Detection of Feature Interactions in Automotive Active Safety Features

by

Alma L. Juarez Dominguez

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

With the introduction of software into cars, many functions are now realized with reduced cost, weight and energy. The development of these software systems is done in a distributed manner independently by suppliers, following the traditional approach of the automotive industry, while the car maker takes care of the integration. However, the integration can lead to unexpected and unintended interactions among software systems, a phenomena regarded as *feature interaction.* This dissertation addresses the problem of the automatic detection of feature interactions for automotive active safety features. Active safety features control the vehicle's motion control systems independently from the driver's request, with the intention of increasing passengers' safety (*e.g.,* by applying hard braking in the case of an identified imminent collision), but their unintended interactions could instead endanger the passengers (*e.g.,* simultaneous throttle increase and sharp narrow steering, causing the vehicle to roll over). My method decomposes the problem into three parts: (I) creation of a definition of feature interactions based on the set of actuators and domain expert knowledge; (II) translation of automotive active safety features designed using a subset of MATLAB's STATEFLOW into the input language of the model checker SMV; (III) analysis using model checking at design time to detect a representation of all feature interactions based on partitioning the counterexamples into equivalence classes. The key novel characteristic of my work is exploiting domain-specific information about the feature interaction problem and the structure of the model to produce a method that finds a representation of all different feature interactions for automotive active safety features at design time.

My method is validated by a case study with the set of non-proprietary automotive feature design models I created. The method generates a set of counterexamples that represent the whole set of feature interactions in the case study. By showing only a set of representative feature interaction cases, the information is concise and useful for feature designers. Moreover, by generating these results from feature models designed in MATLAB's STATEFLOW translated into SMV models, the feature designers can trace the counterexamples generated by SMV and understand the results in terms of the STATEFLOW model. I believe that my results and techniques will have relevance to the solution of the feature interaction problem in other cyber-physical systems, and have a direct impact in assessing the safety of automotive systems.

Acknowledgements

First, I would like to thank my supervisor, Professor Nancy A. Day, for her guidance, as well as her continuous support and inspiration. This thesis is better because of her insightful comments and advice.

I also wish to thank the members of my examining committee, Professors Joanne M. Atlee, Tim Brecht and John G. Thistle, for taking the time to read my thesis and for the comments they provided. Special thanks to Professor Mats P.E. Heimdahl for being on my committee. His valuable feedback and suggestions have given me a fresh perspective on my research.

Funding for my research was partially provided by General Motors (GM) Canada, Critical Systems Labs Inc., the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Cheriton School of Computer Science at the University of Waterloo. My results were shaped by the experience I gained while collaborating with members of the General Motors Research and Development team, particularly I would like to recognize Rami Debouk, Barbara Czerny and Joseph D'Ambrosio. A special recognition to Jeffrey J. Joyce, president of Critical Systems Labs Inc. for his supervision during my industrial collaboration, and also for his support and insightful advice.

To all my friends in Waterloo and around the world, Alex Hudek, Angelica Alvarez, Arturo Santillan, Catherine Donnelly, Cory Kapser, Dasha Hajducek, David Evans, Elodie Fourquet, Ioana Burcea, Irene Pivotto, Joanna Ziembicka, Joyce Nieuwesteeg, Judith Toms, Leticia Ramirez, Mike Patterson, Rob Warren, Sonia Waharte, Wendy Rush, thanks for helping me through the rough patches and for continuously enriching my life no matter where they are! To all the families that have opened their homes and hearts to me, Aagaard-Day, Cowan, Joyce, Richards, I cannot thank them enough for their kindness, for all I have learned from them and the enjoyable times I have spent. Thanks also go to the members of the WatForm lab, past and present, for providing me with valuable discussions and interesting conversations. My recognition and appreciation to the members of Latinas in Computing for their mentorship. And for all the members of #beer, thanks for letting me be part of the club and remind me that I am not alone in this crusade.

Without my family, I could not be who I am today. I would like to thank my parents, Alma Domínguez and Rafael Juárez, for their outstanding support, for always encouraging me to pursue my dreams, but above all for their unconditional love. To my siblings, Karen and Rafael, thank you for showing me how to laugh and live my life intensely. My heart also goes to the Koulis family for their love, support and inspiration.

To my husband, Theo Koulis, thanks for always being there for me. Your patience, support and encouragement help me complete this work, demonstrating that together we can do anything. Your love makes my life special.

Dedication

To my family, for the unconditional love, constant support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"It takes dozens of microprocessors running 100 million lines of code to get a premium car out of the driveway, and this software is only going to get more complex." [50]

The introduction of software into cars first occurred in 1977 to control the electronic spark timing [22]. Nowadays, the amount of software in a premium-class vehicle is close to 100 million lines of code, packed within 70-100 microprocessor-based electronic control units (ECUs) networked throughout the body of the car [50]. In 2005, a study by the Center for Automotive Research, PriceWaterhouseCoopers, VDA and the city of Leipzig reported that software would account for 40% of a car's total value by 2010 [166], which is a continuing trend. Software is advantageous because it enables car makers and suppliers to realize functions with reduced costs, weight and energy, while also making it possible to personalize systems to particular customer's needs [34, 149]. Thus, software helps control functions like the volume of the radio, gas consumption and even the motion control systems (*e.g.,* braking). However, the advantages of software come with challenges, such as avoiding glitches in the software. Software problems have put some automotive companies in the spotlight over the years: In 2003, software problems in the BMW 7-series made engines shut down at highway speeds [139]; In 2005, Mazda's software controlling the fuel injection was flooding the engine in its RX-8 sports car [177]; In 2010, Toyota reported problems with the braking control software used in its Prius line [134]; In 2011, General Motors had to fix a software system that prevents the defroster from clearing the windshield [155]. But not all the challenges disappear by eliminating local software glitches. The modern car has turned from an assembled device into an integrated system, and new challenges now arise from the integration of subsystems from different sources [35]. The integration can lead to unexpected and unintended interactions among subsystems, a phenomena regarded as *feature interaction* [31]. In this dissertation, I tackle the problem of detecting feature interactions in the automotive domain.

Traditionally, the automotive industry has been organized in a very modular manner, thanks to the hard work of mechanical engineers for over a hundred years to make the subsystems of cars independent. Usually, a car's parts are produced by a chain of suppliers, and only an estimated 25% of the product value is created by the car maker, which concentrates on the engine, assembly of the parts, design and marketing [149]. This division of labour has been highly successful because it allows the suppliers to keep the development costs low. However, with more software being part of the vehicle, there are two main changes in the automotive industry: (1) cars change from being purely electro-mechanical devices to being cyber-physical systems (CPS), in which embedded computers and networks monitor and control physical processes [121], and (2) the car maker's responsibility switches from assembly of parts to system integration. But what exactly are these software systems that need to be integrated?

In a premium-class vehicle, over 2000 'atomic' functions are related to software, which are then combined into about 270 'user' functions that a driver or passenger interacts with [34, 35]. When talking about a 'feature', the definition of this concept depends on the level of granularity at which people decide to reason and work. The main difference in definitions is the identification of a 'feature' either as a single function (such as an 'atomic' function or a requirement), or as "a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective" [170] (such as the concept of 'user' function used by Broy *et al.* [35]). In the automotive domain, one can interpret the definition of feature as "a service recognized by the driver" [104, 124], which identifies a set of functionality in one package. Some examples of features in an automobile are automatic door lock, pre-set seat adjustment, monitored tired pressure, and cruise control (CC).

Integration of software subsystems is challenging in itself, but in automotive systems the integration problem is worse because the development is geographically distributed and suppliers have great freedom in how they implement their solutions [35]. Often, the car maker only gets a black box specification of the suppliers' subsystems to integrate [149]. With the huge number of software related functions, and the challenge of distributed and independent development of subsystems, the phenomena of feature interaction[1] in the automotive domain is likely to appear. Feature interactions are not due to malfunctions in individual features, but they arise from the integration of features developed in isolation and whose goals potentially conflict. Therefore, the task of detecting and resolving unexpected and unintended interactions among subsystems must be a priority, and new techniques and tools are needed for this task.

The challenge of integration of automotive systems is also shaped by a recent trend, in which subsystems that used to be highly independent, and controlled only by the driver,

[1]In this dissertation, the term *feature interaction* is always meant to refer to unsafe interactions among subsystems, thus avoiding the confusion that arises when the term can be used to mean both intended and unintended interactions (as used in references such as Zave [192] and Bowen *et al.* [31]).

start interacting with each other. A compelling example of this kind of interaction is the central locking system [118]. This system integrates the functionality of locking and unlocking doors with the functionality related to comfort (*e.g.,* adjusting seats and mirrors based on the key used), with the functionality related to safety/security (*e.g.,* arming a security device while the car is locked and unlocking all doors in case of a crash), and with the functionality related to the Human-Vehicle Interface (HVI) (*e.g.,* using the lighting system to signal when the car is locked and unlocked). Because this new feature is composed of other features, all the interactions among the composed subsystems must be analyzed to ensure there are no feature interactions before the new packaged functionality is delivered. A commonly referred to example of feature interaction occurs between a safety and a security feature, both part of the central locking system [42]. The functionality of the safety feature is to unlock all the doors automatically when a collision is detected, thus allowing passengers to escape and paramedics to reach any injured people. The functionality of the security feature is to keep all the doors locked, thus securing the occupants and their valuables. Ideally, the safety feature must take precedence over the security feature by unlocking all the doors when an accident happens. But, if a parked car gets hit in the front intentionally by a thief, then the security feature is compromised by the safety feature.

Another type of subsystem that used to operate mostly independently and was activated based on driver input only are *motion control systems*, such as steering, braking and propulsion. Recently, the automotive industry began introducing new features that command its motion control systems (*e.g.,* Collision Avoidance (CA) commanding the braking), and these features then might direct the motion control systems to take actions independent of the driver input under some conditions [65]. Due to the introduction of these types of features, multiple control systems may activate simultaneously, based on commands received from one or more features, and this situation can have the potential to create safety-risk related issues *e.g.,* when the features simultaneously request an increase of throttle while also requesting a sharp narrow steering, the request could cause the vehicle to roll over [105].

So far, I have identified challenges that occur during the integration of features within a vehicle, or *intra-vehicle* features. In this thesis, I focus on the problem of detection of *intra-vehicle* feature interactions. However, features within a car can interact with external systems, *e.g.,* integrating infotainment content, communicating with other vehicles or communicating with roadside devices to obtain information regarding traffic, road condition, *etc.* [92, 182]. These external systems are regarded as *inter-vehicle* features, or more concretely, *vehicle-to-vehicle* and *vehicle-to-infrastructure* services [19]. This added functionality, although leading to the vision of self-driving cars [66, 185], comes with many other challenges to overcome, not only technological, such as security [186], but also social and legal [182].

1.1 Automotive Active Safety Features

This dissertation focuses on detection of interactions among features that command the motion control systems of automobiles because they could lead to safety risks. These features use information from sensors, cameras, as well as radar, and they are normally called "active safety features" [126, 5] since they request control of the motion control systems *independently from the driver's requests* when necessary, *e.g.,* requesting hard braking in the case of an identified imminent collision. These features are meant to help the passenger's safety by reacting when the driver is unaware of an unsafe situation. However, an inappropriate resolution to a feature interaction in an automotive system can have safety implications that could endanger the vehicle's occupants, such as allowing requests that make the car go out of control (*e.g.,* simultaneous braking and throttle increase).

Active safety features are realized as software functions within embedded components. An automotive embedded system is a **cyber-physical system** composed of a "cyber" part and a "physical" part. The "cyber" part is software components running on digital hardware that control the mechanical and electrical processes, which is the "physical" part. The cyber and physical components of an automotive embedded system are illustrated in Figure 1.1. The cyber components, *i.e.,* active safety features, receive information from their environment through sensors, and send commands to be executed by their environment through actuators. The environment of active safety features is the physical components, which are the mechanical and electrical processes of the vehicle that are part of the motion control systems (*i.e.,* throttle, brake and steering).

