temperature and the state of the vehicle's heater or air conditioning system when deciding if it should open or close the vehicle's outside air vents.

ARC does not have any inactive behaviour. However, we are not certain if we have the complete set of requirements for ARC. ARC is modelled using the Controlling sub-state of Active. If the vehicle's heater is active then the state machine uses the *Park Heat* states to control the amount of outside and recirculated air in the cabin. If not, then the remaining four states in Controlling implement the controlling behaviour. ARC's requirements do not discuss failures.

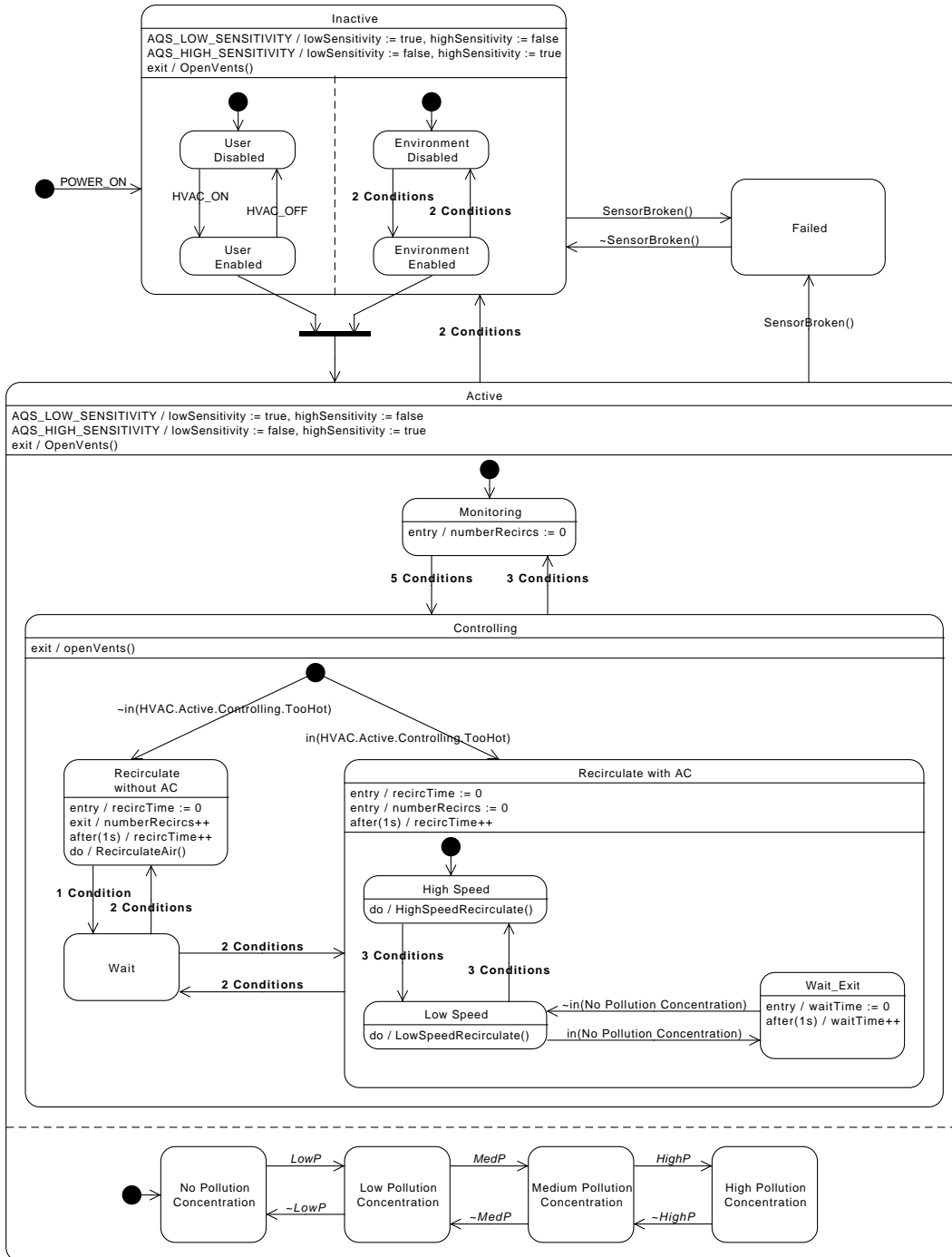


Figure A.18: The Air Quality System (AQS) feature.

A.3.2 Air Quality System (AQS)

The AQS feature (see Figure A.18) minimizes the level of pollution in the vehicle's cabin by controlling the mixing of outside air and recirculated air. The requirements do not define the exact meaning of pollution. When the amount of pollution in the vehicle cabin is low, the outside air vents are opened and fresh air is let into the vehicle. If the pollution concentration is high, then the vents are closed and the air is recirculated (the air passes through filters that clean it). If the air conditioner is active then the air is recirculated differently (the difference is not explained).

The AQS is modelled using the Unordered Enabling extension of Inactive and the Monitoring and Controlling sub-states of Active, as well as a separate concurrent region that models the pollution concentration in the vehicle. When the pollution concentration is low, the state machine waits in the Monitoring sub-state. If the pollution concentration becomes too high, then the state machine transitions to the Controlling sub-state and lowers the pollution concentration in the vehicle cabin. We have also included the separate concurrent region for Active that keeps track of the amount of pollution in the vehicle. The pollution is monitored using discrete steps and depending on the current pollution level one of the 4 states is entered. Using the separate region simplifies the transitions within the main concurrent region of Active.

The only failure requirement that AQS mentions is that if any of the pollution sensors break, then AQS should fail.

A.3.3 Recirculation Control Run (RUN)

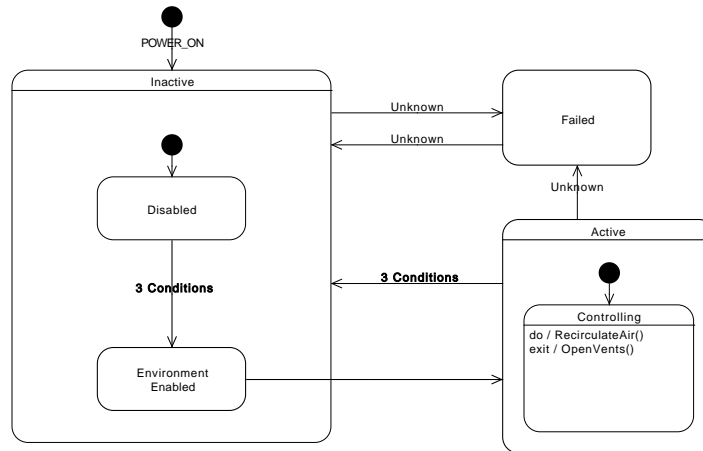


Figure A.19: The Recirculation Control Run (RUN) feature.

The RUN feature monitors the vehicle's internal cabin environment and ensures that the windows do not become fogged. The RUN feature operates by opening and closing the outside air vents to reduce any fogging of the windshield. The RUN feature can not be disabled by the vehicle driver. The RUN feature is modelled using the Ordered Enabling extension of Inactive, and the Controlling sub-state of Active. We did not have access to any failure requirements for RUN.

One thing you may have noticed is that the ARC, AQS, and RUN features all use the vehicle's outside air vents. The requirements documents did not discuss any feature interactions that may occur (these may be discussed in another document that we did not have access to), so we have not modelled any interactions resolution queries between the features.

Appendix B

Cruise Control in DOORS

This chapter contains the complete set of DOORS requirements for the Cruise Control (CC) feature.

B.1 The Primary DOORS Template

This section contains several figures that present the requirements for the CC feature as entered in the primary DOORS template.

1 Cruise Control Requirements

1.1 Description

The Cruise Control (CC) feature will maintain the speed of the vehicle at a driver-specified value.

1.2 List of Environmental Conditions and User Actions

1.2.1 User Actions

POWER_ON - Sent when the vehicle is first turned on.

CRUISE_ON - Sent when the vehicle driver presses the CC feature's On button.

CRUISE_OFF - Sent when the vehicle driver presses the CC feature's Off button. Note that the On and Off button will likely be the same button, although the signals sent to this feature will still correspond to an On and Off action.

CRUISE_SET - Sets the cruising speed (i.e., the speed to maintain the vehicle at) to the current speed when this action is performed.

BRAKE_PRESSED - This signal is sent to the CC feature when the driver presses on the vehicle's brake pedal.

ACCEL_PRESSED - This signal is sent to the CC feature when the driver presses on the vehicle's accelerator pedal.

INC_SPEED(x) - This signal is sent to the CC feature when the driver increments the cruising speed of the vehicle. The argument x holds the exact value (in km/h or mph) by which to increase the cruising speed.

DEC_SPEED(y) - This signal is sent to the CC feature when the driver decrements the cruising speed of the vehicle. The arguments y holds the exact value (in km/h or mph) by which to decrease the cruising speed.

1.2.2 Environmental Conditions

vehicleSpeed - This variable holds the current speed of the vehicle.

1.3 State-Machine Model of Feature

A state-machine model of the CC feature.

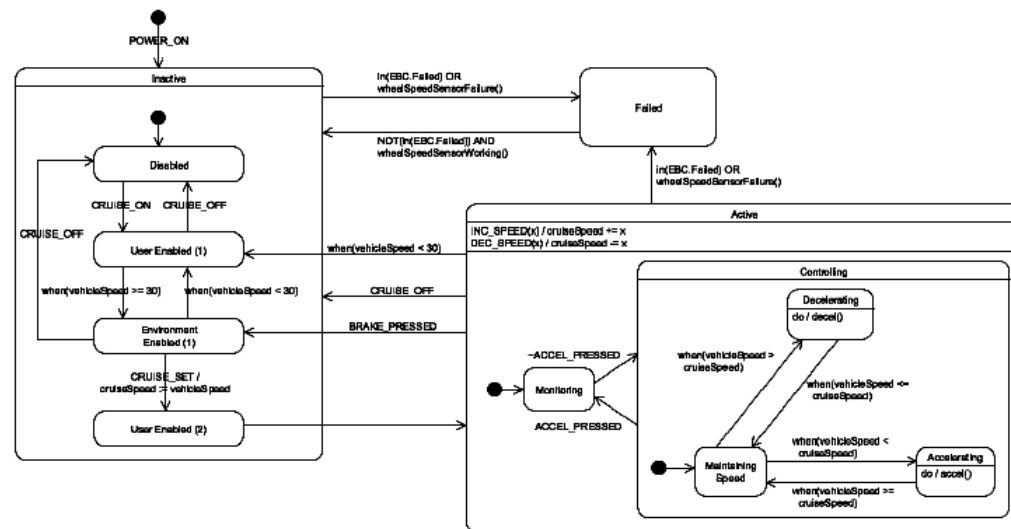


Figure B.1: The Rational DOORS requirements for the Cruise Control (CC) feature (Main Template: part 1/3).