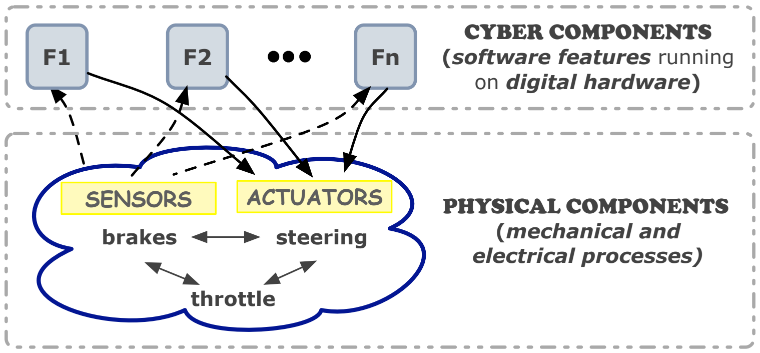

F1
F2
Fn
CYBER COMPONENTS
(software features running on digital hardware)
SENSORS
ACTUATORS
brakes
steering
throttle
PHYSICAL COMPONENTS
(mechanical and electrical processes)

Figure 1.1: Components of an automotive embedded system

Automotive active safety features usually run simultaneously and independently on different hardware with no direct communication between features because these features are developed in isolation by different suppliers or by different teams within the same company. Moreover, the features that are part of a vehicle might be selected at release or retail time, and thus, they should not rely on other features to be present. Indirect communication occurs through the mechanical processes since features share these processes as an environment and because of the closed-loop behaviour. In a closed-loop control system, a sensor monitors the system output (the car's speed) and feeds the data to a controller which adjusts the control (the throttle position) as necessary to maintain the desired system output (match the car's speed to the reference speed.) The output of the features change the environment, which is then read through the sensors and becomes input to the features at a later time. Automotive active safety features are typically modelled using MATLAB's STATEFLOW language.

Embedded systems are often used in safety-critical applications, thus there is a compelling need to ensure their reliability, correctness and safety. Reliability is the ability of a system to perform and maintain its functions in all circumstances, either routine or unexpected. Traditionally, reliability, correctness and safety checks are achieved by performing extensive testing and using techniques such as probabilistic reliability modelling [160] and Failure Modes and Effects Analysis (FMEA) [122]. While the use of FMEA in the design process helps us produce more reliable and safe products, it only focuses on individual features failure and does not help identify and avoid feature interactions.

As I have observed in practice[2], automotive domain experts meet to gather the scenarios in which they think that features under consideration would interact in an unsafe manner. Thus, they traditionally identify feature interactions by listing combinations of behaviours that they consider as potentially problematic. However, there is no way to guarantee that, following this traditional approach, the set of identified scenarios is complete. For a set of automotive active safety features, *feature interactions* arise from the activation of two or more features whose output requests to the actuators cause contradictory physical forces in the environment of the features (*e.g.,* simultaneous request of brake and throttle independent from the driver). While both actions may be correct according to the intended behaviour of each feature, their interaction is undesired. This view, although different from the traditional approach of identifying feature interactions in the automotive domain, will allow me to provide a definition that systematically detects this kind of feature interactions for active safety features.

1.2 The Feature Interaction Problem

The feature interaction problem has been studied extensively for more than two decades, mainly in telecommunications systems [154], but more recently also in internet applications and embedded systems. As an example, a feature interaction in the telecommunications domain occurs when user A subscribes (*i.e.,* pays for selected features) to features Call

[2] Observed during my visits to General Motors (GM) Research and Development as part of the requirements of my NSERC Industrial Postgraduate Scholarship.

Forwarding Unconditional (CFU) and Originating Call Screening (OCS), and CFU tries to forward a call while OCS intends to abort the same call, if the same number is in the forward and call screening lists.

Independent of the domain, the process to identify feature interactions is called **detection**. The process to minimize or eliminate the adverse effects of an interaction is called **resolution**. Another approach to eliminate feature interactions is called **avoidance**, which relies in guidelines or architectures to prevent interactions. However, complete domain analysis is required to have an efficient architecture in place. But completeness is rare as new types of features and technologies are added over time, and thus, detection approaches are often needed to gain the required knowledge [86]. My methods and tools concentrate on detection to acquire the knowledge of all different feature interactions among automotive active safety features before defining an appropriate avoidance or resolution approach. The approaches to deal with the feature interaction problem in telecommunications have been applied either **offline** (*i.e.,* while the feature is specified, designed and implemented), **online** (*i.e.,* while the feature is tested and after deployment), or **hybrid** (*i.e.,* combination of offline and online). Compared to the well-studied problem of feature interaction in the telecommunications domain, the feature interaction problem for active safety systems in the automotive domain has the following characteristics [104]:

1. In automotive systems, there can be multiple outputs from the features, all modifying the environment of the driver. In contrast, in a telecommunication system, the output to a user is a **route**, which connects users or manifests itself as tones or messages when a connection cannot be carried on (*e.g.,* busy tone, away message).
2. In automotive systems, the interactions appear in the physical part in the form of contradictory physical forces in the environment (*e.g.,* actuators for brakes and throttle conflict in their influence on speed). In contrast, in a telecommunication system, the feature interactions appear in the cyber part.
3. In an automotive system, the set of features is fixed at retail or release time, which makes an offline solution based on analysis of the designs appropriate. In contrast, a telecommunication system is more likely to need an online feature detection and resolution solution because of the incremental addition of features by a user.
4. In automotive systems there is only one copy of an active safety feature that can be active in a vehicle, making it possible to analyze these systems as if the features are invoked statically. In contrast, in a telecommunication system, multiple copies of the same feature may be invoked dynamically by different users during a call (*e.g.,* multiple uses of call waiting).
5. In an automotive system, the set of actuators and their range of possible values are known in advance. In contrast, in a telecommunication system we do not know the values that the route can take since subscription information can change dynamically and features can be customized by the users.

Given the characteristics of automotive systems, the feature interaction problem in this domain is bounded as compared to telecommunication systems, which makes offline model checking of feature designs a promising detection technique [42, 105]. Model checking is a powerful technique that searches exhaustively all behaviours of the system [57]. If a model does not satisfy a property describing a desired behaviour, a model checker produces a counterexample, which is a path of the model's behaviour that fails the property.

The feature interaction problem for automotive active safety features requires a comprehensive view of the problem by looking at **all** different feature interactions before deciding on a resolution strategy that reduces any safety risks for the passengers, as illustrated in the following example.

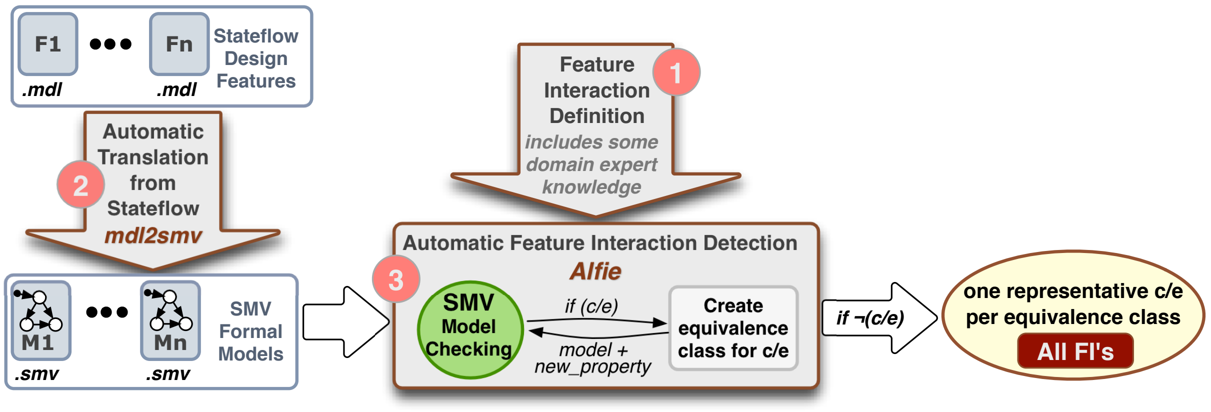
> ***Example 1.1:*** Consider two active safety features running concurrently. *Collision Avoidance (CA)* helps mitigate collisions while driving forward. *Emergency Vehicle Avoidance (EVA)* pulls the vehicle over when an emergency vehicle needs the road to be cleared. Using a model checker, a feature interaction can be detected if simultaneously CA requests hard braking while EVA requests soft braking. Based on this feature interaction, the feature precedence of CA over EVA could be chosen as a resolution strategy because applying hard braking is safer for the passengers to prevent a collision. However, given this feature precedence, the system would still reach an unsafe situation in the case that CA applies soft braking regardless of EVA's request of mid-force braking, which are allowed actions by both features. By looking at all different feature interactions, a different resolution strategy could be chosen.

Domain experts can determine an appropriate resolution scheme or provide a correction to the design features if during the formal analysis all feature interactions are found in the feature models.

Chapter 2 provides an overview of the different techniques, including model checking, that have been used in the literature to deal with the feature interaction problem in various domains. The following couple of approaches attempt to deal with the feature interaction problem in the automotive domain, but they do not solve the problem. The approach of Lochau and Goltz uses test case generation for feature interaction analysis, defining test cases whenever two features access (read or write) to a shared variable [124]. Even though their method is aimed to deal with STATEFLOW feature models, it does not match STATEFLOW's semantics. Moreover, as with other test case generation approaches, it does not discover all different feature interactions. The approach by D'Souza, Gopinathan *et al.* uses concepts of supervisory control theory to detect interactions, based on a notion of blocking supervisor conjunction [67]. However, not all different feature interactions might be identified and properly handled by the resolutions included in their features. I believe that this approach would benefit from using my method to recognize all classes of conflicts before applying their resolution strategy. To the best of my knowledge, my work is the first to propose a method to detect *all* different feature interactions in the automotive domain.

1.3 Thesis Overview

A high-level description of my feature interaction detection method and tools used at design time is illustrated in Figure 1.2. The rest of this section provides an overview of the three steps in my method. In this dissertation, **FI** is often used instead of "feature interaction".



F1
Fn
.mdl
.mdl
Stateflow Design Features
Automatic Translation from Stateflow
mdl2smv
2
Feature Interaction Definition
includes some domain expert knowledge
1
M1
Mn
.smv
.smv
SMV Formal Models
Automatic Feature Interaction Detection
Alfie
3
SMV Model Checking
if (c/e)
model + new_property
Create equivalence class for c/e
if ¬(c/e)
one representative c/e per equivalence class
All FI's

Figure 1.2: Feature interaction detection method at design-time

1.3.1 Feature Interaction Definition in the Automotive Domain

I create a definition of feature interactions that is independent from the set of features, and detects contradictory requests to the actuators. This definition is written as temporal logic formulas based on each element of the system or its environment that is influenced by multiple features. My definition is then used for detection of feature interaction as illustrated by number 1 in Figure 1.2. Knowledge about the actuators in the system and their influence on the environment are provided by domain experts. My definition contains two main parts: (i) *Same Actuator* for direct actuator conflicts (*e.g.,* sufficiently different request to Brake); (ii) *Conflicting Actuators* for actuators unrelated in name that cause conflicting physical forces in the environment (*e.g.,* requests to Brake and Throttle). Within these cases, the feature interaction might be: (a) *Immediate* when the conflict is caused by two features making requests in the same step; (b) *Temporal* when the conflict is caused by two requests happening within a certain time threshold of each other. In this dissertation, only immediate feature interactions are detected and temporal ones are left for future work.

1.3.2 Translation of Feature Models Designed in Matlab's Stateflow into SMV

To be most relevant to feature designers, I analyze models by translating automotive active safety features designed using a subset of the STATEFLOW [63] language to the input notation of the model checker SMV [130]. My translator is called *mdl2smv*, and its use in my method is shown by number 2 in Figure 1.2. MATLAB's STATEFLOW is used extensively for designing embedded components in various domains such as the automotive and avionics industries. In STATEFLOW, features are hierarchical state machines, with syntax similar to that of Statecharts [89], but with different semantics. An important distinction from Statecharts is that in STATEFLOW a model runs in a single thread, so there is no true concurrency. I use the SMV model checker because of its powerful features and general-purpose, flexible input language that can reflect precisely the semantics of the design models in STATEFLOW (*e.g.,* the sequential execution of AND-states in STATEFLOW), as well as the semantics of integrated features while detecting feature interactions (where features run in parallel). The translated SMV models contain the same level of description as the design (there is no abstraction), so that the findings of my analysis can be directly understood in terms of the feature model in STATEFLOW. My STATEFLOW models do not consider the vehicle dynamics, thus making analysis using formal verification practical, however, the behaviour of the vehicle dynamics is left completely unrestricted to identify any potential conflicting requests to actuators during analysis. I created a set of non-proprietary feature design models in STATEFLOW to use in my case study because there are no publicly available models. Each of the non-proprietary feature models were designed to fulfill a goal, with no explicit intention of interacting in unsafe ways with other features.