1.4 Inactive
The Inactive state is the initial state of the CC feature. It is entered when the vehicle is first powered on.
While the CC feature is inactive the vehicle driver cannot increase or decrease the vehicle's cruising speed.
1.4.1 Enabling Type
Ordered
1.4.2 Link to several enabling steps here. These are found as separate modules in the same folder.
1.4.2.1 Disabled
1.4.2.2 User Enabled (1)
1.4.2.3 Environment Enabled (1)
1.4.2.4 User Enabled (2)
1.5 Active
1.5.1 General Requirements
At any time while the CC feature is Active the vehicle driver should be able to increment or decrement the cruising speed of the vehicle (INC_SPEED and DEC_SPEED user actions).
If the vehicle's speed drops below 30 km/h the CC feature should deactivate immediately and transition to the User Enabled (1) sub-state of Inactive.
If the vehicle's brake pedal is pressed (BRAKE_PRESSED user action) then the CC feature should deactivate immediately and transition to the Environment Enabled (1) sub-state of Inactive.
If the vehicle's driver turns off the CC feature (CRUISE_OFF user action) then the feature should deactivate immediately and transition to the Disabled sub-state of Inactive.
If the Electronic Brake Control (EBC) feature fails then the CC feature should transition to the Failed state.
If there is a failure of any of the wheel speed sensors then the CC feature should transition to the Failed state.
1.5.2 Monitoring
The Active sub-machine initializes in Monitoring.
There is not specific behaviour that only occurs in the Monitoring state.
If the vehicle's driver is not pressing on the accelerator pedal (ACCEL_PRESSED user action) then the machine transitions to the Controlling state.
1.5.3 Controlling
The Controlling state's sub-machine models the speed maintaining behaviour of the CC feature. The Controlling sub-machine has three states: Maintaining Speed, Decelerating, and Accelerating.
1.5.3-1.1 Maintaining Speed
Maintaining Speed is the initial state of the Controlling sub-machine.
If the vehicle's current speed becomes greater than the cruising speed (e.g., if the vehicle is going down a hill) then the machine should transition to the Decelerating state.
If the vehicle's current speed becomes less than the cruising speed (e.g., if the vehicle is going up a hill) then the machine should transition to the Accelerating state.
1.5.3-1.2 Decelerating
While in the Decelerating state the CC feature should be actively reducing the vehicle's speed (modelled in the state machine in Section 1.3 using the decel function).
If the vehicle's speed becomes less than or equal to the cruising speed then the machine should transition to the Maintaining Speed state.

Figure B.2: The Rational DOORS requirements for the Cruise Control (CC) feature (Main Template: part 2/3).

1.5.3-1.3 Accelerating
While in the Accelerating state the CC feature should be actively increasing the vehicle's speed (modelled in the state machine in Section 1.3 using the accel function).
If the vehicle's speed becomes greater than or equal to the cruising speed then the machine should transition to the Maintaining Speed state.
1.6 Failed
The failed behaviour requirements.
1.7 Undefined Functions
1.7.1 Accel()
This function will accelerate the vehicle. In our simple CC feature we assume that the acceleration is linear, but in reality the requirements for an accel() function would likely be much more complex.
1.7.2 Decel()
This function will decelerate the vehicle. In our simple CC feature we assume that the deceleration is linear, but in reality the requirements for a decel() function would likely be much more complex.

Figure B.3: The Rational DOORS requirements for the Cruise Control (CC) feature (Main Template: part 3/3).

B.2 Templates for the Enabling Stages

This section presents the requirements of the four enabling stages as structured with the DOORS enabling template. These four stages link to the four stage placeholders in the primary DOORS template in Figure B.2.

1 Step Disabled
1.1 State Requirements
No behavioural requirements.
1.2 Proceed to User Enabled (1)
In order to proceed to the next state (User Enabled (1)) the driver must perform the CRUISE_ON action.

Figure B.4: The Rational DOORS requirements for the Cruise Control (CC) feature (Enabling State: Disabled).

1 Step User Enabled (1): User Enabled
1.1 State Requirements
No behavioural requirements.
1.2 Proceed to Environment Enabled (1)
To proceed to the next state (Environment Enabled (1)) the vehicle speed must be greater than or equal to 30 km/h.
1.3 Return to Disabled
To return to the previous state (Disabled) the vehicle driver must perform the CRUISE_OFF action.

Figure B.5: The Rational DOORS requirements for the Cruise Control (CC) feature (Enabling State: User Enabled 1).

1 Step Environment Enabled (1): Environment Enabled
1.1 State Requirements
No specific behaviour.
1.2 Proceed to User Enabled (2)
To proceed to the next state (User Enabled (2)) the vehicle driver must perform the CRUISE_SET action. When the driver performs this action, the cruising speed of the vehicle is set to the current speed of the vehicle (i.e., <code>cruiseSpeed := vehicleSpeed</code>).
1.3 Return to immediately User Enabled (1)
Return to the previous state (User Enabled (1)) if the vehicle speed becomes less than 30 km/h.
1.4 Skip backwards to Disabled
If the vehicle driver performs the CRUISE_OFF action then the CC feature returns to the Disabled state.

Figure B.6: The Rational DOORS requirements for the Cruise Control (CC) feature (Enabling State: Environment Enabled 1).

1 Step User Enabled (2): User Enabled: FINAL
1.1 State Requirements
Immediately upon entry to this state the CC feature should transition to Active.

Figure B.7: The Rational DOORS requirements for the Cruise Control (CC) feature (Enabling State: User Enabled 2).

Appendix C

User Study Materials

This chapter contains the tutorial for the Pattern+Interface group of the user study and all three versions of the study questionnaire. We have only included one version of the tutorial but point out where the differences between the three tutorials arise.

C.1 User Study Tutorial

The user study tutorial that we include here is for the Pattern+Interface group. The tutorials for the Control and Pattern groups did not present some of the information present in the Pattern+Interface tutorial (we point out these differences using blue annotations). We have included all three versions of the Road Change Alert state-machine model (one for each participant group) in this version of the tutorial.

Tutorial

Introduction

Model-driven engineering is a relatively new development methodology in which models, instead of code, are the primary development artifacts. One particular area where model-driven engineering is being used is in automotive software. A type of model used in model-driven engineering are behaviour models, which specify the behaviour of a system (i.e., how it reacts to different inputs). A popular type of behaviour model is state-machine models.

The purpose of this tutorial is three-fold: (1) to familiarize you with some of the complex syntactic structures of state machines that are commonly used to model automotive software, (2) to describe a pattern for designing state-machine models of automotive features, and (3) to describe a generic state-machine interface that arises by using the pattern.

During the study, we will ask you to read a state machine that is modelled using the pattern and to write a state-machine model using the pattern. You are allowed to keep the tutorial materials and use them during the study. However, we ask that you do not share the tutorial materials, or your experiences during the tutorial, with others until after the study is finished.

The Vehicle

The software in a vehicle is decomposed into distinct software features. We define a feature as a coherent and identifiable unit of system functionality. For example, Cruise Control (which controls the vehicle at a driver-set speed) is a single feature in a vehicle, as is Anti-Lock Brakes (which aids braking), and Automatic Headlights (which turns on the vehicle's headlights when the environmental illumination becomes too low). There is a one-to-one correspondence between state machines and features (i.e., Cruise Control has a single state machine that describes its behavioural requirements).

You may find the domain model presented in this section useful when reading and writing state-machine models. The domain model contains concepts in the environment (e.g., *Vehicle*) and their attributes (e.g., *Vehicle.speed*), and associations between concepts (e.g., *Vehicle.oncoming*) which can also have attributes (e.g., *Vehicle.oncoming.distance*). The domain model also describes the various **features** that are in the vehicle (e.g., *Adaptive Cruise Control (ACC)*), their public data (e.g., *ACC.cruiseSpeed*), and their input signals (e.g., *ACC.ACC_ON*).

The domain model is the source of inputs to a state-machine model. A **user action** is an action performed directly by the user or human operator (e.g., the user turning on the feature) modelled as a feature's input signals (e.g., *ACC.ACC_ON*). An **environmental condition** is a predicate over domain-model variables (e.g., *Vehicle.speed > 30*) or over the current state of another feature. The domain model is also the destination of actions performed by the state-machine model.

The “dot” notation is used to reference objects and attributes in the domain model (similar to referencing an object's data fields in OOP languages). For example, a state-machine model

could determine the current speed of the vehicle by checking the domain model's *Vehicle.speed* attribute. The same notation is used to navigate over associations to write expressions about related objects. For example, a state-machine model can determine if the current lane that the vehicle is in is changing by using the expression *Vehicle.currentLane.changing*.

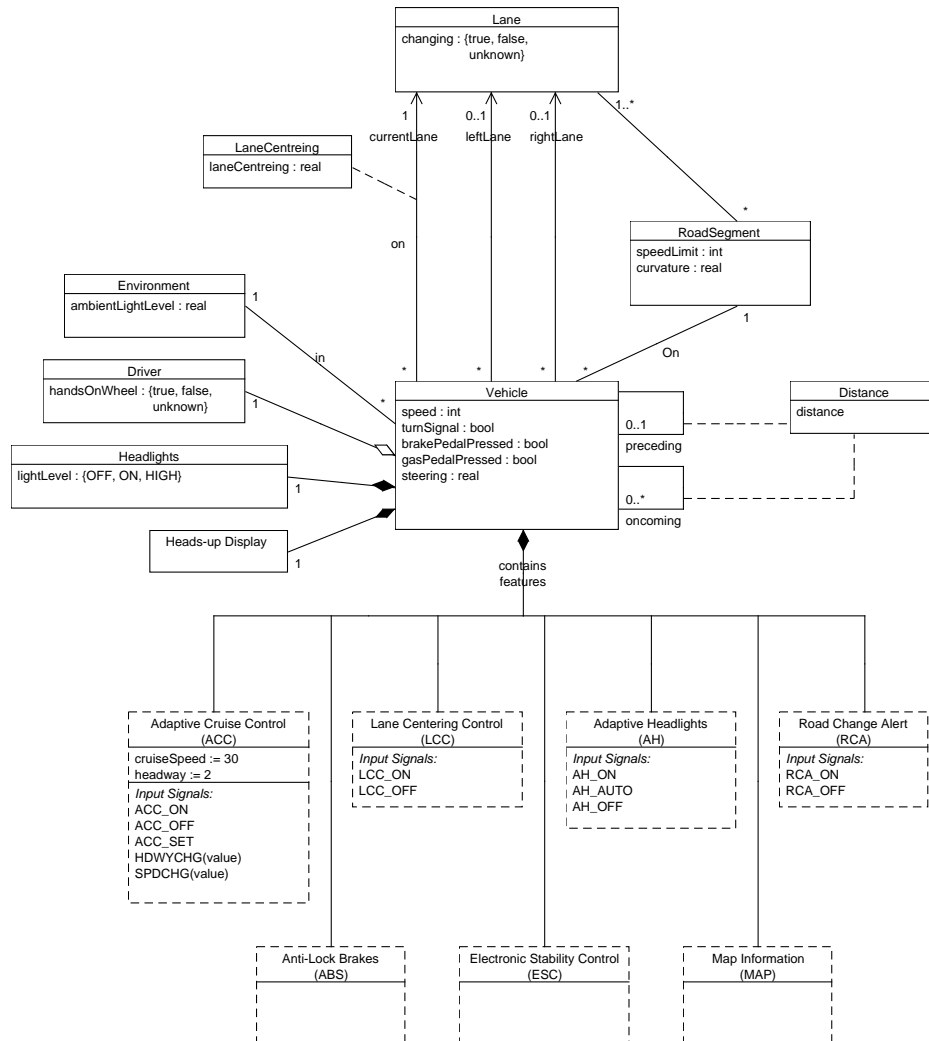


Figure 1: A simple domain model for several features.

State Machines

We assume that you have some familiarity with state machines created using a Statecharts-like notation and semantics (e.g., UML State Machines). This section explains some of the more complex features of state machines that may be useful when reading and writing state-machine models of real-world software.

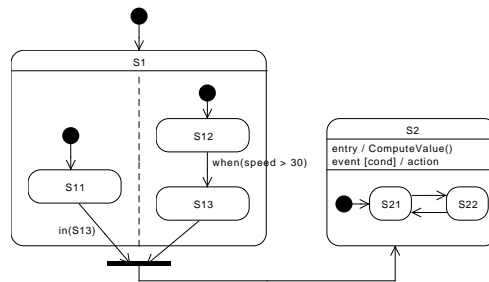


Figure 2: State-machine notation.

Consider the state machine exhibited in Figure 2. A state may contain sub-states; in this case, the former is called a *superstate* (e.g., S1, S2). A superstate may be decomposed into one or more *concurrent regions* that are separated by dashed lines (e.g., S1); regions model orthogonal behaviour that can occur in parallel. A transition from a black circle to a state designates the initial state of a machine (e.g., S1). If a transition's destination is a superstate, then the next state is the initial state of the superstate's sub-machine (e.g., S21) or the initial states of the superstate's regions (e.g., S11 and S12).

Transitions can be annotated with an event, which is a user action or predicate over the environmental variables; a guard condition, which is a boolean condition over environmental variables; and a set of actions on environmental variables – all of which are optional. Events initiated by the user are denoted in upper-case and all other conditions are lower-case. Actions are prefaced with a slash. The following are several examples of valid transition labels:

1. Vehicle.speed > 30 / Headlights.lightLevel := ON
2. LCC_ON
3. SPDCHG(value) / ACC.cruiseSpeed += value

Note how literals in the labels refer to events and variables from the domain model. When a transition is unlabelled it is executed as soon as its source state is entered.

Transition annotation *when(c)* refers to the event of condition *c* becoming true (e.g., *when(Vehicle.speed > 30)*). Transition annotation *[in(S)]* is a condition that is satisfied when the system's execution is in state *S* (e.g., *in(S13)*); state *S* might refer to a state in another feature. The *join* pseudo-state (modelled as a black bar) is used to aggregate multiple transitions (e.g., the tran-

sitions from source states S11 and S13 to destination state S2). The join’s outgoing transition executes only when all of its incoming transitions are enabled.

A state may be annotated with actions that are enabled by events and guard conditions (e.g., S2). Such a state action is performed whenever its triggering event occurs while the system’s execution is in the state and the guard condition is true.

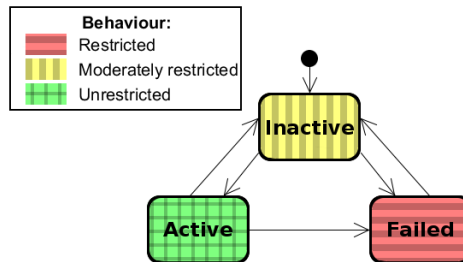
A feature’s requirements may specify some complex data computation that would be difficult and time consuming to model. This computation can be abstracted as an *undefined function* that models the computation being performed without explicitly defining it. Undefined functions can be events, guard conditions, or actions. For example, in Figure 2 the *ComputeValue()* function is an action that is executed when state S2 is entered.

A pattern for modelling state-machines

Included with Pattern and Pattern+Interface groups

In this section we describe a pattern for decomposing and structuring the behaviour model of a software feature, expressed as a state machine, according to modes of operation that are common to all features. High-level modes are *Active* (which captures a feature’s essential requirements), *Inactive* (which captures a feature’s enabling and disabling requirements), and *Failed* (which captures a feature’s failure and recovery requirements).

The following model shows how the *Inactive*, *Active*, and *Failed* states relate to one another:



The important thing to note is that features normally start in the Inactive state, and when a feature recovers from a failure it always transitions to Inactive.

The pattern’s high-level states correspond to distinct major modes of operation. Consider the different ways in which a feature can interact with its environment:

- the feature monitors the environment
- the feature acts on the environment
- the environment monitors the feature
- the environment acts on the feature

Each state reflects different types of interaction. In restricted (red) states, the only allowable interaction is that the feature can monitor the environment – to determine if any of the state’s outgoing transitions are enabled. For example, a feature that has Failed can monitor the environment for signs that recovery conditions have been met. In moderately restricted (yellow) states, the feature can monitor the environment and the environment can act on the feature. For example, it may be possible for a user to manipulate feature settings when the feature is still Inactive (e.g., a driver can set the cruising speed before the cruise-control feature becomes Active). In unrestricted (green) states, all four types of interactions are allowed. In this manner, the pattern’s high-level states partition the features’ behaviours into separate modes of operation.

Inactive Sub-Patterns

Sub-patterns for the Inactive mode provide advice on how to decompose and structure the enabling process of a feature, according to the type and order of enabling conditions (user actions or environmental conditions).

1. Ordered Enabling Sub-Pattern of Inactive

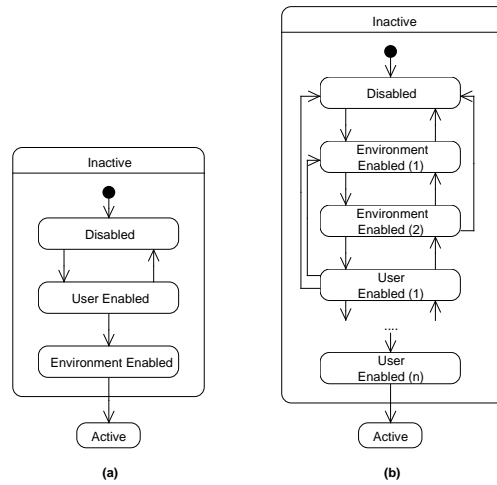


Figure 3: The Ordered Enabling sub-pattern.

The Ordered Enabling sub-pattern applies when a feature becomes enabled in stages. The Inactive sub-machine is a sequence of user actions and environmental conditions that must be satisfied in the specified order. Figure 3a shows the default sub-pattern where user actions precede environmental-condition checks. There is a second default sub-pattern (not shown) where environmental conditions must hold before user actions are recognized. In the sub-patterns, each transition can be triggered by a combination (i.e., conjunctions,

disjunctions, negations) of user actions or a combination of environmental conditions. When the final state in the sequence is reached, the feature transitions automatically to the Active state.

The Inactive state in Figure 3b uses the Ordered Enabling sub-pattern to specify a multi-stage enabling process. The enabling sequence may include back transitions from later states in the sequence to earlier states, if enabling conditions become unsatisfied and cause the feature to revert to a less-enabled state. As in the default sub-pattern, when the final state in the sequence is reached, the feature transitions automatically to the Active state.

2. Unordered Enabling Sub-Pattern of Inactive

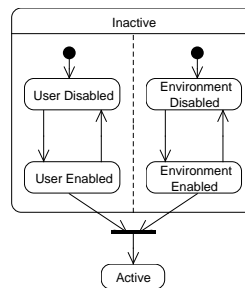


Figure 4: The Unordered Enabling sub-pattern.

It often does not matter in what order a feature’s enabling conditions become true; as soon as they all hold, the feature becomes Active. The Unordered Enabling sub-pattern applies in these situations (shown in Figure 4). The concurrent regions separate user actions (on the left) from the environmental conditions (on the right). A transition can be triggered by a combination of user actions or a combination of environmental conditions. When all of the regions are simultaneously in their most-enabled state, the feature transitions automatically to the Active state. We model this behaviour using a join pseudo-state whose source states are User Enabled and Environment Enabled and whose destination state is Active.

We recommend using the Unordered Enabling sub-pattern when a feature’s enabling process includes only user actions or only environmental conditions, and not both. In such a case, the region that has no enabling conditions simply initiates in its enabled sub-state. This makes it explicit that the user actions or environmental conditions have been considered, but that none exist.

The Active Sub-Pattern

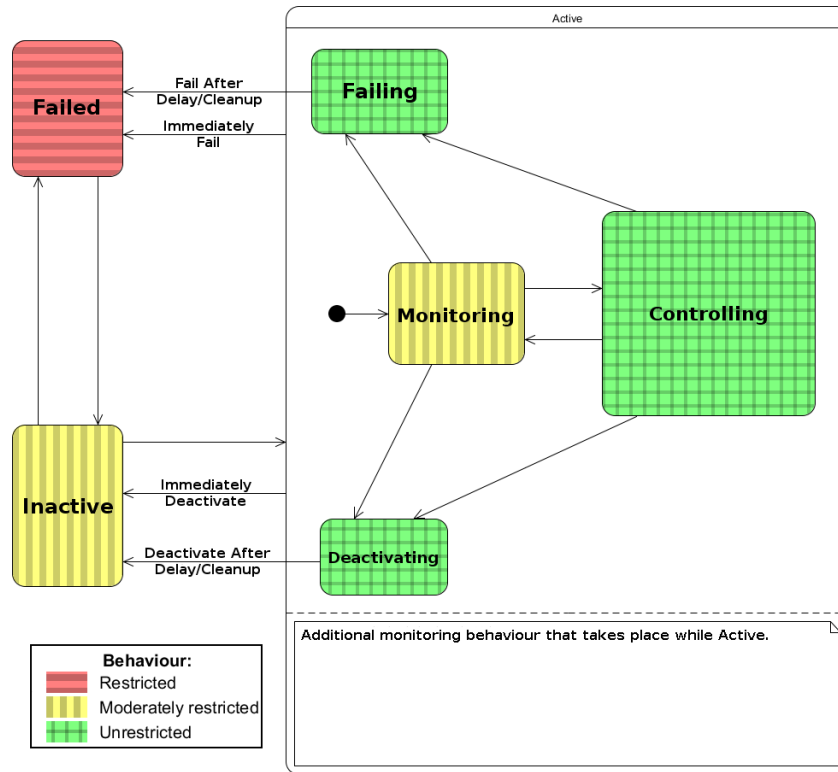


Figure 5: The Active sub-pattern.

The Active sub-pattern provides advice on how to model the active behaviour of a feature. Figure 5 shows the Active sub-pattern and how it relates to the two other high-level states (i.e., Inactive and Failed).

The Active sub-pattern is composed of four states:

- Monitoring:** When in the monitoring state, the feature is only monitoring the behaviour of the vehicle; it is not actively controlling the vehicle's behaviour.
- Controlling:** This is the state in which a feature actively affects the vehicle's behaviour. A feature's Controlling state typically contains one or more sub-machines that model the controlling behaviour of the feature.
- Deactivating:** In this state, a feature is in the process of deactivating, but needs to perform some actions before it becomes Inactive (e.g., a feature that automates

some driver’s task will notify the driver that it is deactivating and will attempt to remain operational for some time to allow the driver to resume responsibility for that task).

- (d) **Failing:** The purpose of this state is similar to the purpose of the Deactivating state – when a feature is failing, it warns the driver and attempts to remain operational temporarily to allow the driver to resume responsibility over the feature’s task.

An activating feature normally initializes in the Monitoring sub-state and transitions between the Monitoring and Controlling sub-states, depending on whether the feature is currently performing some actions that control some aspect of the vehicle or not. An Active feature will transition to Deactivating as an intermediate sub-state towards transitioning to Inactive, or to Failing as an intermediate sub-state towards transitioning to Failed. The events, conditions, or actions with which these transitions are labelled are feature specific and thus are not part of the pattern.

Some features will transition from the Monitoring or Controlling sub-states directly to Inactive or Failed (either because no intermediate state is necessary, or because no intermediate state is possible). Such transitions should be from the border of the Active superstate to the Inactive or Failed states.

The colours of the Active sub-states in Figure 5 depict the ways in which the feature can interact with its environment: In moderately restricted (yellow) states, the feature can monitor the environment and the environment can act on the feature (e.g., the driver can modify feature settings). In unrestricted (green) states, all types of interactions are allowed.

In addition to the four states within the pattern, there may be additional concurrent regions within Active. Those additional regions are useful when the feature continuously monitors multiple environmental phenomena when in the Active state.

Referencing Other Features

In a typical vehicle, there may be hundreds of features that control the vehicle, log information, interface with the driver, etc. Sometimes a feature will rely on another feature being in a specific behaviour mode. For example, a feature **A** may be able to activate only when another feature **B** is active.

When features are modelled according to the pattern, *all* features have the same high-level behaviour model. This means that the high-level states can serve as a generic feature interface of a feature. As a modeller, this simplifies your task because you know that all of the features in the vehicle have an Inactive, Active, and a Failed superstate. You also know what kind of behaviour is occurring (i.e., restricted, moderately restricted, or unrestricted) when a feature is in each of those states. Consider the example in the previous paragraph. The model of feature **A** could incorporate condition *in(B.Active)* which evaluates to true whenever feature **B** is in its Active state. When features are modelled according to the proposed pattern, the engineer who

models feature **A** can confidently include such a condition in his model without even consulting the model of feature **B**.

Example: Road Change Alert

The RCA model varies depending on group

The automotive feature Road Change Alert (RCA) alerts the driver if the lane in front of the vehicle is about to change (e.g., if the lane is about to merge into another lane, or fork into multiple lanes).