1.3.3 Detection of all Different Feature Interactions at Design-time

In the automotive domain, a comprehensive view of the problem is required by looking at all different feature interactions before deciding on a resolution strategy. Therefore, I propose a method to detect automatically all different feature interactions by performing analysis of the feature models at design time using model checking. However, generating all different counterexamples proved very challenging, and thus, I divide the problem into two subgoals: ***(i)*** Find a summary representing all counterexamples for invariant checking of an extended finite state machine (EFSM), therefore detecting all errors in a model, and ***(ii)*** Generalize the method to find a summary of all counterexamples for a pair of STATEFLOW models of active safety features, therefore detecting a representation of all the feature interactions in the integrated set of features. This section overviews the solution for ***(i)*** followed by the generalization to solve ***(ii)***.

The traditional use of model checking follows a cycle of find bug - fix bug - re-run model checker, until no more counterexamples are found. However, it can be useful to find multiple or all bugs prior to fixing the model [17, 60, 84, 51]. One counterexample by itself may not contain enough information to isolate the error and correctly fix the bug [60]. Additionally, seeing all bugs at once rather than the user iterating this cycle can improve the user's experience of model checking and ultimately, the amount of time it takes to create a correct model, in a similar way to having a compiler return all (or multiple) errors in one pass [17, 60, 51].

But, what is a distinct bug in a model? There is no single answer to this question. For extended finite state machine (EFSM) [52] models with control states and transitions that manipulate data in triggers and actions, the paths through the EFSM provide a way of differentiating one bug from another and removing the non-essential details of the data variations of a single path that fails an invariant. I focus on EFSMs because many commonly-used modelling languages are based on EFSMs (*e.g.,* Statecharts [89], Specification and Description Language (SDL) [7]) and these languages match the internal conceptual models that people use to understand and represent complex systems [122].

Consider the simplified EFSM model of an air conditioning (AC) in Figure 1.3. The input variable e can take on the values enter and exit, and the input variable t (temperature) and controlled variable pt (previous temperature) range over the values 0..2. Checking the invariant that when AC is not in control state **OFF**, the previous temperature has to be less than or equal to 1 if and only if the model is in **IDLE**, the model checker generates the counterexample

$$\begin{array}{l}\langle(\textbf{OFF},\ \text{e=enter, t=1, pt=0}),\\ \ (\textbf{IDLE},\ \text{e=enter, t=1, pt=1}),\\ \ (\textbf{ON},\ \text{e=exit, t=2, pt=1})\rangle\end{array}$$

where AC is not in **IDLE** when the previous temperature is equal to 1. Another counterexample for the same property is

$$\begin{array}{l}\langle(\textbf{OFF},\ \text{e=enter, t=1, pt=0}),\\ \ (\textbf{IDLE},\ \text{e=enter, t=1, pt=1}),\\ \ (\textbf{ON},\ \text{e=enter, t=0, pt=1})\rangle\end{array}$$

and yet another counterexample is

$$\begin{array}{l}\langle(\textbf{OFF},\ \text{e=enter, t=0, pt=0}),\\ \ (\textbf{IDLE},\ \text{e=enter, t=1, pt=0}),\\ \ (\textbf{ON},\ \text{e=enter, t=1, pt=1})\rangle.\end{array}$$

From the user's perspective, these counterexamples are all instances of the EFSM path $\langle$**OFF**-t_1-**IDLE**-t_4-**ON**$\rangle$. The bug is that the model reaches **ON** when the previous temperature is 1. It can be corrected by changing the condition on t_4 to (t>1) instead of (t≥1). The data variations in these counterexamples do not likely help the user in choosing a resolution to this error in the model. We would much rather find a path that shows us another bug in the model, such as counterexample

$$\begin{aligned}\langle(&\textsf{\textbf{OFF}}, \ \textsf{e=enter, t=2, pt=0}),\\ (&\textsf{\textbf{ON}}, \ \textsf{e=enter, t=2, pt=2}),\\ (&\textsf{\textbf{IDLE}}, \textsf{e=exit, t=1, pt=2})\rangle,\end{aligned}$$

which is an instance of the EFSM path $\langle$**OFF**-t_3-**ON**-t_5-**IDLE**$\rangle$. This new bug illustrates that the model reaches **IDLE** when the previous temperature is 2, which can be corrected by changing the condition on t_5 to (t$\leq$1) instead of (t$\leq$2). Therefore, I developed a method to produce automatically one counterexample for each distinct path of an EFSM that fails the invariant, *i.e.,* for each EFSM path that has a bug.

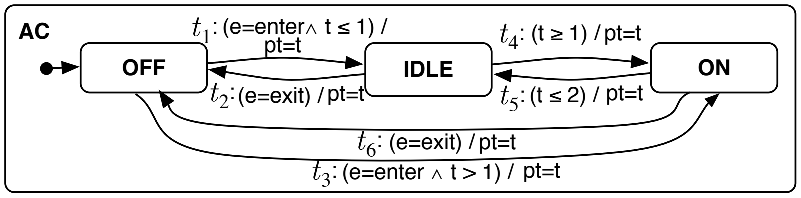


Figure 1.3: EFSM of a flawed air conditioning (AC) model

My solution to the problem of generating all different counterexamples (where "different" is defined using their similarities in the EFSM) is a method that automatically modifies the property, unlike other methods that change either the model checking algorithm ([93, 60, 98]) or the model ([17]). My method and tool, called *Alfie*[3], can work with any model checker (explicit or symbolic) that uses linear temporal logic (LTL), and it covers the complete set of counterexamples because the model is never changed during the process. The use of *Alfie* is illustrated by number 3 in Figure 1.2. *Alfie* executes a cycle of (1) run model checker to find counterexample, (2) find the equivalence class of the counterexample, and (3) re-run model checker on the same model but ruling out all counterexamples within the same equivalence class using a modified LTL property. The counterexample equivalence classes are based on the EFSM's control states and transitions. By ruling out counterexamples on-the-fly, the number of model checking iterations needed is greatly reduced as compared to a process that generates all counterexamples for a given property and then summarizes them after the model checking process (*e.g.,* [60]). Moreover, *Alfie* does not rely on the order in which the counterexamples are generated, such as generation of the shortest one first. One representative of each equivalence class is included in the set of counterexample paths presented to the user, creating a small, useful set of counterexamples to study.

The idea of grouping paths based on control states and transitions of the model's EFSM can be extended to groupings other than just paths. I define four different levels each of which groups the complete set of counterexamples into equivalence classes on-the-fly based on their different properties in the EFSM. For example, if the feature designer wants

[3]The name *Alfie* is derived from All Failed Invariants or All Feature Interactions.

less detail, my method can group all counterexamples that end at the same control state together. The different levels will be useful as different ways to isolate an error at different times during the analysis process.

Then, *Alfie* is extended to handle pairs of STATEFLOW models and reports equivalence classes of feature interactions. Beyond an EFSM, a STATEFLOW model also allows composite states, which are similar to AND-states in Statecharts, but with different semantics [104]. Because of these composite states, a model in STATEFLOW responds to an input in several steps, *i.e.,* as a big-step[4] [72], that is, a sequence of transitions within the components of the feature. Furthermore, two STATEFLOW models representing features must be analyzed concurrently. Thus, the big-step of the combined model is a sequence of sets of transitions. A feature interaction occurs if there are contradictory outputs from both features generated on transitions anywhere within this big-step. To extend my method to detect feature interactions between automotive features modelled in STATEFLOW:

- First, I generalize the description of the LTL representations of the equivalence classes of counterexamples for two concurrent components without calculating the flattened cross product of the two models. The LTL expression must cover all stuttering variants of the combined path.
- Second, I describe a strategy to handle scalability by partitioning the LTL property into two (or more) separate model checking runs that together cover the original property. For large models with many counterexamples, the size of the LTL property representing equivalence classes of counterexamples already seen becomes too large to model check because LTL model checking depends on the size of the property, in addition to the size of the model [143].

My thesis statement summarizes this overview:

Thesis statement: It is possible to make a systematic, complete, and general definition of feature interactions for active safety features that identifies conflicting requests to actuators. Automotive active safety features designed using a subset of MATLAB's STATEFLOW can be translated into the input language of the model checker SMV without losing any details, so the detected feature interactions can be understood in terms of the feature design models in STATEFLOW. Feature interactions in active safety software features can be detected using model checking. Using definitions of equivalent paths based on the control states and transitions of the STATEFLOW model, the set of all feature interactions can be summarized into a manageable set on-the-fly during model checking via property modification. Scalability of the process can be achieved by partitioning the LTL property.

[4]The details on STATEFLOW and big-steps will be explained in Chapter 4.

1.4 Validation

My definition of feature interactions for a set of automotive active safety features, presented in Chapter 3, is *systematic*, *complete*, and *general* with respect to the set of actuators that are controlled by the features and domain expert input, and it is independent of the set of active safety features that are part of the system.

- My definition is *systematic* because from a list of the actuators influenced by the active safety features and the value thresholds at which these actuators interact, LTL properties can be automatically created for the features to analyze, as I show and validate in Section 3.3.
- My definition is *complete* with respect to the set of actuators and thresholds provided by domain experts, meaning it will identify any conflicting actuator requests between two features, as justified in Section 3.3.
- My definition is *general* because it is independent of the behaviours of features that are part of the system. In this way, it differs from the traditional approach used by the automotive domain, in which automotive domain experts list the behaviours in which they expect features to interact. I validate the generality of my definition in Section 3.3, where a justification by cases is provided with respect to new features added to the system and the actuators these features control.

The translated SMV models generated by my tool *mdl2smv* from automotive active safety features, which are designed using a subset of MATLAB's STATEFLOW, do not lose any details during translation. I validate the translation by checking the traceability of the results, *i.e.,* verifying that the SMV counterexamples can be understood in terms of the STATEFLOW model, as shown in Chapter 6 and Chapter 7.

My novel method and tool *Alfie* uses model checking to detect at design time a representation of all feature interactions for a set of active safety systems. For the validation of my method, I created a set of non-proprietary automotive features designed in STATEFLOW, which I call the "University of Waterloo Feature Model Set" (**UWFMS**). The features in the UWFMS are representative in type and complexity of models that I have seen developed in industrial practice [99], but do not include failure modes (*e.g.,* fail-safe states for degraded modes of operation). Therefore, I validate that *Alfie*'s results are *manageable* by illustrating the reduction achieved by my method in Section 5.5 and by observation of the results of my case study that uses the UWFMS, as listed in Chapter 7.

Alfie is made *scalable* by partitioning the problem when the size of the LTL property becomes too big to model check. This validation is done experimentally by analyzing the partitioning performed in my case study, described in Chapter 7.

1.5 Thesis Contributions

The list of the contributions of this dissertation is the following:

- The dissertation identifies the characteristics of automotive active safety systems that make model checking a promising technique to detect feature interactions [105].
- The dissertation introduces a systematic, complete and general definition of feature interactions that identifies contradictory requests by software features to actuators. This definition requires the set of actuators controlled by the features and domain expert knowledge [104].
- The dissertation introduces the UWFMS, a set of non-proprietary feature design models in MATLAB's STATEFLOWto use in my case study because there is not a publicly available set of models [102].
- The dissertation creates the translator *mdl2smv* [103] that generates SMV models from automotive features designed using a subset of the MATLAB's STATEFLOW language. The translated SMV models contain the same level of description as the design.
- The dissertation introduces a novel method and tool *Alfie* to detect a set of counterexamples that is representative of the set of all counterexamples to an invariant for an EFSM by modifying the property being verified on-the-fly. The set of all counterexamples is divided into equivalence classes based on similarity in states and transitions in the EFSM path. *Alfie* produces one counterexample from each equivalence class.
- The dissertation introduces the process to detect a feature interaction when two STATEFLOW models are running in parallel, using the model checker SMV, where the behaviour of the vehicle dynamics is left completely unrestricted to identify any conflicting requests to actuators. This process preserves the individual semantics of STATEFLOW models.
- The dissertation introduces the generalization of *Alfie* to multiple concurrent STATEFLOW models running in parallel to detect a set of counterexamples that is representative of the set of all counterexamples (feature interactions), producing a representative counterexample from each equivalence classes.
- The dissertation introduces a partitioning strategy to deal with scalability, by breaking down the LTL property when it becomes to large for the model checking verification to complete.

1.6 Thesis Organization

Chapter 2 provides background and an overview of related work.

Chapter 3 describes my definition of feature interactions for the automotive domain. My fundamental ideas on feature interactions in the automotive domain were published [105], followed by the more detailed definition for the detection, which has also been published [104].