When the road ahead of the vehicle is changing, the driver will be alerted by a flashing light on the vehicle dashboard and by an audible warning (the warning does not distinguish between the ways in which the lane is changing). The alert will continue until the vehicle has passed the changing portion of the road.

The Control group's model

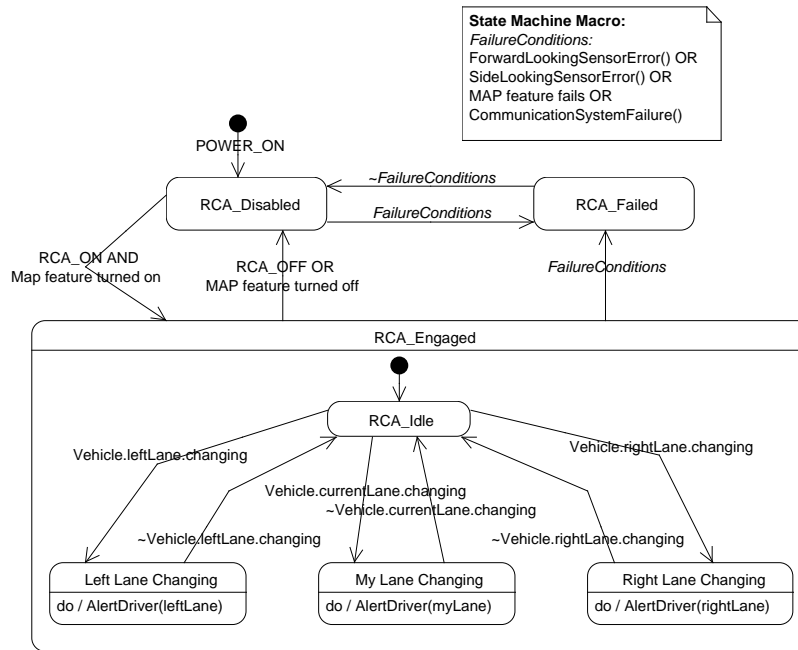
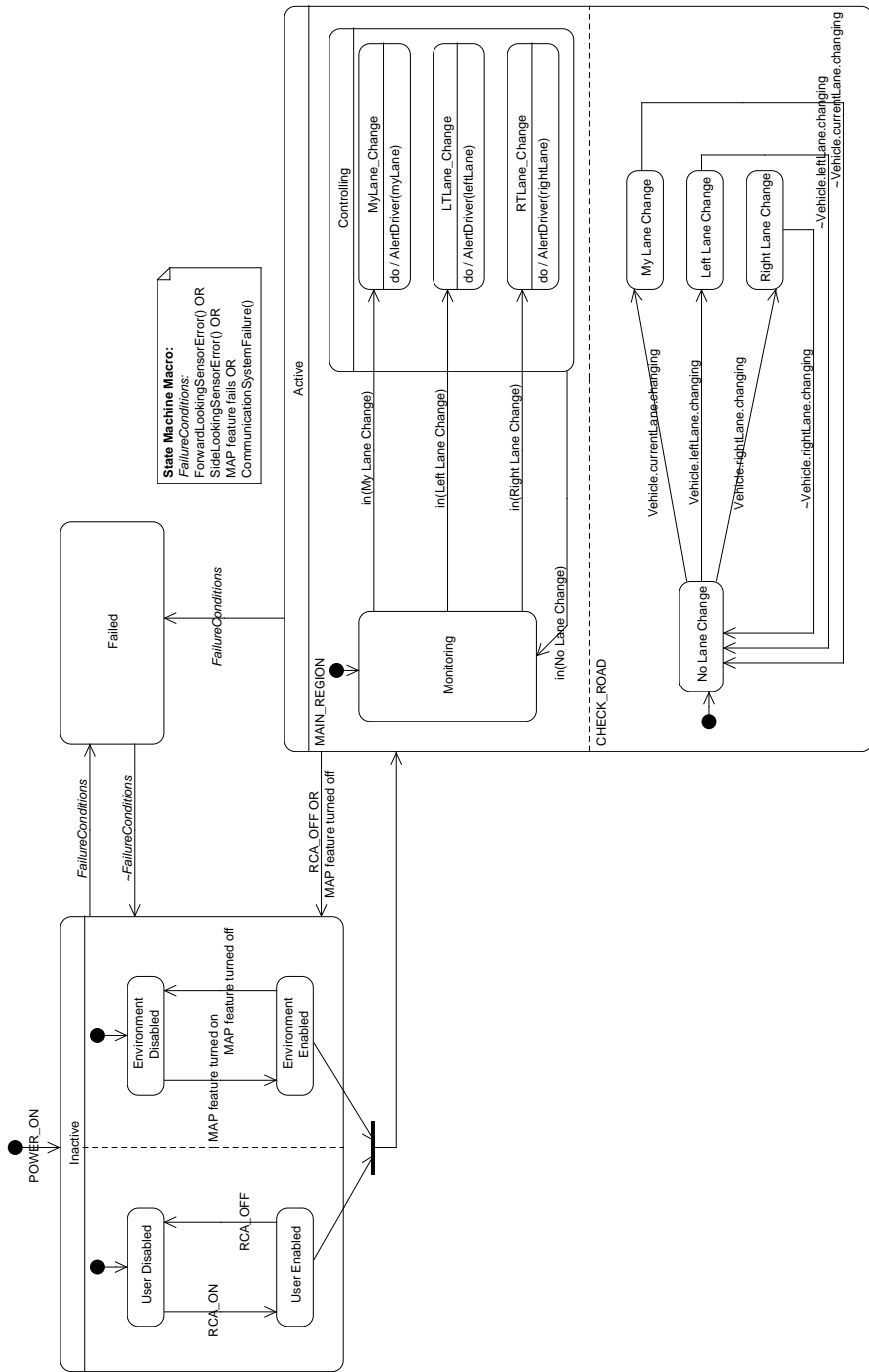
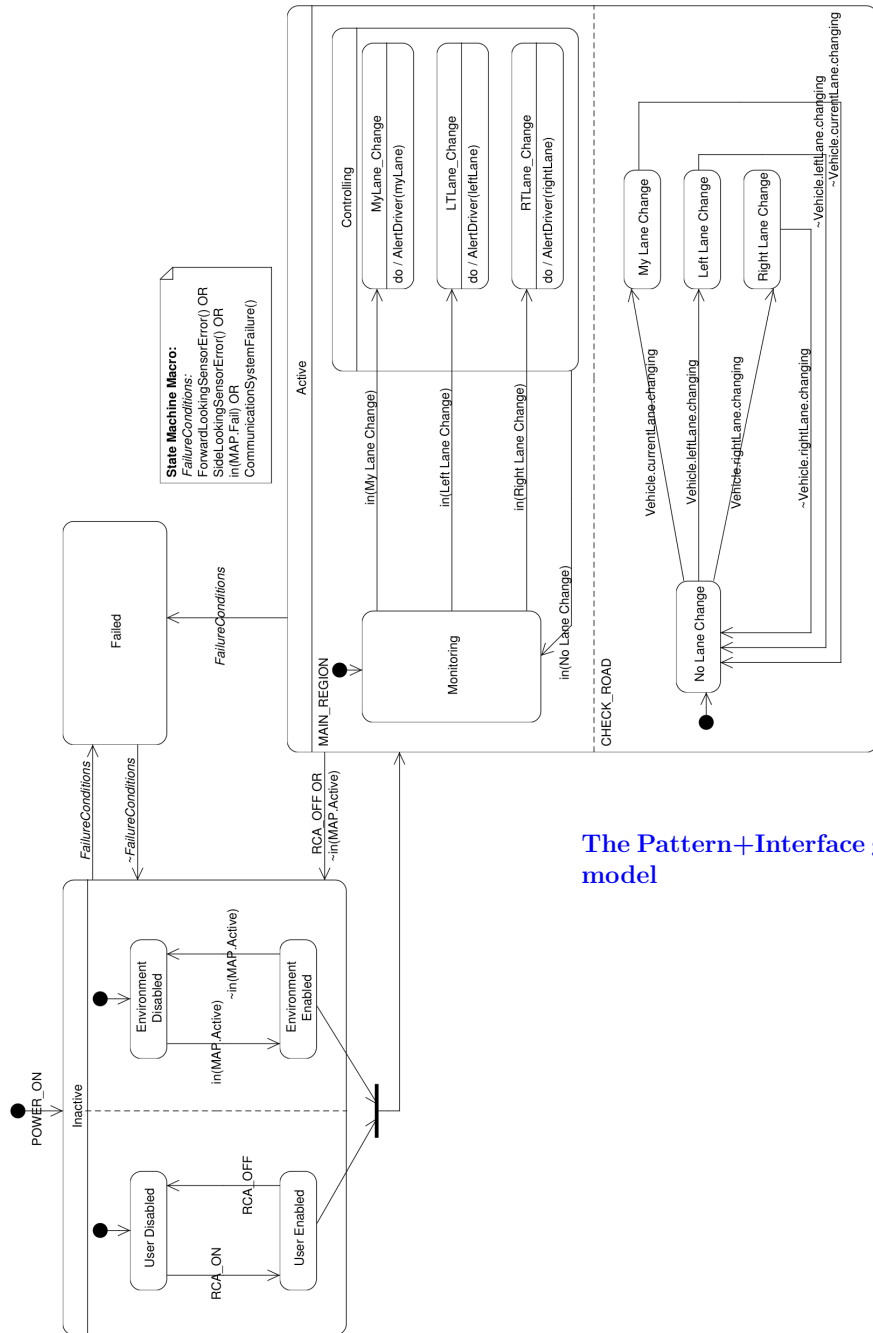


Figure 6: The Road Change Alert feature.



el



The Pattern+Interface group's model

Figure 8: The Road Change Alert feature.

RCA applies the Unordered Enabling sub-pattern of Inactive and a subset of the Active sub-pattern. To activate RCA, the driver must manually press an RCA-on button (resulting in input event RCA_ON). RCA also checks to ensure that the MAP feature is active (note how even though the MAP feature is not provided, we know that it has an Active state that implements its behaviour).

Once activated, RCA will be in state Monitoring until it notices that the road is changing in front of the vehicle. These road changes are modelled using several events and conditions from the domain model (e.g., RoadSegment.leftLane.changing) that reflect threshold readings from vehicle sensors or positioning data from the MAP feature. When one of the lanes near the vehicle is changing, the lane-changing condition of the appropriate lane (current lane, left lane, right lane) becomes true, which causes the sub-machine MAIN_REGION to transition to the sub-state of Controlling that corresponds to the lane that is changing. When the vehicle has passed the changing portion of the road, RCA transitions back to the Monitoring state.

RCA deactivates when the driver deactivates RCA (by sending the RCA_OFF event) or the feature MAP becomes Inactive. RCA fails due to sensor failure, MAP failure, or if there is a failure within the vehicle's communication network.

Modelling an example feature

Your task is to write a state-machine model for an automobile feature called the Automatic Headlight feature (AH). When active, AH turns the vehicle headlights off and on automatically depending on the level of illumination outside the car. It will also turn on high beams if the distance to the next vehicle or any oncoming vehicles is sufficiently far to avoid blinding the other drivers with high-beam headlights. Thus, the appropriate headlight brightness level is calculated using the presence of a preceding or oncoming vehicle and the luminosity level of the environment.

The driver interacts with the AH feature via buttons near the steering wheel. The driver can turn the headlight knob to ON, OFF, or AUTO (in which automatic headlights are engaged). The AH feature will activate only when the Adaptive Cruise Control (ACC) feature is already active.

In addition to the AH knob, the other inputs to the feature are:

- `Vehicle.preceding` $\neq \emptyset$ - *This condition is true when there is another vehicle ahead of this one within a threshold distance. If the value of this condition is Unknown, then the object-detecting sensor has failed and AH fails. When the value becomes known again, AH recovers from the failure.*
- `Vehicle.oncoming` $\neq \emptyset$ - *This condition is true when there is another vehicle approaching this one within a threshold distance. If the value of this condition is Unknown, then the object-detecting sensor has failed and AH fails. When the value becomes known again, AH recovers from the failure.*
- `Environment.ambientLightLevel` : real - *The valid range for the variable is 0 to 1. If the value lies outside of the accepted range then the light-detecting sensor has failed and AH also fails. When the value returns to the valid range, AH recovers from the failure.*

For safety reasons, when a failure occurs the driver is warned about the failure and the headlights are set to their normal intensity (by setting the `Headlights.lightLevel` variable to ON).

The feature's actions are commands to the headlights to change their lighting levels. To model the actions, you can use the following assignments and undefined functions:

- `Headlights.lightLevel := ON`
- `setHeadlightIntensity(targetIntensity)`
- `Headlights.lightLevel := OFF`

To calculate the desired intensity of the vehicle's headlights the undefined function *calculateIntensity(preceding or oncoming vehicle, Environment.ambientLightLevel)* can be used.

The following use case description may be of assistance in modelling this feature:

Driver	AH	Road	Environment
1. Driver sets the AH knob to AUTO.			
	2. Confirm that ACC is Active.		
	3. Poll sensors.		
		4. Check Vehicle.preceding and Vehicle.oncoming values.	4. Check Environment. ambientLightLevel value.
	5. Adjust headlight luminosity. Goto step 3.		
Alternative 1: ACC not Active			
	2. If ACC is not Active, <i>Use case ends.</i>		
Alternative 2: Driver turns headlights off			
n(3 - 5). Driver sets the AH knob to OFF.			
	n+1. Deactivate AH. Turn off headlights. <i>Use case ends.</i>		
Alternative 3: Driver turns headlights on			
n(3 - 5). Driver sets the AH knob to ON.			
	n+1. Deactivate AH. Turn on headlights. <i>Use case ends.</i>		
Exception 1: Light-detecting sensor fails			
			4. Environment. ambientLightLevel value outside range.
	5. Warn driver.		
	6. Deactivate AH. Turn on headlights. <i>Use case ends.</i>		
Exception 2: Object-detecting sensor fails			
		4. Vehicle.preceding or Vehicle.oncoming is <i>Unknown.</i>	
	5. Warn driver.		
	6. Deactivate AH. Turn on headlights. <i>Use case ends.</i>		

C.2 User Study - Control Group

This is the version of the study that was provided to the Control group of the user study.

Feature Specification Study

Thank you for participating in our study. To help protect the integrity of our results, we ask that you do not share this questionnaire with other students or collaborate on the tasks provided below. We also ask that you do not talk to anyone about the questionnaire or the tutorial during or after your completion of it - until all participants have completed the study. You are free to refer to tutorial materials while completing the study. The domain model provided on the next page is identical to the one in the tutorial.

You are free to omit answers to any of the following questions or tasks. If you have any questions or concerns, please contact us at cbocovic@uwaterloo.ca or d4dietri@uwaterloo.ca.

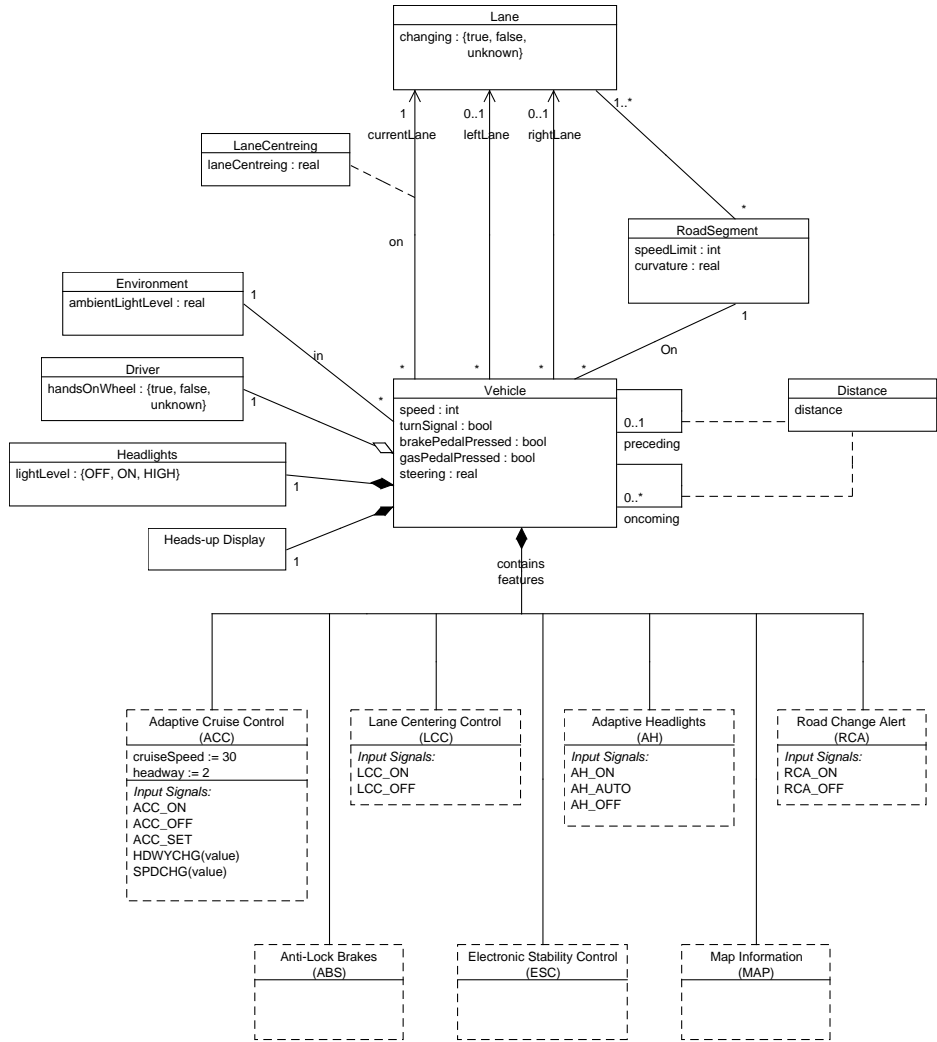
1 Background Questions

Please answer the following questions to the best of your ability. The questions in this section concern your previous experience in modelling software requirements.

1. Do you have previous experience with requirements modelling or state-machine modelling?
If so, briefly state the types and levels of experience (include notations/methods/tools used, length of time used, and whether your experience is from coursework or industrial experience.)

2. On a scale of 1 to 5, express your level of comfort with UML State Machines or statecharts (1 = never heard of them, 2 = heard of them and have looked at some models, 3 = used the notation in the past but do not recall a lot of details, 4 = can probably sketch a model, 5 = have good knowledge of them).

3. What is your experience with modelling automotive features (if any)?



2 Reading Assignment

We want to collect information on how long it takes to comprehend a model that was created by someone else. This includes the time devoted to examining and understanding the model as well as the time spent answering questions about it. Please review the model below and answer the following questions. Record the time at which you start the task and the time you complete it. If you take breaks or do the section in several stages, please record each start and stop time.

Below is a model of an Adaptive Cruise Control (ACC) feature for automobiles. ACC is a more advanced version of a basic Cruise Control feature. In addition to maintaining the vehicle's speed at a constant driver-set cruising speed, it also maintains a safe distance to the vehicle ahead. Additionally, ACC will react to traffic conditions by deactivating if an upcoming object gets too close. The driver activates ACC by pressing a button located near the steering wheel of the vehicle. ACC will deactivate automatically in the event that the vehicle's speed drops below 30 km/h or gets too close to an upcoming object. The driver can deactivate ACC manually by pressing the same switch that used to activate the feature. While activated, ACC can be overridden by pressing the gas pedal or the brake pedal. This temporarily suspends its controlling behaviour. In the event that hardware related to the operation of this feature fails, the feature itself will fail.

Start Time:
Stop Time:

After each question, please indicate your level of confidence in your answer.

1. List all of the *user actions* that the user performs as part of the process to *activate* the feature.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

2. List all of the *environmental conditions* that must hold for the feature to *remain active*, once the feature is active and controlling the vehicle's speed.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

3. Circle the features that the ACC feature refers to, and list all of the information that ACC obtains from these features.

- ACC
- LCC
- AH
- RCA
- ESC
- ABS
- MAP

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

- List all of the states in which the feature can directly affect the behaviour of the automobile.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

- Describe how ACC behaves on the first detection of a failure.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

- What is the name of the initial state(s) of the ACC feature (when the car is first powered on)?

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

Undefined Functions

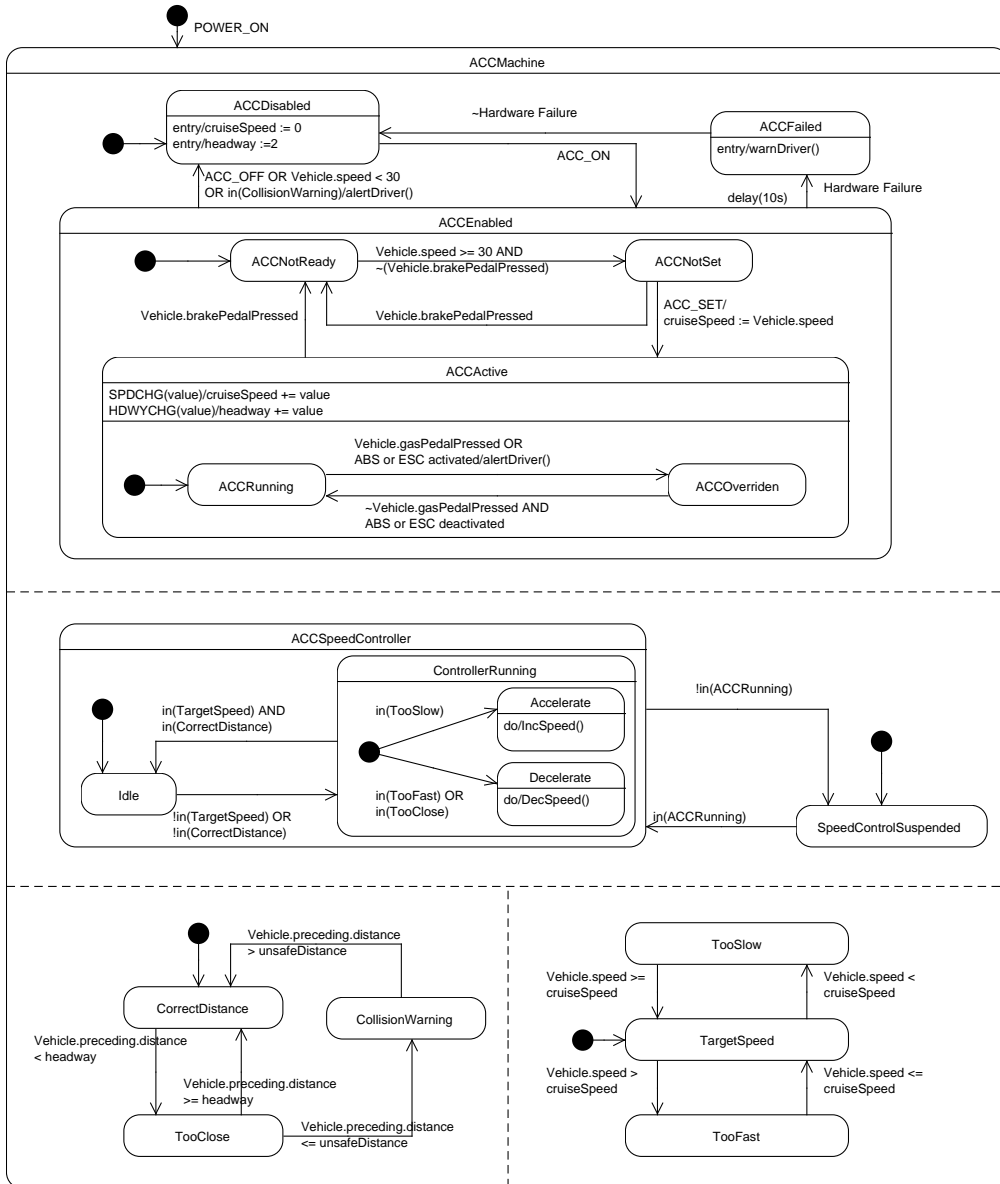
SPDCHG(value) - allows driver to increment or decrement the targetSpeed variable
 HDWYCHG(value) - allows driver to increment or decrement the headway variable.
 AlertDriver() - sends an alert to the driver of the vehicle
 WarnDriver() - warns the driver of the vehicle of an impending collision

Other features

ABS - Anti-lock Braking System
 ACC cannot modify the vehicle's while this is controlling the vehicle.