Chapter 4 describes how to translate automotive features designed using a subset of the MATLAB's STATEFLOW language into the input language of the model checker SMV. The translation tool created is called *mdl2smv*. The research results reported in this chapter have been published [104, 103].

Chapter 5 describes a novel method and tool, called *Alfie*, to represent levels of counterexample equivalence classes that each identify a distinct bug of an EFSM. These levels of equivalence classes can be represented in LTL on-the-fly.

Chapter 6 describes the generalization of *Alfie* to generate levels of counterexample equivalence classes for concurrent components, and shows how the equivalence classes for each level can be represented in LTL on-the-fly to detect feature interactions.

Chapter 7 describes the details of my case study, using the non-proprietary UW feature model set (UWFMS), which I created and is presented in Appendix A. An early version of most of the feature design models was made available as a technical report [102], but now includes updates and two more features. The partitioning strategy used in my case study is described in this chapter.

Finally, a brief summary, a list of the limitations of my method and a discussion of the milestones for future work are presented in Chapter 8.

Chapter 3, Chapter 4, Chapter 5, and Chapter 6 have their own separate related work section at the end of each chapter.

Chapter 2

Background

This chapter starts with an overview of the feature interaction problem in different domains in Section 2.1. The main characteristics of MATLAB's STATEFLOW [162] are described in Section 2.2. A brief overview of model checking is given in Section 2.3. Within the model checking framework, the input language of the model checker SMV is introduced. SMV is used to check for feature interactions, not only because of its powerful features and flexible input language, but also because the language can describe precisely the semantics of individual STATEFLOW feature design models as well as the features' integration.

2.1 Feature Interaction Problem

This section gives an overview of the work that has been done on the feature interaction problem. This has been a very active field in the last few decades, mainly in telecommunications systems, and more recently in Internet applications and embedded systems. I start by defining the concept of feature and feature interaction, followed by an overview of the feature interaction approaches that have been used in various domains.

2.1.1 Feature and Feature Interaction: Definitions and Variants

The concept of a feature depends on the level of granularity at which people decide to work. In an early tutorial on feature interactions, Cameron and Velthuijsen [47] formulate precise definitions of terms such as feature and feature interaction, which vary depending on the perspective and usage of the terms. For instance, from a business view, a feature can simply be a tariffable unit, while from the implementers' view, a feature is any increment of functionality added to an existing system. In both cases, a feature interaction occurs when the behaviour of one feature is altered by the use of another feature.

The main differences in the definitions seem to identify a 'feature' either as a single functionality, or as a bundle of functionality to perform a particular task or goal. A 'functionality' can be described as a capability of what a system can do for a user. More technically, it can be thought of as an individual requirement. A 'service', which has been frequently defined as a set of features, can also be seen as a synonym for a feature if it is referred to as a collection of functionality. When identifying a feature as a module or component, an 'aspect' can be interpreted as functionality that is scattered across several features. However, a common problem recognized by the community is that aspects break modularity [169], so they are not useful in my context. The level of granularity at which a feature is defined also influences the methods proposed to solve the feature interaction problem, since some approaches try to work at the level of individual functionality, while others intend to solve the problem having bundles of system functionality.

I use the definition of *feature* as "a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective" [170]. More technically, a feature can be thought of as "a modularization of individual requirements" of a requirements specification. In the automotive domain, I interpret the definition of feature as "a service recognized by the driver" [104, 124], which identifies a set of functionality in one package. Therefore, my definition of feature interaction intends to identify conflicts among features, thinking of them as modules comprised of user-desired functionality with a common goal.

2.1.2 Classification of Feature Interaction Approaches

Most existing approaches for analyzing and identifying feature interactions have been developed and applied to the telecommunications domain. Approaches for dealing with feature interactions in telecommunications were first classified by Cameron and Velthuijsen [47], and refined by Bouma and Velthuijsen [30] as:

- ***Detection:*** techniques that, if a feature interaction is present, identify and locate such an interaction.
- ***Resolution:*** techniques that, if a feature interaction is detected, try to minimize, or if possible eliminate, the potential adverse effects of the interaction.
- ***Avoidance:*** techniques to prevent undesired feature interactions, where a resolution strategy is usually integrated by design.

This categorization has been widely used since then. Approaches from the three classes might be applied **offline** (*i.e.,* while a feature is being specified, designed and implemented), **online** (*i.e.,* while a feature is tested and after deployment), as well as using a **hybrid** approach (*i.e.,* combination of offline and online). Figure 2.1 illustrates when each of these approaches is applied in the software development process.

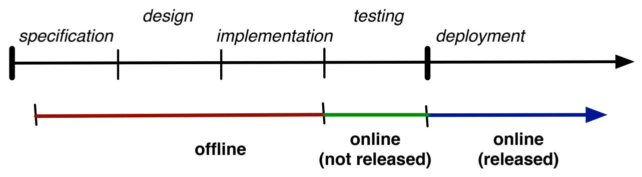


Figure 2.1: Lifecycle timeline for the approaches

2.1.3 Approaches to Deal with Feature Interactions for Telecommunications

This section provides a broad overview of the approaches that have been proposed to solve the feature interaction problem in telecommunications. Examples of contributions in each class of approaches are given throughout the section.

Offline Approaches for Telecommunications

Offline techniques are mainly used in the early stages of software development and are applied to existing requirements specification documents, design models and during implementation. Analysis offline is also commonly referred to as analysis at design-time or static analysis.

Most offline detection techniques are exhaustive and based on **formal methods**, which involve the application of symbolic analysis techniques to check all possible behaviours of a model. Examples of this approach can be found in papers by Accorsi *et al.* [9], Au and Atlee [15], Bergstra and Bouma [23], Blom *et al.* [26], Blom [25], Boström and Engstedt [29], Bruns *et al.* [36], Calder and Miller [44], Capellmann *et al.* [48], Combes and Pickin [59], Felty and Namjoshi [73], Frappier *et al.* [75], Gammelgaard and Kristensen [78], Gibson [80, 81], Hall [86], Kamoun and Logrippo [107], Khoumsi [111], Khoumsi and Bevelo [112], L. de Bousquet *et al.* [64], LaPorta *et al.* [120], Nakamura *et al.* [135], Plath and Ryan [146, 147], Pomakis and Atlee [148], Rochefort and Hoover [156], Siddiqi and Atlee [161], Stepien and Logrippo [163], Thistle *et al.* [164], Thomas [165], Van Der Straeten and Brichau [174], and Yoneda and Ohta [190].

Offline detection of feature interactions can be aided by **filtering** (*i.e.,* pruning) to reduce the number of cases to be considered for feature interactions. There has also been some work on informal offline approaches using heuristics and probabilistic search for undesired feature interactions. Examples of this approach can be found in papers by Bredereke [33], Heisel and Souquières [91], Keck [109], Kimbler [113], Kimbler *et al.* [114], Kimbler and Sobirk [115].

Other offline techniques for detection, that also include resolution, are **architectures** and design policies. To create an architecture, one normally starts with exhaustive search for interactions and a study of domain specific attributes to produce design guidelines or architectural rules that will prevent interactions. Examples of this approach can be found in papers by Braithwaite and Atlee [32], Hay and Atlee [**?**], Jackson and Zave [96], Utas [172], van der Linden [173], Zibman *et al.* [193], Turner [171], and Zimmer and Atlee [194].

Online Approaches for Telecommunications

Online techniques are mostly used during testing, and after deployment. Online analysis is also commonly referred to as analysis at run-time or dynamic analysis. Online detection techniques work at run-time, checking for undesired interactions at each step of the system's execution. Usually, online detection techniques are simple and are used to aid the online resolution process.

One of the main techniques used for online resolution is the introduction of a **feature interaction manager** (FIM) entity into the network to observe and control the processes, and all components communicate with the FIM, which in turn determines if an interaction occurs and proceeds to resolve it. Examples of this approach can be found in papers by Aggoun and Combes [11], Cain [40], Fritsche [77], Homayoon and Singh [94], Jia and Atlee [97], Marples and Magill [129], Pang and Blair [140], Reiff [153], and Tsang and Magill [167, 168].

Another technique used online is **negotiation**, where features have the capability of communicating their intentions to each other and, in the case of an interaction, negotiating an acceptable resolution. Features are implemented with knowledge of the other features that will be executing. Examples of this approach can be found in papers by Amer *et al.* [13], Buhr *et al.* [38], Griffeth and Velthuijsen [83], and Velthuijsen [176].

Hybrid Techniques for Telecommunications

A hybrid technique uses an online approach for detection, which is complemented with resolution of interactions based on offline information. Examples of this approach can be found in papers by Calder and Reiff [45], and Calder *et al.* [43, 41]. An example of a hybrid approach from Calder *et al.* [43] is the following: the knowledge from offline analysis such as feature precedence relations, generic constraints laws (*e.g.,* message x must never be followed by message y as it lead to a deadlock), and theories of maximal satisfaction of a set of features are incorporated in a *hybrid feature manager* for pruning/selection, while the manager's run-time experience will drive more knowledge for detection and resolution.

Summary

Usually, the set of possible feature interactions is not well defined because a domain often evolves by adding new types of features and integrating new technologies, so architectures are not likely to be complete enough to detect all the feature interactions in a system [24, 86]. A known disadvantage of exhaustive verification analysis, using formal methods, is that determining when the feature interactions occur often encounters the state explosion problem because of the combinational explosion in the number of features to analyze. When dealing with online resolution strategies, a disadvantage is that it makes integration or removal of features difficult because the online monitors need to be modified based on a specific feature set. Finally, a disadvantage when applying hybrid techniques is finding people with the expertise to integrate offline and online techniques.

For automotive active safety features, I propose the use of model checking at design-time to exhaustively detect feature interactions, so a resolution strategy can be chosen at a later time. My method also introduces a partitioning strategy to deal with scalability, breaking down the verification problem into two or more subproblems that cover the original, thus, potentially overcoming the state explosion problem. Moreover, my proposed method can use a definition of feature interactions that is independent of the set of features in the system.

2.1.4 Approaches to Deal with Feature Interactions in Internet Applications

This section provides an overview of the techniques that have been developed and applied to networked systems connected to the Internet. The systems that have been considered are VoIP, e-mail, web services and networked home appliances, and examples of the techniques proposed to deal with the feature interaction problem are given in the rest of this section.

Several approaches have been proposed to deal with Internet communication systems. Crespo *et al.* [62] describe the detection and resolution of feature interactions in Internet applications using a two-phase approach: (1) Filter features based on priorities among features' actions, and (2) Use negotiation to select the features to execute based on policies that identify compulsory features. A similar two-phase approach is proposed by Gouya and Crespi [82], where an offline selection is performed first, using a feature conflict database, and the remaining interactions are handled by an online feature interaction manager. Chi and Hao introduce two techniques to generate test sequences to check for correctness of a feature-rich communication system, as well as to detect feature interactions in this type of system [53]. Wu *et al.* describe an approach for the detection and resolution of functional and non-functional interactions between Internet and telecommunication features using a manager [187]. Nakamura *et al.* introduce an approach that detects feature interactions at run-time when the behaviour of one feature is not executed as described when other features are executing concurrently [136]. Following a different trend, Crespo proposes a proactive approach for feature interaction detection, where the system's events, predicates and potential inconsistent behaviours are used to generate hypothetical features that interact with the features that are part of the system [61].

Hall was the first to propose an approach to detect feature interactions in electronic mail based in part on human intuition and in part on simulation and test coverage [87]. Pang and Blair introduce a method to resolve feature interactions among distributed e-mail features [141]. Their method proposes two complementing resolution strategies: (1) Features are augmented with logic that explicitly deals with potential conflicts with other features, and (2) Negotiation based on operation precedence is used for any interactions that cannot be handled by the augmented feature logic.

Weiss and Esfandiari describe a method to detect and resolve feature interactions between web services at the requirements level, based on the analysis of a goal graph derived from the goal-oriented requirements [178]. They extend these ideas to propose a classification of web service feature interactions, classified by their nature or by their causes [179]. Zheng *et al.* proposed a model checking-based method to detect feature interactions in web services [189]. They classify feature interactions in five categories (deadlock, loop, invocation error, race condition and resource contention), which are detected using properties specified in LTL.