ESC - Electronic Stability Control
 ACC cannot modify the vehicle's behavior while this feature is controlling the vehicle.

State Machine Variables
 unsafeDistance := 0.5



3 Writing Assignment

This section involves writing a state-machine model of an automotive feature. After reading the description, please record the time at which you start the modelling task and please record the time when you complete the task. If you take breaks or work on this section of the study in several stages, please record each start and stop time. Spend no more than one hour on the modelling part of the task (Set an alarm for yourself).

The task is to write a state-machine model for an automotive feature called Lane Centring Control (LCC). When active, LCC attempts to automatically steer the vehicle to stay in the current lane. LCC can be activated only if the Adaptive Cruise Control (ACC) feature is already active. If ACC is active, then the driver can turn the LCC feature ON and OFF by pressing a button on the steering wheel.

LCC is automatically deactivated if ACC is no longer active, the driver removes his or her hands from the steering wheel for too long, the vehicle's speed becomes less than 60km/h, or if the ACC feature detects an anticipated collision. Additional inputs to the system are:

- RoadSegment.curvature: real - *This environmental reflects the upcoming curve in the road, if any. The condition is sensed by the GPS feature. If GPS fails, then the LCC feature also fails. If the GPS feature and sensor recover from a failure, LCC will also recover.*
- Vehicle.currentLane.laneCentring: [-1..1] - *This environmental condition indicates the degree to which the vehicle is centred in the lane. It is sensed by the CameraSensor feature. The target value for this environmental condition is 0, meaning the vehicle is centred between the left and right markings on the road. A positive value indicates that the vehicle is diverging to the right, and a value of 1 or greater indicates that the car is crossing the right lane marker. Negative values similarly indicate that the vehicle is diverging to the left. Values out of range indicate that the vehicle has diverged too far out of the lane to be automatically re-centred or that the CameraSensor has failed. In both of these cases, LCC fails. When the value reenters the range -1..1, the LCC feature recovers from failure.*
- Vehicle.turnSignal: boolean - *This environmental condition is true when the driver activates the turn signal and false otherwise. When the turn signal is activated, LCC should remain enabled but not have control of the vehicle.*
- Driver.handsOnWheel: boolean - *This environmental condition is true when the driver's hands are currently on the steering wheel and false when the driver has taken his or her hands off the steering wheel.*
- Vehicle.speed: [0..300] - *The vehicle's speed is sensed by the speedometer. If the speedometer is not able to determine the speed, the value will be -1 and LCC fails. When the speedometer recovers from a failure, the LCC feature also recovers.*

For safety reasons, LCC is deactivated whenever a failure occurs. The feature must then wait for any failure conditions to be removed before recovering from a failed state.

LCC's outputs are commands that attempt to keep the vehicle in the lane and warnings to the driver if the feature fails or deactivates. You can use the following undefined functions in your model:

- `calculateSteeringValue(Vehicle.currentLane.laneMarking, RoadSegment.curvature)` - *calculates the exact value for how much the vehicle should veer to be recentred in the lane.*
- `warnDriver()` *alerts the driver of a change in the feature's state (e.g. deactivation)*

The following use case description may be helpful.

Driver	LCC	SteeringWheel	GPS	CameraSensor	Speedometer
1. Driver activates LCC by pressing button					
	2. Confirms that ACC is active				
	3. Poll sensors				
		4. Determine whether driver's hands are off steering wheel.	4. Determine road curvature	4. Determine vehicle's distance to lane markings	4. Determine current speed
		5. If hands are off wheel, increment time counter, else reset time counter. Go to step 3.	5. If road is straight, go to step 3.	5. If centered in lane, go to step 3.	5. If speed is above 60 km/h, go to step 3.
Alternative 1: Driver uses turn signal					
n:(3-5). Driver uses the turn signal					
	n+1.Suspends LCC control.				
n+2. Driver turns off turn signal					
	n+3. LCC control regained. Go to step 3.				
Alternative 2: Driver's hands are off the steering wheel for too long					
		4. Counter exceeds threshold			
	5. Warn driver.				
	6. Deactivate LCC. <i>Use case ends</i>				
Alternative 3:ACC is not active					
	2. Confirms ACC inactive				
	3. Deactivate LCC. <i>Use case ends.</i>				

Alternative 4: Driver deactivates LCC					
n:(3-5). Driver deactivates LCC by pressing button					
	n+1. Deactivate LCC. Use case ends				
Alternative 5: Lane Markings suggest necessary steering					
				4. Vehicle is not in center of lane	
	5. Steer to the right or left accordingly				
	6. Go to step 3.				
Alternative 6: GPS reports sharp bend					
			4. Detects a sharp bend ahead		
	5. Warns driver.				
	6. Deactivate LCC. Use case ends				
Alternative 7: ACC is deactivated					
	n. Detects ACC deactivation				
	n+1. Warn driver.				
	n+2. Deactivate LCC. Use case ends				
Alternative 8: Vehicle speed less than 60km/h.					
					4. Detects vehicle speed less than 60 km/h
	5. Warns driver				
	6. Deactivate LCC. Use case ends				
Exception 1: Hardware failure.					
			4. Reports no GPS data OR Reports no lane marking OR Reports no speed data		
	5. Warn driver.				
	6. LCC fails.				

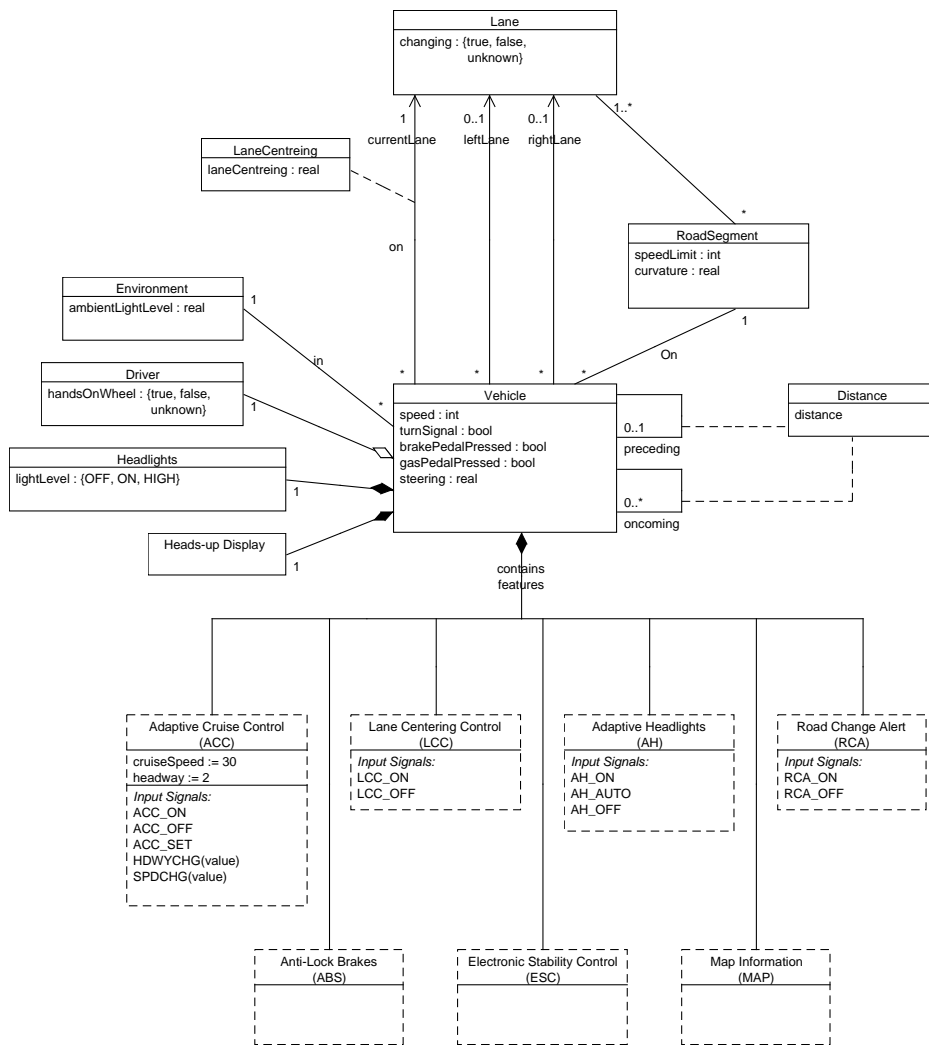
Start Time:
Stop Time:

Please write your model on the page provided and state how confident you are in your model. It may be difficult to create an elegant model in such a short time and without the aid of modelling tools. We are interested in the functionality of the model and not so much in how neat it is or how efficient it is. Remember, this exercise should take at most an hour to complete.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

C.3 User Study - Pattern Group

This is the version of the study that was provided to the Pattern group of the user study.



2 Reading Assignment

We want to collect information on how long it takes to comprehend a model that was created by someone else. This includes the time devoted to examining and understanding the model as well as the time spent answering questions about it. Please review the model below and answer the following questions. Record the time at which you start the task and the time you complete it. If you take breaks or do the section in several stages, please record each start and stop time.

Below is a model of an Adaptive Cruise Control (ACC) feature for automobiles. ACC is a more advanced version of a basic Cruise Control feature. In addition to maintaining the vehicle's speed at a constant driver-set cruising speed, it also maintains a safe distance to the vehicle ahead. Additionally, ACC will react to traffic conditions by deactivating if an upcoming object gets too close. The driver activates ACC by pressing a button located near the steering wheel of the vehicle. ACC will deactivate automatically in the event that the vehicle's speed drops below 30 km/h or gets too close to an upcoming object. The driver can deactivate ACC manually by pressing the same switch that used to activate the feature. While activated, ACC can be overridden by pressing the gas pedal or the brake pedal. This temporarily suspends its controlling behaviour. In the event that hardware related to the operation of this feature fails, the feature itself will fail.

Start Time:
Stop Time:

After each question, please indicate your level of confidence in your answer.

1. List all of the *user actions* that the user performs as part of the process to *activate* the feature.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

2. List all of the *environmental conditions* that must hold for the feature to *remain active*, once the feature is active and controlling the vehicle's speed.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

3. Circle the features that the ACC feature refers to, and list all of the information that ACC obtains from these features.

- ACC
- LCC
- AH
- RCA
- ESC
- ABS
- MAP

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

- List all of the states in which the feature can directly affect the behaviour of the automobile.