Kolberg, Magill, and Wilson consider feature interactions in home automation systems connected to the Internet [116, 183]. They use an online feature interaction manager (FIM), which detects an interaction as shared access to environmental variables with different attributes. For the resolution strategy, features priorities and access attributes are used. However, this technique requires the model environment to be given, with the shared variables and their attributes explicitly defined. Also, FIMs have to be redesigned if new devices influence physical environmental variables that were not considered before.

Summary

Crespo *et al.* [62] point out that given the distributed nature of the Internet, with multi-vendor and multi-provider environments along with end user capability to program and tailor features, it is not possible to rely on avoidance. Most approaches proposed have an online component, as the features might meet for the first time when executing, and offline approaches may not be sufficient to detect and resolve all kinds of interactions. In my work, I handle inter-vehicle features where the feature set for a vehicle is known in advance, making an offline approach applicable. This is a significant difference with respect to automotive active safety features, although some of these techniques might prove useful in the future, when considering detection of intra-vehicle feature interactions.

2.1.5 Approaches to Deal with Feature Interactions for Embedded Systems

This section provides an overview of the techniques that have been developed and applied to embedded systems, which is the domain most similar to that of automotive active safety features. Not much work has been done in this domain, and many challenges regarding the proper management of feature interactions still remain. The last two techniques described in this section, although developed to address the detection of feature interaction in the automotive domain, do not solve the problem or report any case studies to compare with.

Metzger introduces an approach for the automatic detection of feature interactions in embedded control systems, where a feature interaction is defined as an element of multiple feature influence (using my terminology, see page 34) on an object diagram [132, 131]. Lochau and Goltz propose a test case generation method for feature interaction analysis between Statechart-like behavioural models, defining test cases whenever two features access (read or write) to a shared variable [124]. D'Souza, Gopinathan *et al.* use concepts of supervisory control theory to detect and resolve feature interactions, based on a notion of "conflict-tolerance" [67, 68]. Thinking of each feature as a *supervisor* or *controller*, this framework follows the process of Thistle *et al.* [164, 184] and detects an interaction as a blocking controller conjunction. Their resolution strategy involves two parts: (1) Use of a predefined priority of execution among features, and (2) Features are aware of potential conflicts, extending the functionality of each feature to continue operating based on features' priorities.

Summary

The approach by Metzger is conservative because it does not consider the operational behaviour of each feature, and therefore, false positives are likely to be reported, unlike the method I propose for detection of feature interactions. Also, Metzger's detection approach requires the existence of environment models, in contrast to my method that does not need an explicit model of the environment.

The method by Lochau and Goltz, similar to other test case generation approaches, does not identify all feature interactions, as opposed to my method, which creates all equivalence classes of feature interactions for pairs of Stateflow models and avoids slight data variations of paths that might be generated by test cases. Moreover, even though they describe the input of their analysis as Stateflow models, the semantics described in the paper does not match Stateflow, as an AND-state is not truly concurrent.

The disadvantages of the approach by D'Souza, Gopinathan *et al.* are that it only detects one class of feature interaction, and that the priorities used or the conflict-tolerance functionality added to features might not consider all possible conflicts in the system. In contrast, the method I propose detects all classes of feature interactions, and thus, can help to the process of adding complete conflict-tolerance functionality.

2.2 Stateflow

This section describes the main attributes of Matlab's Stateflow, providing a brief overview of its notation. Automotive features are designed in Matlab's Stateflow by multiple companies and suppliers. Therefore, I use Stateflow feature design models as input to my feature interaction detection analysis. In Chapter 4, a method to translate from Stateflow models to the modelling language of SMV is presented, so the detection process using model checking tools can be performed. Matlab's Stateflow does not have formal semantics, thus, its behaviour is defined via simulation, as described in this section. The information presented in this section is derived from the Stateflow documentation [2], as well as Dabney and Harman's book [63].

Matlab is a numerical computing environment and programming language. Simulink is a software package extension to Matlab that lets engineers rapidly and accurately model and simulate dynamic systems using block diagram notation. Stateflow is an interactive graphical design and development language for complex control and supervisory systems. Stateflow supports visual modelling and simulation of complex reactive systems by integrating finite state machine (FSM) concepts, Statecharts' formalisms (as developed by Harel [89]), and flow diagram notations. A Stateflow model can be included in a Simulink model as a subsystem.

The syntax of Stateflow is similar to that of Statecharts. Some of the differences from Statecharts are that the Stateflow action language has been extended to reference Matlab functions, use of early return logic to resolve conflicts arising from event broadcasts, and that Stateflow does not perform true concurrency for AND-states as Statecharts does. In Stateflow, AND-states execution is sequential: each AND-state reacts to the same input, but only one AND-state executes at a time.

A Stateflow design model consists of a set of *states* connected by arcs called *transitions*. A state can be refined into a Stateflow diagram, creating a hierarchical state diagram. The Stateflow documentation defines two kinds of decomposition for a state, which are: (1) 'exclusive' or 'OR-states' (indicated by solid borders) and (2) 'parallel' or AND-states' (indicated by dashed borders). In Stateflow, at each level of the hierarchy, only one kind of decomposition can be chosen, *i.e.,* all states at the same hierarchy level must be either OR-states or AND-states, which is illustrated by Figure 2.2. Unlike Statecharts, Stateflow AND-states are not truly concurrent since Stateflow actually runs in a single thread during simulation. I call Stateflow AND-states *ordered-compositions* to differentiate them from the ones in Statecharts.

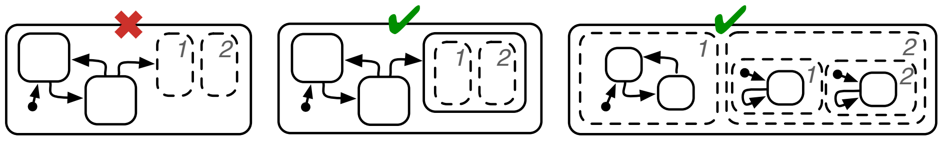


Figure 2.2: Decomposition allowed at a hierarchy level in Stateflow

Each ordered-composition is executed sequentially following its respective execution order. The execution order is based on the geometric position of the siblings in an ordered-composition, where priority is assigned from top to bottom and then from left to right, according to the rules:

- The higher the vertical position of a sibling in an ordered-composition, the higher its priority for execution.
- Among siblings in an ordered-composition with the same vertical position, the left-most sibling receives highest priority.

The lower the number, the higher the priority. This order determines when each sibling in an ordered-composition executes its actions, only one sibling at a time. The same set of inputs is used for all the siblings in an ordered-composition.

A Stateflow model can have data input/output ports and event input/output ports. Both data and events can be defined as local to the Stateflow model or external, *i.e.,* communicated from the Simulink parent model through ports. The types of data allowed by Stateflow are: Boolean, integer, real (fixed and floating point), and enumerated. For numerical data types, the range limit can be indicated, and if not, it assumes the default of (`-inf`, `inf`), which basically means undefined type. The default initial value for a variable is 0. Each transition's label follows the syntax:

event[condition]{condition action}/transition action

Each part of the label is optional. The event specifies an event that causes the transition to be taken, provided the condition, if included, is true; the condition is a boolean expression on data that, when true, allows a transition to be taken; the condition_action is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid; the transition_action is executed after determining that the transition destination can be reached. Each transition has also a priority of execution, determined by the hierarchy level of the transition's destination state, the type of information in its label (*e.g.,* events have priority over conditions) and the geometric position of the transition source. The lower the number, the higher the priority.

A history junction represents historical decision points in the Stateflow diagram, indicating that historical state activity information is used to determine the next state to become active.

In Chapter 4, I will describe the subset of Stateflow used to model automotive active safety features, which I support in my translator.

2.3 Model Checking

Formal methods are mathematically-based analysis techniques for the specification, development and verification of software and hardware systems [8]. One such technique is model checking, whose main characteristics are described in this section. Because the feature interaction problem in the automotive domain studied in this dissertation is bounded, offline model checking of feature designs is a promising detection technique [105]. The descriptions of model checking in this section are based on Clarke *et al.* [57], Peled [143] and Bérard *et al.* [21].

Model checking is an automatic technique for verifying finite state concurrent systems, which has been successfully used to verify complex sequential circuit designs and communication protocols. It is a powerful technique for finding errors, inconsistencies and contradictions in a model because it searches exhaustively all behaviours of the system, unlike other traditional approaches such as simulation and testing [57]. If a model does not satisfy a property describing a desired behaviour, a model checker produces a **counterexample**, which is a path of the model's behaviour that fails the property. The main challenge of model checking is the state explosion problem, which will be described in Section 2.3.4. A user of model checking needs to perform three main tasks: modelling, specification and verification, which are explained next.

2.3.1 Modelling

This task consists of constructing a formal model of the system in a notation that is accepted by the model checking tool, and thus, allowing the automatic analysis of the system specified by the model.

Common notations to describe a model are labelled transition systems, state machines (*e.g.,* finite or Büchi), Kripke structures and Petri nets. These notations use the concepts of states and transitions to define the behaviour of the system. A *transition* can be described as the rules defining the change from one state to another in the system. Therefore, the computations of a system are defined in terms of its transitions.

2.3.2 Temporal Logic Specification

This task consists of identifying the properties that the model must satisfy. The properties are normally described using a logical description such as temporal logic, which is a formalism for describing paths: sequences of states in a system [128]. Temporal logics are useful to describe the ordering of system events over time without introducing time explicitly, but differ in the operators they provide. The main temporal logics used in model checking tools are Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). The analysis in this dissertation uses LTL.

For ψ and ϕ, predicates on states, informally the operators of LTL are:

- Next ($\mathbf{X}\ \psi$): ψ must hold at the next position in the path.
- Eventually ($\mathbf{F}\ \psi$): ψ must hold at a future position in the path.
- Globally ($\mathbf{G}\ \psi$): ψ must hold on the entire path.
- Strong Until ($\psi\ \mathbf{U}\ \phi$): ϕ must hold at the current or a future position, and ψ has to hold until that position. From there on, ψ does not need to hold.

The *syntax* of LTL is the following, where AP is a set of atomic propositional formulas:

- Every formula of AP is a formula of LTL,
- If ψ and ϕ are formulas, then so are $(\neg\psi)$, $(\psi \wedge \phi)$, $(\psi \vee \phi)$, (X ψ), (F ψ), (G ψ), and (ψ U ϕ).

The semantics of LTL are normally defined with respect to a **Kripke structure** (KS), which is the simplest model to represent a system. Its formal definition is as follows.

Definition 2.1 *Let AP be a set of atomic propositions. A* ***Kripke structure*** *$\mathcal{M}$ over AP is a four tuple $\mathcal{M} = (S, S_0, R, L)$ where*

1. *S is a finite set of states.*
2. *$S_0 \subseteq S$ is the set of initial states.*
3. *$R \subseteq S \times S$ is a transition relation that must be total, i.e., for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.*
4. *$L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.*

Sometimes, the set of initial states are not of interest, and this set is omitted from the definition. A *path* in the structure $\mathcal{M}$ is an infinite sequence of states π=$s_0, s_1, s_2, \cdots$ such that $s_0 \in S_0$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$. Therefore, an LTL formula is interpreted over an infinite sequence of states, $\pi{=}s_0, s_1, s_2, \cdots$. The *suffix* of π starting at s_i is denoted by π^i. If ψ is a state formula, the notation $\pi \models \psi$ means that ψ holds along path π. Then, the definition of the *semantics* of LTL for an arbitrary suffix π^i of a sequence π is the following:

- $\pi^i \models p$, where $p \in AP \Leftrightarrow s_i \models p$
- $\pi^i \models (\neg\psi) \Leftrightarrow \pi^i \not\models \psi$
- $\pi^i \models (\psi \wedge \phi) \Leftrightarrow \pi^i \models \psi$ and $\pi^i \models \phi$
- $\pi^i \models (\psi \vee \phi) \Leftrightarrow \pi^i \models \psi$ or $\pi^i \models \phi$
- $\pi^i \models (\mathrm{X}\ \psi) \Leftrightarrow \pi^{i+1} \models \psi$
- $\pi^i \models (\mathrm{F}\ \psi) \Leftrightarrow$ there is a $k \geq i$ such that $\pi^k \models \psi$
- $\pi^i \models (\mathrm{G}\ \psi) \Leftrightarrow$ for every $k \geq i$, $\pi^k \models \psi$
- $\pi^i \models (\psi\ \mathrm{U}\ \phi) \Leftrightarrow$ there is a $k \geq i$ such that $\pi^k \models \psi$, and for all j, where $i \leq j < k$, $\pi^j \models \phi$

The notation $\mathcal{M} \models \psi$ means that ψ holds for all paths of $\mathcal{M}$.

2.3.3 Verification

A model checking procedure searches the state space of the system exhaustively to determine if some specification is true or not. More formally, the model checking problem consists of verifying if a model of the system $\mathcal{M}$ satisfies a formal specification ϕ, *i.e.,* $\mathcal{M} \models \phi$. Given enough resources, the procedure will terminate with a yes/no answer. Moreover, if the answer is no, the algorithm normally provides a counterexample, *i.e.,* a path of the model's behaviour that does not satisfy the specification.

The original implementation of the model checking algorithm was *explicit*, representing transition relations by adjacency lists [55, 152]. However, for concurrent systems with many components, the number of states in the global transition graph became too large, and the model checking procedure could not successfully terminate. Then, Burch, Clarke, McMillan *et al.* introduced a *symbolic* model checking algorithm [39, 130], which uses a more compact representation for the state transition graphs based on Bryant's *binary decision diagrams* (BDDs) [37].

The automata theoretic framework for model checking was suggested by Kurshan [10] and also by Vardi and Wolper [175], which is the classical view of model checking for LTL. Representing the system model $\mathcal{M}$ and the specification $\mathcal{S}$ over the same alphabet, the general strategy for model checking described by Peled [143] is as follows:

First, complement the automaton $\mathcal{S}$, *i.e.,* construct an automaton $\overline{\mathcal{S}}$ that recognizes the language $\overline{\mathcal{L}(\mathcal{S})}$. Then, intersect the automata $\mathcal{M}$ and $\overline{\mathcal{S}}$. If the intersection is empty, the specification $\mathcal{S}$ holds for $\mathcal{M}$. Otherwise, use an accepted word of the nonempty intersection as a counterexample.

The basic idea of symbolic model checking methods is to represent sets of states concisely and to manipulate them as sets [21]. If both, the state transition relation of the model and the specification in LTL are represented as Boolean functions, such as BDDs, the model checking algorithm can be implemented as operations on those functions, therefore, not only reducing the model checking effort, but also increasing the size of the models that can be analyzed [69].

A path reported by the model checker is an infinite path, composed of a finite prefix and a cycle at the end, where the cycle must contain the state that fails the property. However, in the case of an invariant property[1], formulated as G p with $p \in AP$, the model checker can modify its search to find reachable bad states from the initial states, and thus, a counterexample would simply be a finite sequence ending in a bad state since no matter how the infinite sequence is completed, the property was already violated [143]. In this case, the counterexample returned by the model checker is a truncated path.

2.3.4 State Explosion Problem

A challenge that prevents the wide spread use of model checking is the *state space explosion problem.* The problem occurs in systems with a large number of interacting components, since the verification of the behaviour of these systems consists of enumerating and analyzing the set of system states that can ever be reached, so the number of system states can be too big to be handled.

Many techniques have been proposed to aid with the state space explosion problem, and some examples are described next.

- Application of *partial order reduction* [144, 142] to reduce the size of the state space by constructing a reduced state graph. This technique exploits the commutativity of concurrently executed transitions that result in the same state when executed in different orders. In the abstract model, all those transitions are lumped together into one.

- Application of *abstraction* techniques to produce a high level description of the system. One technique, *cone of influence reduction* [119], attempts to reduce the model size by only focusing on variables that contribute to the verification of the property.

[1]An invariant property is one that must be true at all times during the execution of the model.

Another technique, *data abstraction* [125, 56], attempts to map the actual data values in the system to a small set of abstract data values.

- Application of *symmetry reduction* [70, 138] to reduce the size of the model by exploiting symmetry in the system and finding a model that is equivalent to the original one, but smaller. Symmetry in a system means that there exist nontrivial permutation groups that preserve the state labelling and the transition relation, so the permutation groups are used to define equivalent classes on the state space of the system.

2.3.5 The Model Checker SMV

In this section, the Cadence SMV notation that is used in the translated models from STATEFLOW is described. SMV was chosen because it allows me to describe precisely the semantics of STATEFLOW feature design models, as well as the integration of features. The description is based on McMillan [130].

The general format of an SMV module is shown in Figure 2.3. An SMV model consists of a set of modules and a main module. An SMV module is composed of declarations, assignments, and optionally, assertions. Each model can also have formal parameters, which are often declared as inputs (assigned outside the module) or outputs (assigned inside the module).

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

        /* *** Assignment Section *** */
        name := value;
        init(name2) := value;
        next(name2) := NEWvalue;
        DEFINE name3 := condition;
        ...

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

Figure 2.3: General format of SMV modules

The *Declaration Section* contains the input, output and local type declarations, with the input and output declarations occurring before any local declarations and assignments.

In the *Assignment Section*, a set of assignments of the form 'name := value;' are declared, indicating how the variables change value. Different operators can be used in the *Assignment Section*, and for my translator, I used operators such as Boolean ("and", "or", "not"), conditional ("if-then-else", "case", "switch"), arithmetic ("+", "-", "*", "/"), and comparison ("=", "<", ">", ">=", "<="). Special operators for describing recurrences are "init" and "next". The sequence of values of a variable in a path of the computation is described using `init(x)`, which specifies the initial value of `x` (*i.e.,* the first value of `x`) and `next(x)`, which denotes the next value of `x` (*i.e.,* the (i+1)-th value of `x`). The next value of `x` is defined using operators and constants from the range of values that `x` can take given its declaration.

A macro uses the `DEFINE` statement to give a concise and meaningful name to a constant or a conditional directive.

The *Assertion Section* contains the properties that need to be checked. An assertion is a condition that must hold true in every possible execution of the program, and in SMV each assertion is written in LTL.

2.4 Summary

In this chapter, I defined the term feature, and provided a classification of the approaches that have been used to deal with feature interactions in various domains. I have also explained the main characteristics of MATLAB's STATEFLOW, which is the main tool used in practice by the automotive and avionics domain to design embedded controllers, such as automotive features. Finally, I described model checking, a formal methods technique for the automatic verification of software and hardware systems. In the same section, I provided an overview of the model checker SMV, which will be used for the detection of feature interactions. My past experience with formal methods techniques has been the use of a combination of model checking and theorem proving approaches to evaluate LTL correctness properties of Distributed Feature Composition (DFC) architecture [100, 101].

Chapter 3

Definition of Feature Interactions for Automotive Systems

This chapter presents a systematic, complete and general definition of feature interactions for the automotive domain that identifies contradictory requests to the actuators. It is independent of the set of features, and it is based on the set of actuators controlled by active safety features and domain expert knowledge.

The present chapter is organized as follows. Section 3.1 describes the characteristics of the automotive domain that allows me to identify feature interactions for automotive active safety features. Section 3.2 presents my systematic, complete and general definition of feature interactions for automotive active safety features. Section 3.3 provides validation of the definition presented in Section 3.2. Section 3.4 discusses related work.

3.1 Characteristics of Automotive Active Safety Features and their Interactions

Across all domains, an interaction arises when the features have conflicting effects on a system and its environment. While the behaviour of both features may be correct according to each feature's intended behaviour, their interaction is undesired[1].

I have observed that traditionally, automotive domain experts identify feature interactions by trying to come up with all the scenarios in which they think that the features under consideration would interact, making the detection dependent on the behaviours of individual features, and therefore, on the set of features that are analyzed.

[1]Recall that in this dissertation, the term *feature interaction* is always meant to refer to unsafe interactions among subsystems.

Example: When identifying interactions for features that command the vehicle's motion control systems (*e.g.,* Collision Avoidance commanding the brakes and Cruise Control commanding the throttle), domain experts describe the steps in each feature's behaviour that lead to the interaction (*e.g.,* the behaviour of one feature leading to braking while another feature's behaviour leads to acceleration, causing an unsafe situation where the driver loses control of the car). The set of behaviours are identified by domain experts as the interactions to detect.

However, even domain experts may not be able to list all the possible features' behaviours, and thus, this traditional approach does not guarantee that all feature interactions are identified.

While classifications of feature interactions in telecommunications exist (*e.g.,* [42, 110]), these classifications do not systematically lead to a set of properties to detect feature interactions. But the characteristics of automotive active safety features make it possible to create a systematic, complete and general definition of feature interactions. The characteristics for a definition of feature interactions, as introduced in Section 1.1, are:

- There is only one copy of an active safety feature that can be active in a vehicle at any time, making it possible to consider these systems as if the features are invoked statically.
- The set of actuators of automotive active safety features and their range of possible values are known in advance, and this information is available from the domain experts.
- The interactions appear in the physical part of the system, *i.e.,* the environment of the active safety features, in the form of contradictory physical forces in the environment (*e.g.,* actuators for brakes and throttle conflict in their influence on speed).

Given these observations of the characteristics of automotive active safety features, a prerequisite for a feature interaction in the automotive domain is that two features both influence (*i.e.,* modify) the same element of the system or its environment. Thus, the search for feature interactions can be limited to those that influence the same element, creating unsafe situations.

In automotive systems, the elements influenced by active safety features are either (1) the actuators that receive requests from the features, or (2) elements of the environment influenced by the actuators' actions. Figure 3.1 illustrates that in the automotive domain an element influenced by multiple features is either directly modified by the system and referred to by both features using the same name (*e.g.,* throttle); or it might exist outside the system as a conflict manifested in the environment caused by outputs of the system (*e.g.,* speed in the environment affected by brakes and throttle). For active safety features, the actuators are brakes, steering and throttle, while the elements in the environment influenced by multiple actuators are speed (influenced by brakes and throttle) and position (influenced by throttle and steering). Both, the set of actuators and the elements of the environment influenced by actuators, are well-known to all feature designers. In contrast, a definition similar to the one proposed here would not work in telecommunications because in that domain feature interactions are conflicts on the creation of the one actuator, the route, which has an unbounded set of possible values.

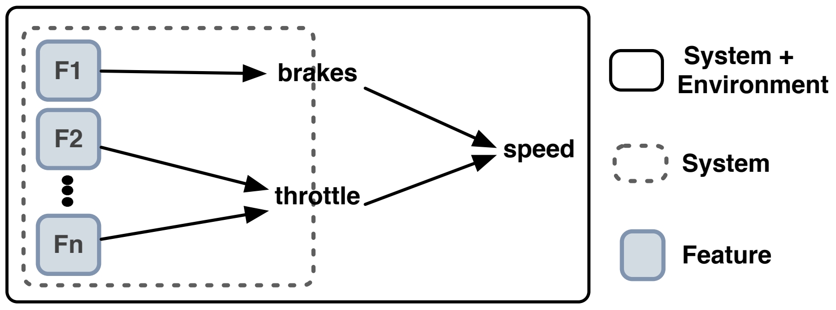


Figure 3.1: Example of output dataflow of feature influences

Simply searching the system's dataflow for influences among the features to determine feature interactions, as done by Metzger [131], would be too conservative an approach alone because even if two features influence the same element, they might never influence the element at the same time when the features are operating. Thus, the operational behaviour of the features needs to be considered. In addition, searching the system's dataflow would not uncover conflicts that appear outside the system of features. Therefore, I use the elements of multiple feature influences to construct temporal logic properties that check the operational behaviour of the features during model checking to detect whether a feature interaction can occur. My definition of feature interactions takes as input (a) the names of the actuators and (b) sets of actuators' names that all influence the same element in the environment. This information can be determined by domain experts and is unlikely to change across projects that deal with the same kind of features, *e.g.,* active safety features.

3.2 Definition of Feature Interactions

Table 3.1 and Table 3.2 present the schemas of my definition of feature interactions for automotive active safety features, described as formulas in linear temporal logic (LTL). All the properties are described using the globally operator (G) to verify the absence of feature interactions. This definition contains two main parts: (1) *Same Actuator* for direct actuator conflicts, and (2) *Conflicting Actuators* for actuators that cause feature interactions in the environment. In the tables, the schema uses assign_X and assign_Y to represent a value assignment made to actuator X and actuator Y respectively, meaning that the feature is setting the value of X or Y. A feature interaction may be *Immediate*, in that the conflict is caused by features making requests in the same step, or *Temporal*, in that the conflict is caused by requests happening within a certain time threshold of each other. I abbreviate the phrase "feature interaction detection property" to FIDP.