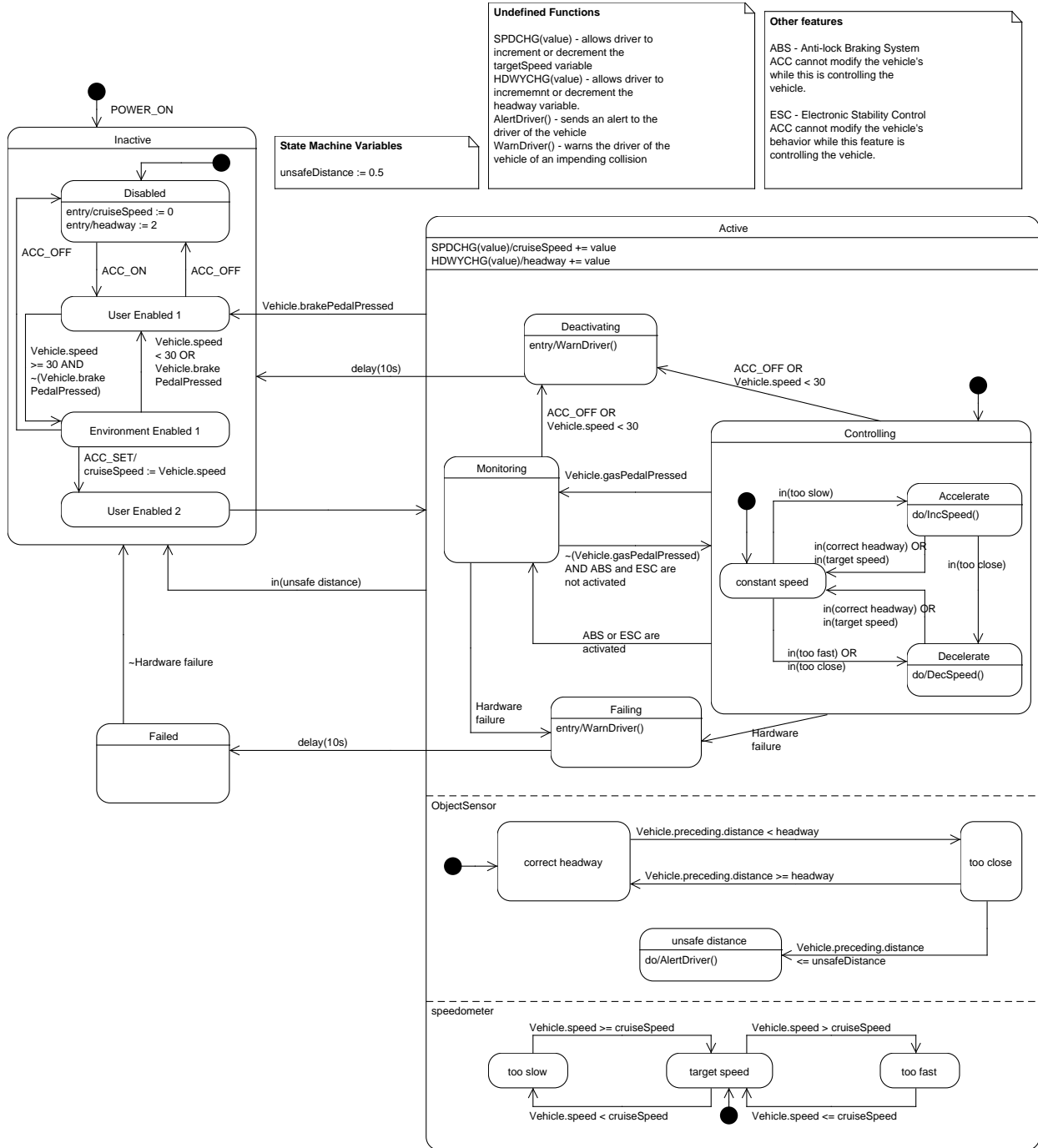
Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

- Describe how ACC behaves on the first detection of a failure.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

- What is the name of the initial state(s) of the ACC feature (when the car is first powered on)?

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)



3 Writing Assignment

This section involves writing a state-machine model of an automotive feature. After reading the description, please record the time at which you start the modelling task and please record the time when you complete the task. If you take breaks or work on this section of the study in several stages, please record each start and stop time. Spend no more than one hour on the modelling part of the task (Set an alarm for yourself).

The task is to write a state-machine model for an automotive feature called Lane Centring Control (LCC). When active, LCC attempts to automatically steer the vehicle to stay in the current lane. LCC can be activated only if the Adaptive Cruise Control (ACC) feature is already active. If ACC is active, then the driver can turn the LCC feature ON and OFF by pressing a button on the steering wheel.

LCC is automatically deactivated if ACC is no longer active, the driver removes his or her hands from the steering wheel for too long, the vehicle's speed becomes less than 60km/h, or if the ACC feature detects an anticipated collision. Additional inputs to the system are:

- RoadSegment.curvature: real - *This environmental reflects the upcoming curve in the road, if any. The condition is sensed by the GPS feature. If GPS fails, then the LCC feature also fails. If the GPS feature and sensor recover from a failure, LCC will also recover.*
- Vehicle.currentLane.laneCentring: [-1..1] - *This environmental condition indicates the degree to which the vehicle is centred in the lane. It is sensed by the CameraSensor feature. The target value for this environmental condition is 0, meaning the vehicle is centred between the left and right markings on the road. A positive value indicates that the vehicle is diverging to the right, and a value of 1 or greater indicates that the car is crossing the right lane marker. Negative values similarly indicate that the vehicle is diverging to the left. Values out of range indicate that the vehicle has diverged too far out of the lane to be automatically recentred or that the CameraSensor has failed. In both of these cases, LCC fails. When the value reenters the range -1..1, the LCC feature recovers from failure.*
- Vehicle.turnSignal: boolean - *This environmental condition is true when the driver activates the turn signal and false otherwise. When the turn signal is activated, LCC should remain enabled but not have control of the vehicle.*
- Driver.handsOnWheel: boolean - *This environmental condition is true when the driver's hands are currently on the steering wheel and false when the driver has taken his or her hands off the steering wheel.*
- Vehicle.speed: [0..300] - *The vehicle's speed is sensed by the speedometer. If the speedometer is not able to determine the speed, the value will be -1 and LCC fails. When the speedometer recovers from a failure, the LCC feature also recovers.*

For safety reasons, LCC is deactivated whenever a failure occurs. The feature must then wait for any failure conditions to be removed before recovering from a failed state.

LCC's outputs are commands that attempt to keep the vehicle in the lane and warnings to the driver if the feature fails or deactivates. You can use the following undefined functions in your model:

- `calculateSteeringValue(Vehicle.currentLane.laneMarking, RoadSegment.curvature)` - *calculates the exact value for how much the vehicle should veer to be recentred in the lane.*
- `warnDriver()` *alerts the driver of a change in the feature's state (e.g. deactivation)*

The following use case description may be helpful.

Driver	LCC	SteeringWheel	GPS	CameraSensor	Speedometer
1. Driver activates LCC by pressing button					
	2. Confirms that ACC is active				
	3. Poll sensors				
		4. Determine whether driver's hands are off steering wheel.	4. Determine road curvature	4. Determine vehicle's distance to lane markings	4. Determine current speed
		5. If hands are off wheel, increment time counter, else reset time counter. Go to step 3.	5. If road is straight, go to step 3.	5. If centered in lane, go to step 3.	5. If speed is above 60 km/h, go to step 3.
Alternative 1: Driver uses turn signal					
n:(3-5). Driver uses the turn signal					
	n+1.Suspends LCC control.				
n+2. Driver turns off turn signal					
	n+3. LCC control regained. Go to step 3.				
Alternative 2: Driver's hands are off the steering wheel for too long					
		4. Counter exceeds threshold			
	5. Warn driver.				
	6. Deactivate LCC. <i>Use case ends</i>				
Alternative 3:ACC is not active					
	2. Confirms ACC inactive				
	3. Deactivate LCC. <i>Use case ends.</i>				

Alternative 4: Driver deactivates LCC					
n:(3-5). Driver deactivates LCC by pressing button					
	n+1. Deactivate LCC. Use case ends				
Alternative 5: Lane Markings suggest necessary steering					
				4. Vehicle is not in center of lane	
	5. Steer to the right or left accordingly				
	6. Go to step 3.				
Alternative 6: GPS reports sharp bend					
			4. Detects a sharp bend ahead		
	5. Warns driver.				
	6. Deactivate LCC. Use case ends				
Alternative 7: ACC is deactivated					
	n. Detects ACC deactivation				
	n+1. Warn driver.				
	n+2. Deactivate LCC. Use case ends				
Alternative 8: Vehicle speed less than 60km/h.					
					4. Detects vehicle speed less than 60 km/h
	5. Warns driver				
	6. Deactivate LCC. Use case ends				
Exception 1: Hardware failure.					
			4. Reports no GPS data OR Reports no lane marking OR Reports no speed data		
	5. Warn driver.				
	6. LCC fails.				

Start Time:
Stop Time:

Please write your model on the page provided and state how confident you are in your model. It may be difficult to create an elegant model in such a short time and without the aid of modelling tools. We are interested in the functionality of the model and not so much in how neat it is or how efficient it is. Remember, this exercise should take at most an hour to complete.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

C.4 User Study - Pattern+Interface Group

This is the version of the study that was provided to the Pattern+Interface group of the user study.

Feature Specification Study

Thank you for participating in our study. To help protect the integrity of our results, we ask that you do not share this questionnaire with other students or collaborate on the tasks provided below. We also ask that you do not talk to anyone about the questionnaire or the tutorial during or after your completion of it - until all participants have completed the study. You are free to refer to tutorial materials while completing the study. The domain model provided on the next page is identical to the one in the tutorial.

You are free to omit answers to any of the following questions or tasks. If you have any questions or concerns, please contact us at cbocovic@uwaterloo.ca or d4dietri@uwaterloo.ca.

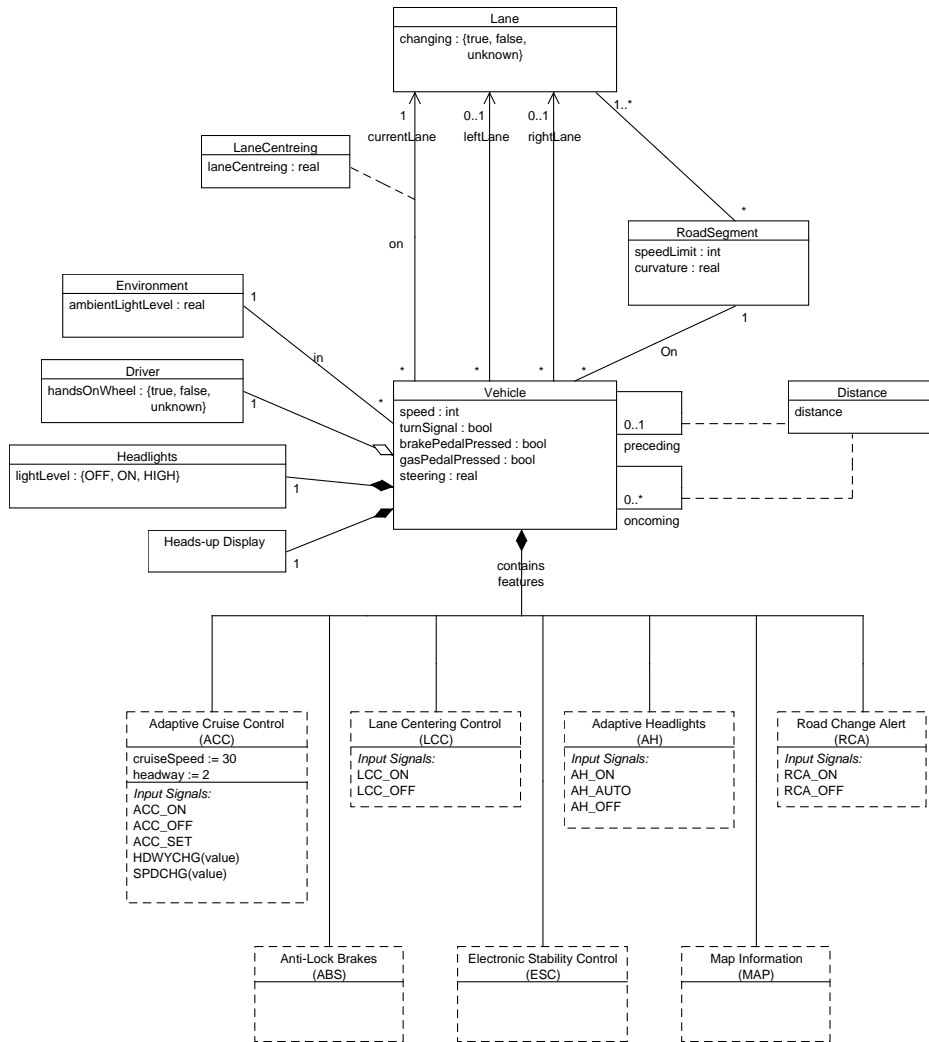
1 Background Questions

Please answer the following questions to the best of your ability. The questions in this section concern your previous experience in modelling software requirements.