It is sufficient to analyze automotive features pairwise since any interaction in a set of k features, with $k > 2$, would be detected as an interaction between two features because the output of one feature never directly enable another feature. In the telecommunications domain, Kawauchi and Ohta indicate that 3-way interactions appear when one of the three features under consideration enables certain behaviour [108] (*i.e.,* Terminating Call Screening (TCS) restricts the reception of calls from certain phone numbers or area codes, but another feature enables the restricted behaviour). When two features are provided simultaneously, if the execution of the feature that causes the potential interaction is disallowed, a feature interaction does not occur. However, when the third feature is applied to the pair under consideration, its output becomes an input that enables the execution condition of the feature that was prevented from executing when only two features were present, and therefore, causes the interaction. In contrast, in the automotive domain, all the active safety features have independent inputs, and their outputs never directly enable other features. Therefore, there is no potential of 3-way interactions that are not 2-way interactions for active safety features. Pairwise detection is advantageous for the model checking process because it reduces the size of model to analyze.

I assume that each feature is designed correctly with no inner feature interactions (*i.e.,* no conflicting request to actuators come from the same feature), but a procedure to check for these interactions within a feature as a design error could be created easily. My definition detects feature interactions in a pair of Stateflow models running in parallel with no modelling of vehicle dynamics.

3.2.1 Immediate Feature Interactions

For all pairs of active safety features that influence an actuator, an *Immediate* feature interaction is a race condition on such an actuator. The schema for the *Immediate* FIPDs are shown in Table 3.1. The goal of the *Immediate Same Actuator* FIDPs is to detect a situation where two features request sufficiently different values for an actuator that it is considered a feature interaction. The *Immediate Same Actuator* set of FIDPs detects if the actuator requests from a pair of features have sufficiently different values based on a value threshold, *e.g.,* $\mid \texttt{assign\_throttle}_1 - \texttt{assign\_throttle}_2 \mid > \texttt{value\_threshold}_{\texttt{throttle}}$, with $\texttt{assign\_throttle}_1$ being the request from one of the features and $\texttt{assign\_throttle}_2$ being the request from the other feature, which is illustrated by Figure 3.2 **(a)**. The value thresholds would be determined by a domain expert and could be zero to detect if the requests to the actuators are different.

	Immediate
Same Actuator	$\mathrm{G}\ \neg(\mid \mathsf{assign\_}X_1 - \mathsf{assign\_}X_2 \mid > \mathtt{value\_threshold})$
Conflicting Actuators	$\mathrm{G}\ \neg(\ (\mathsf{assign\_}X > \mathtt{value\_threshold}_X)$ $\wedge\ (\mathsf{assign\_}Y > \mathtt{value\_threshold}_Y))$

Table 3.1: Property to detect immediate feature interactions

The *Conflicting Actuators* FIPDs are properties for pairs of different actuators (X and Y in the schema) that both influence an element of the environment. The *Immediate Conflicting Actuators* case detects if the requests to the conflicting actuators are both greater than each actuator's value threshold *e.g.,* $((\mathtt{assign\_brake} > \mathtt{value\_threshold}_{\mathtt{brake}}) \wedge (\mathtt{assign\_throttle} > \mathtt{value\_threshold}_{\mathtt{throttle}}))$, with `assign_brake` being the request from one of the features and `assign_throttle` being the request of the other feature, as illustrated by Figure 3.2 **(b)**. The value thresholds for *Same Actuator* FIPDs and *Conflicting Actuators* FIPDs do not need to be the same, as different conditions are checked in each case.

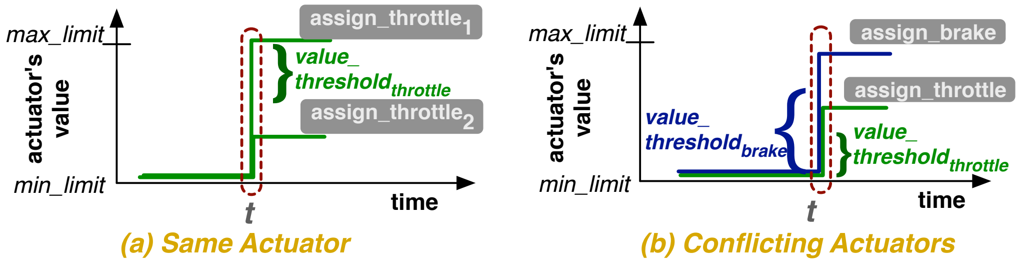


Figure 3.2: Illustration of immediate feature interactions

3.2.2 Temporal Feature Interactions

Because the actuators are requests to the mechanical processes, such as "reach maximum braking", the effects of these requests will not always be instantaneous, so a feature interaction can occur between requests of the features at distinct times. For example, a feature interaction occurs if a feature requests full throttle force at time t, and while the throttle is still increasing, another feature requests a small amount of throttle at a time $t + 1$. I call this case a *Temporal* interaction. The schema for *Temporal* FIPDs are shown in Table 3.2.

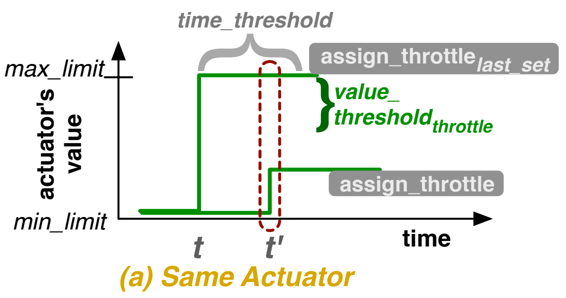
	Temporal
Same Actuator	$G\ \neg((\mid \mathsf{assign\_}X - \mathsf{assign\_}X_{\texttt{last\_set}} \mid > \texttt{value\_threshold}) \wedge ((t_{\texttt{now}} - t_{\texttt{last\_}X}) < \texttt{time\_threshold}))$
Conflicting Actuators	$G\ \neg(\ (\mathsf{assign\_}X > \texttt{value\_threshold}_X) \wedge (\mathsf{assign\_}Y_{\texttt{last\_set}} > \texttt{value\_threshold}_Y) \wedge ((t_{\texttt{now}} - t_{\texttt{last\_}Y}) < \texttt{time\_threshold}))$

Table 3.2: Property to detect temporal feature interactions

A domain expert determines the threshold of time within which contradictory requests to an actuator constitute a feature interaction. To describe a temporal feature interaction in LTL, two history variables are maintained: one to capture the last time an actuator was assigned a value (*e.g.*, $t_{\texttt{last\_brake}}$), and one to hold the last value that was assigned (*e.g.*, $\texttt{assign\_brake}_{\texttt{last\_set}}$).

The *Temporal Same Actuator* FIDP detects (a) whether the difference between the last and current values requested exceeds the value threshold, *e.g.*, $\mid \texttt{assign\_throttle} - \texttt{set\_throttle}_{\texttt{last\_set}} \mid > \texttt{value\_threshold}_{\texttt{throttle}}$, and (b) whether sufficient time has passed for the previous output to take effect, *e.g.*, $\mid t_{\texttt{now}} - t_{\texttt{last\_throttle}} \mid < \texttt{time\_threshold}$, illustrated by Figure 3.3 **(a)**. It might be the case that both immediate and temporal feature interactions exist with different value thresholds. In model checking, optimizations to handle time could be used, *e.g.*, only storing the differences in time, rather than absolute times.

The *Temporal Conflicting Actuators* FIDP detects whether the current value of X exceeds its threshold of conflict and the last value of Y exceeds its threshold of conflict, and that the time between now and when Y was last set is less than a time threshold, which is illustrated by Figure 3.3 **(b)**. An FIDP for the symmetric case also exists.



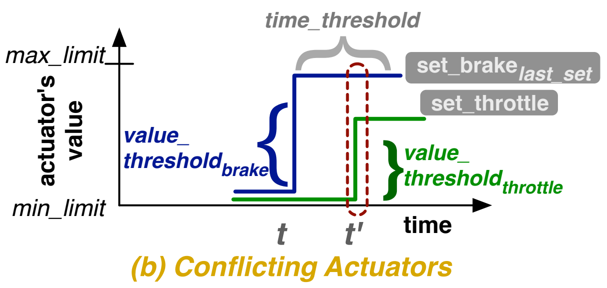


Figure 3.3: Illustration of temporal feature interactions

3.3 Validation of Definition of Feature Interactions

Because the set of actuators that are controlled by automotive active safety features is well-known by the feature designers, I believe it is reasonable to ask these domain experts to provide the elements that are influenced by active safety features (in the system or its environment) as input to the process that creates the FIDPs for feature interaction detection. The list of FIDPs generated and the set of non-proprietary automotive active safety features, described in Appendix A, are both used in my case study shown in Chapter 7. For the features in Appendix A, the actuators Brake and Throttle range from 0 to 100, while actuator Steering has three values: -1 indicating that the vehicle shall turn the wheels to the right, 0 indicating that the wheels shall be centred, and 1 indicating that wheels shall turn to the left. In this dissertation, only immediate feature interactions will be detected and temporal ones are left for future work.

My proposed definition of feature interactions for automotive active safety features, based on the actuators and environmental elements of multiple feature influence, is

Systematic: Because the definition is based on actuators controlled by active safety features and the thresholds at which these actuators interact, the FIDPs for a set of features using these actuators can be created automatically. This is justified by listing the Immediate FIDPs that can be automatically generated, given the actuators and their value thresholds in Table 3.3.

	Actuator	**Value threshold**
Same Actuators	Brake	30
	Throttle	20
	Steering	1
Conflicting Actuators	Brake Throttle	40 30
	Throttle Steering	40 0

Table 3.3: Elements of multiple feature influence and thresholds for case study

A list of actuators and value thresholds would be expected from domain experts, but for my case study in Chapter 7, I selected the value thresholds shown in Table 3.3 as sufficiently different values to identify feature interactions given the actuator's ranges of possible values. Based on the information in Table 3.3, the following list of immediate FIDPs can be automatically generated, with one FIDP created when feature F_1 and feature F_2 both (a) influence the same actuator, and thus, checking for same actuator interactions, or (b) influence the same element in the environment given the actuator's actions, and thus, checking for conflicting actuators interactions:

- G $\neg(\mid \texttt{assign\_Brake}_{\mathsf{F_1}} - \texttt{assign\_Brake}_{\mathsf{F_2}} \mid > 30)$
- G $\neg(\mid \texttt{assign\_Throttle}_{\mathsf{F_1}} - \texttt{assign\_Throttle}_{\mathsf{F_2}} \mid > 20)$
- G $\neg(\mid \texttt{assign\_Steering}_{\mathsf{F_1}} - \texttt{assign\_Steering}_{\mathsf{F_2}} \mid > 1)$
- G $\neg((\texttt{assign\_Brake}_{\mathsf{F_1}} > 40) \wedge (\texttt{assign\_Throttle}_{\mathsf{F_2}} > 30))$ and the symmetric case G $\neg((\texttt{assign\_Brake}_{\mathsf{F_2}} > 40) \wedge (\texttt{assign\_Throttle}_{\mathsf{F_1}} > 30))$
- G $\neg((\texttt{assign\_Throttle}_{\mathsf{F_1}} > 40) \wedge (\texttt{assign\_Steering}_{\mathsf{F_2}} > 0))$ and the symmetric case G $\neg((\texttt{assign\_Throttle}_{\mathsf{F_2}} > 40) \wedge (\texttt{assign\_Steering}_{\mathsf{F_1}} > 0))$

Complete: Because the definition is based on actuators, which are well-known to all feature designers and defined in advance, my definition identifies an interaction every time two features have conflicting requests on these actuators. However, my definition of feature interactions is only complete with respect to the set of actuators and thresholds provided by domain experts, and it does not consider vehicle dynamics.

General: Because the definition is independent from the set of features, my definition remains the same even when features are modified or new features are added to the system. In the traditional approach to identify feature interactions in automotive systems, domain experts list the behaviours in which they expect features to interact, and therefore, the definition would have to change each time a feature is modified or a new feature is added to the system. In contrast, my definition based on actuators would still hold. The justification of the generality of my definition by cases is as follows:

- *Case of feature modified* – Assume that all active safety features, including the one being modified, use the same set of actuators already provided by domain experts. In this case, my definition can be used without change because the list of actuators and thresholds was not modified, and therefore, the FIDPs derived from these actuators and thresholds will still identify all the same and conflicting actuator interactions. In contrast, in the traditional approach, the definition of feature interaction would have to be modified if it relies on the particular behaviour of the feature that was modified.
- *Case of new feature added* – Assume that all active safety features, including the one being added, use the same set of actuators already provided by domain experts. In this case, my definition can be used without change even when new features are added to the system because the list of actuators and thresholds was not modified, and therefore, the same FIDPs will identify all the same and conflicting actuator interactions. Moreover, only the new feature added has to be checked against the features that were already part of the system,

given that the definition of feature interaction did not change, and therefore, the features that were part of the system have already been checked and their interactions identified. In contrast, the traditional approach would have to change the definition of feature interaction, considering any cases in which the behaviour of the newly added feature can interact with all the features that were part of the system, and the analysis will likely have to be completely re-done.