1. Do you have previous experience with requirements modelling or state-machine modelling?
If so, briefly state the types and levels of experience (include notations/methods/tools used, length of time used, and whether your experience is from coursework or industrial experience.)

2. On a scale of 1 to 5, express your level of comfort with UML State Machines or statecharts (1 = never heard of them, 2 = heard of them and have looked at some models, 3 = used the notation in the past but do not recall a lot of details, 4 = can probably sketch a model, 5 = have good knowledge of them).

3. What is your experience with modelling automotive features (if any)?



2 Reading Assignment

We want to collect information on how long it takes to comprehend a model that was created by someone else. This includes the time devoted to examining and understanding the model as well as the time spent answering questions about it. Please review the model below and answer the following questions. Record the time at which you start the task and the time you complete it. If you take breaks or do the section in several stages, please record each start and stop time.

Below is a model of an Adaptive Cruise Control (ACC) feature for automobiles. ACC is a more advanced version of a basic Cruise Control feature. In addition to maintaining the vehicle's speed at a constant driver-set cruising speed, it also maintains a safe distance to the vehicle ahead. Additionally, ACC will react to traffic conditions by deactivating if an upcoming object gets too close. The driver activates ACC by pressing a button located near the steering wheel of the vehicle. ACC will deactivate automatically in the event that the vehicle's speed drops below 30 km/h or gets too close to an upcoming object. The driver can deactivate ACC manually by pressing the same switch that used to activate the feature. While activated, ACC can be overridden by pressing the gas pedal or the brake pedal. This temporarily suspends its controlling behaviour. In the event that hardware related to the operation of this feature fails, the feature itself will fail.

Start Time:
Stop Time:

After each question, please indicate your level of confidence in your answer.

1. List all of the *user actions* that the user performs as part of the process to *activate* the feature.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

2. List all of the *environmental conditions* that must hold for the feature to *remain active*, once the feature is active and controlling the vehicle's speed.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

3. Circle the features that the ACC feature refers to, and list all of the information that ACC obtains from these features.

- ACC
- LCC
- AH
- RCA
- ESC
- ABS
- MAP

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

- List all of the states in which the feature can directly affect the behaviour of the automobile.

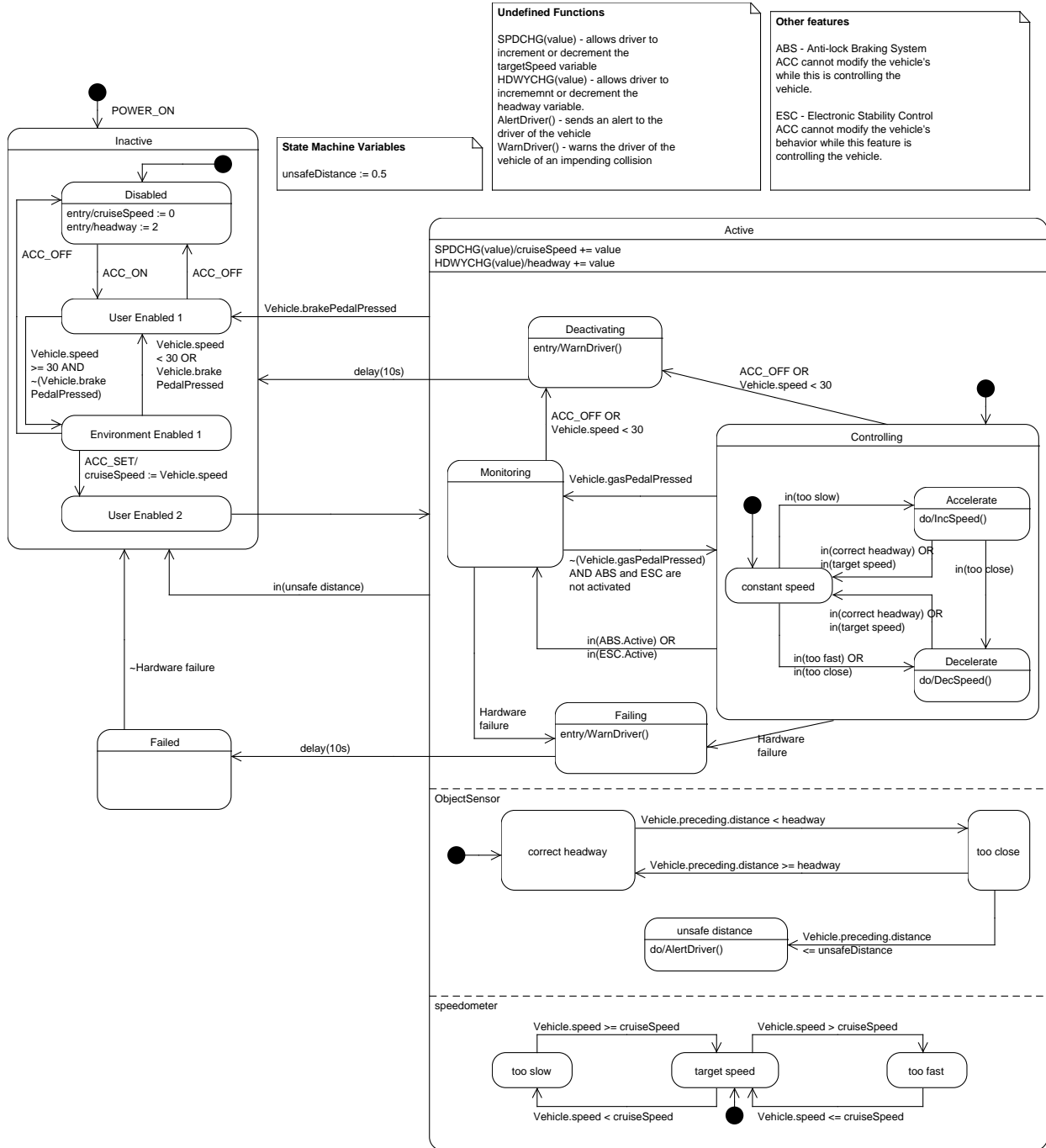
Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

- Describe how ACC behaves on the first detection of a failure.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)

- What is the name of the initial state(s) of the ACC feature (when the car is first powered on)?

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)



3 Writing Assignment

This section involves writing a state-machine model of an automotive feature. After reading the description, please record the time at which you start the modelling task and please record the time when you complete the task. If you take breaks or work on this section of the study in several stages, please record each start and stop time. Spend no more than one hour on the modelling part of the task (Set an alarm for yourself).

The task is to write a state-machine model for an automotive feature called Lane Centring Control (LCC). When active, LCC attempts to automatically steer the vehicle to stay in the current lane. LCC can be activated only if the Adaptive Cruise Control (ACC) feature is already active. If ACC is active, then the driver can turn the LCC feature ON and OFF by pressing a button on the steering wheel.

LCC is automatically deactivated if ACC is no longer active, the driver removes his or her hands from the steering wheel for too long, the vehicle's speed becomes less than 60km/h, or if the ACC feature detects an anticipated collision. Additional inputs to the system are:

- RoadSegment.curvature: real - *This environmental reflects the upcoming curve in the road, if any. The condition is sensed by the GPS feature. If GPS fails, then the LCC feature also fails. If the GPS feature and sensor recover from a failure, LCC will also recover.*
- Vehicle.currentLane.laneCentring: [-1..1] - *This environmental condition indicates the degree to which the vehicle is centred in the lane. It is sensed by the CameraSensor feature. The target value for this environmental condition is 0, meaning the vehicle is centred between the left and right markings on the road. A positive value indicates that the vehicle is diverging to the right, and a value of 1 or greater indicates that the car is crossing the right lane marker. Negative values similarly indicate that the vehicle is diverging to the left. Values out of range indicate that the vehicle has diverged too far out of the lane to be automatically re-centred or that the CameraSensor has failed. In both of these cases, LCC fails. When the value reenters the range -1..1, the LCC feature recovers from failure.*
- Vehicle.turnSignal: boolean - *This environmental condition is true when the driver activates the turn signal and false otherwise. When the turn signal is activated, LCC should remain enabled but not have control of the vehicle.*
- Driver.handsOnWheel: boolean - *This environmental condition is true when the driver's hands are currently on the steering wheel and false when the driver has taken his or her hands off the steering wheel.*
- Vehicle.speed: [0..300] - *The vehicle's speed is sensed by the speedometer. If the speedometer is not able to determine the speed, the value will be -1 and LCC fails. When the speedometer recovers from a failure, the LCC feature also recovers.*

For safety reasons, LCC is deactivated whenever a failure occurs. The feature must then wait for any failure conditions to be removed before recovering from a failed state.

LCC's outputs are commands that attempt to keep the vehicle in the lane and warnings to the driver if the feature fails or deactivates. You can use the following undefined functions in your model:

- `calculateSteeringValue(Vehicle.currentLane.laneMarking, RoadSegment.curvature)` - *calculates the exact value for how much the vehicle should veer to be recentred in the lane.*
- `warnDriver()` *alerts the driver of a change in the feature's state (e.g. deactivation)*

The following use case description may be helpful.

Driver	LCC	SteeringWheel	GPS	CameraSensor	Speedometer
1. Driver activates LCC by pressing button					
	2. Confirms that ACC is active				
	3. Poll sensors				
		4. Determine whether driver's hands are off steering wheel.	4. Determine road curvature	4. Determine vehicle's distance to lane markings	4. Determine current speed
		5. If hands are off wheel, increment time counter, else reset time counter. Go to step 3.	5. If road is straight, go to step 3.	5. If centered in lane, go to step 3.	5. If speed is above 60 km/h, go to step 3.
Alternative 1: Driver uses turn signal					
n:(3-5). Driver uses the turn signal					
	n+1.Suspends LCC control.				
n+2. Driver turns off turn signal					
	n+3. LCC control regained. Go to step 3.				
Alternative 2: Driver's hands are off the steering wheel for too long					
		4. Counter exceeds threshold			
	5. Warn driver.				
	6. Deactivate LCC. <i>Use case ends</i>				
Alternative 3:ACC is not active					
	2. Confirms ACC inactive				
	3. Deactivate LCC. <i>Use case ends.</i>				

Alternative 4: Driver deactivates LCC					
n:(3-5). Driver deactivates LCC by pressing button					
	n+1. Deactivate LCC Use case ends				
Alternative 5: Lane Markings suggest necessary steering					
				4. Vehicle is not in center of lane	
	5. Steer to the right or left accordingly				
	6. Go to step 3.				
Alternative 6: GPS reports sharp bend					
			4. Detects a sharp bend ahead		
	5. Warns driver.				
	6. Deactivate LCC. Use case ends				
Alternative 7: ACC is deactivated					
	n. Detects ACC deactivation				
	n+1. Warn driver.				
	n+2. Deactivate LCC. Use case ends				
Alternative 8: Vehicle speed less than 60km/h.					
					4. Detects vehicle speed less than 60 km/h
	5. Warns driver				
	6. Deactivate LCC. Use case ends				
Exception 1: Hardware failure.					
			4. Reports no GPS data OR Reports no lane marking OR Reports no speed data		
	5. Warn driver.				
	6. LCC fails.				

Start Time:
Stop Time:

Please write your model on the page provided and state how confident you are in your model. It may be difficult to create an elegant model in such a short time and without the aid of modelling tools. We are interested in the functionality of the model and not so much in how neat it is or how efficient it is. Remember, this exercise should take at most an hour to complete.

Confidence level (circle one): (Guessed, at least 50% correct, 50%-75% correct, 75%-90% correct, 90%-100% correct)