3.4 Related Work

This section gives a brief overview of the definitions that have been proposed in the literature to detect feature interactions in telecommunications, Internet applications and embedded systems.

3.4.1 Definitions in Telecommunications

For telecommunications systems, which can change dynamically as features from different users can join or drop the call as time progresses, the criteria for detection could be *general* (*e.g.,* presence of deadlock/lifelock, ambiguities) or *specific* (*e.g.,* conflicts with respect to shared resources), as described by Keck and Kuehn [110]. My definition falls into the specific criteria, regarding conflicts with respect to shared resources. However, the shared resources for automotive active safety features, *i.e.,* the actuators, are well-known and not likely to change across projects when new features are added into the system. In contrast, in telecommunications, the shared resource is a signal, and the set of these signals might vary between networks where the features are deployed. Also, in the telecommunications domain, the conflicts on shared resources are detected as logical inconsistencies, *e.g.,* $(\mathsf{set\_busy} \wedge \neg \mathsf{set\_busy})$, whereas in the automotive domain, my definition recognizes conflicts using value thresholds, not only for the same actuator, but also for conflicting actuators whose effects are recognized in the environment.

Given the above classification of interactions, there are different ways that the definitions can be expressed, depending on the techniques used for detection of interactions, as well as the language used to describe the features. Calder *et al.* [42] define several options:

- Service and Software Engineering – the application of techniques within the development process to address feature interaction. An example of these techniques is filtering, introduced in Section 2.1.3, which aims to remove combinations that that are unlikely to produce interactions and has been used by Heisel and Souquières [91], Bredereke [33], Keck [109], Kimbler [113], Kimbler *et al.* [114, 115].

- Formal techniques – The application of formal description, modelling and reasoning techniques. These techniques include: classical, constructive, modal and non-monotonic logics, process algebra, finite and infinite state automata, extended state automata, petri-nets, transition systems, and languages such as State Description Language (SDL) [7], State Transition Rules (STR) [191], Promela [93], Z [6] and LOTOS [28]. This work characterizes these methods by three major approaches:
  - Properties Only: The definition of features and that of interactions are expressed in terms of a logic, where normally the interaction takes the form of inconsistency or unsatisfiability. Examples of this approach can be found in papers by Blom *et al.* [26], Boström and Engstedt [29], Felty and Namjoshi [73], Frappier *et al.* [75], Gammelgaard and Kristensen [78], Gibson [81], Rochefort and Hoover [156], Stepien and Logrippo [163].
  - Behaviour Only: The behavioural description of features is defined using variants of automata and transition systems, while the definition of the interaction is expressed in a variety of ways such as deadlock, non-determinism, *etc.* Examples of this approach can be found in papers by Accorsi *et al.* [9], Au and Atlee [15], Bergstra and Bouma [23], Blom [25], Bruns *et al.* [36], Gibson [80], Hall [86], Khoumsi [111], LaPorta *et al.* [120], Nakamura *et al.* [135], Plath and Ryan [147], Pomakis and Atlee [148], Thistle *et al.* [164], Thomas [165], Yoneda and Ohta [190].
  - Properties and Behaviour: The features are described by a behavioural description, while a property specifies the intended functionality of a feature or a set of features. The definition of the interaction is such that the intended functionality of the system does not hold when some combination of features execute together. Examples of this approach can be found in papers by Calder and Miller [44], Capellmann *et al.* [48], Combes and Pickin [59], Kamoun and Logrippo [107], L. de Bousquet *et al.* [64], Plath and Ryan [146], Thomas [165].

My proposed definition uses the approach of properties and behaviour, where properties describe how an unexpected combined behaviour of a pair of automotive feature models can be identified.

3.4.2 Definitions in Internet Applications

Previous work on feature interactions for Internet application has not been as extensive as in the telecommunications domain, and the systems studied are VoIP, e-mail, web services and networked home automation. As described in the corresponding background section in Chapter 2, most of the approaches related to Internet applications require an online component (*e.g.,* [62, 82, 187, 136, 141]) as features come from multiple distributed sources, where the end-user or Internet providers have flexibility to create and modify features. In contrast, because active safety features are unmodifiable and the features to be delivered with the vehicle are fixed at release or retail time, my solution can be implemented offline and it is based on analysis of the design models. Other definitions proposed for Internet applications are based on test case generation (*e.g.,* [87, 53]) which does not detect all feature interactions, unlike my definition for active safety features.

Weiss and Esfandiari defines a feature interaction between web services as an element of multiple feature influence (using my terminology) on a goal graph derived from the goal-oriented requirements [178]. This method would miss feature interactions if the requirements are incomplete or if the goals are not correctly specified. Zheng *et al.* [189] define feature interactions as a general criteria, as defined by Keck and Kehn [110], while my definition uses specific criteria based on actuators of the motion control systems in the vehicle, although both, their and my definition, specify properties in LTL.

Kolberg, Magill, and Wilson consider feature interactions in home automation systems, where appliances are connected to the Internet [116, 183], defining a feature interaction when devices access shared variables with different attributes. Unlike my definition, the shared variables considered in their definition depend on the devices integrated into the home system, and because these appliances come from different vendors, it would not be possible to produce a systematic and complete definition when the shared variables and their attributes are not part of the model of the environment provided.

3.4.3 Definitions in Embedded Systems

Contributions to the feature interaction problem for embedded systems are fairly recent, and only a few contributions can be reported and compared with my proposed definition.

Metzger also defines a feature interaction as an element of multiple feature influence (using my terminology) on an object diagram [131]. However, this definition relies on the existence of an environment model to be combined with the system, so the object diagram can be constructed, whereas my definition does not require a model of the environment. Moreover, this approach is conservative because it does not consider the operational behaviour of each feature, so the definition by Metzger identifies false positives.

Lochau and Goltz propose a test case generation method for feature interaction analysis between Statechart-like behavioural models [124], defining test cases whenever two features access (read or write) to a shared variable. In contrast, my method defines a feature interaction when two features try to change (*i.e.,* write) to the same actuator, but providing the data threshold in which an interaction exist. Moreover, my definition also accounts for interactions that occur in the environment based on changes to actuators that are not related in name, unlike their definition.

D'Souza, Gopinathan *et al.* use concepts of supervisory control theory to detect interactions, based on a notion of blocking supervisor conjunction [67, 68]. Thus, this work uses behavioural descriptions only, whereas my definition describes LTL properties to verify with behavioural models of features.

3.5 Summary

Feature interaction in the automotive domain is a relatively new area of study and will remain of interest in the future as vehicles continue to increase in complexity. To the best of my knowledge, I am the first to propose a systematic, complete and general definition of feature interactions in the automotive domain. My definition is based on the set of actuators controlled by active safety features and results in a set of LTL properties to detect contradictory requests to actuators from models of software features without vehicle dynamics through model checking. My definition requires that the domain experts provide the set of actuators that all influence the same elements in the environment, as well as the thresholds (value and time) that produce a conflict in the environment. Since the set of actuators controlled by active safety features is relatively small and they are well-known to all feature designers, I believe it is reasonable to ask the domain experts for this information. In the rest of this dissertation, only immediate feature interactions will be detected, whereas temporal ones will be left as future work.

Chapter 4

Translating STATEFLOW Feature Design Models to SMV: *mdl2smv*

This chapter describes the translation process from automotive design models created using a subset of MATLAB's STATEFLOW into the input language of SMV. I also summarize the design decisions made during the creation of my translator tool, called *mdl2smv*.

The present chapter is organized as follows. Section 4.1 provides an overview of the process to translate models created in STATEFLOW to the input language of SMV. Section 4.2 lists the syntactic rules used for modelling automotive features in STATEFLOW, as observed in practice within the automotive industry, which define the subset of STATEFLOW supported by my tool. The core of this chapter explains the two main parts of the translation: (1) Section 4.3 through Section 4.6 describe how the different elements of an individual model designed in STATEFLOW are translated into the SMV notation; (2) Section 4.7 describes what is needed to perform the integration process, as a feature interaction can only be detected in the integrated model when features are executing concurrently. Section 4.8 discusses related work.

4.1 Process Overview

This section starts by reminding the reader of the elements of STATEFLOW, which were first described in Section 2.2. The rest of the section provides an overview of the translation process, and introduces two models that will help illustrate the translation as well as the detection of feature interactions in this dissertation: a STATEFLOW model of a simple air conditioning system and a STATEFLOW model of a simple heater system.

A STATEFLOW model is a state machine that can include hierarchical and composite states. The syntax of STATEFLOW is similar to that of Statecharts [89], but with different semantics. Unlike Statecharts, a STATEFLOW model runs in a single thread, so there is no true concurrency. Therefore, AND-state execution is sequential: each sibling of an AND-state is given an order of execution, running one at a time. I call STATEFLOW AND-states *ordered-composition* to differentiate them from AND-states in Statecharts. Non-determinism is avoided because STATEFLOW defines strict ordering rules of execution for siblings within ordered-compositions as well as for transitions, as illustrated in Figure 4.1. MATLAB's STATEFLOW does not have formal semantics, although its behaviour is defined during simulation. A formal definition of the semantics of STATEFLOW is out of the scope of this dissertation, but I follow precisely STATEFLOW's simulation semantics.

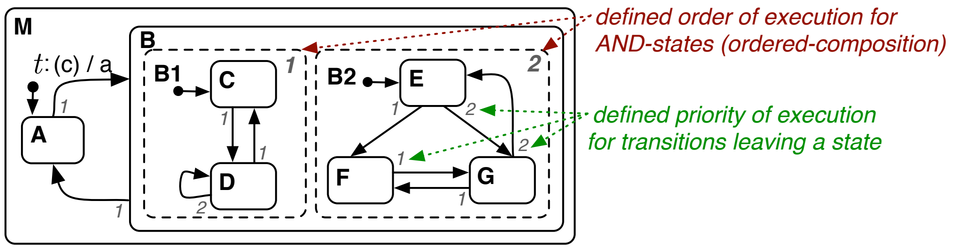

M
B
t: (c) / a
B1
C
1
B2
E
2
A
D
F
G
defined order of execution for AND-states (ordered-composition)
defined priority of execution for transitions leaving a state

Figure 4.1: Strict execution order for ordered-compositions and transitions in STATEFLOW

In addition to the explicit priority of execution associated with transitions leaving a state, transitions whose source is a superstate have higher priority of execution than transitions within a superstate. For instance, the transition exiting B has priority of execution over any other transition within B. If the trigger of the transition exiting B is not satisfied, then STATEFLOW allows a transition to be taken in B1 (respecting the priority of execution for transitions within B1) followed by a transition taken in B2. In STATEFLOW, all the states at the same hierarchy level have the same type of decomposition, *e.g.,* in Figure 4.1, M's main superstate has only OR-states, while B is an ordered-composition.

One can think of the behaviour of an ordered-composition as each sibling executing in a **small-step**, and the execution of all siblings completing in a **big-step**[1]. When an ordered-composition is part of the model and it is executing, a big-step is a sequence of small-steps with one transition taken per small-step. All the siblings in an ordered-composition respond to the same set of inputs. A model is **stable** when the execution of a big-step has been completed (more details are given in Section 4.6), which is the point at which a feature has generated all its outputs for the current inputs and receives new inputs. The produced SMV model includes the macro `stable` (or `sys_stable` for integrated features), which is true when the execution of the big-step is complete.

[1]Big-steps and small-steps are often called macro-steps and micro-steps, respectively. I follow Esmaeilsabzali *et. al* [72] and use big- and small-steps.

The key aspects that make this translation interesting are:

- The sequential execution of ordered-compositions (Section 4.6), which requires that:
    1. each sibling executes following a predefined order, one sibling per small-step;
    2. the inputs are semi-controlled, so all the siblings in the ordered-composition react to the same set of inputs.
- The parallel execution of features in the integrated model (Section 4.7), which requires that:
    1. each feature follows its own execution constraints, such as sequential execution for ordered-compositions;
    2. the inputs are semi-controlled, so all features react to the same set of inputs;
    3. a feature that has fewer siblings in its ordered-composition or contains no ordered-composition has to idle and hold its outputs constant while the other feature in the integrated model completes its execution.
- The use of parameterized events for output requests to actuators (Section 4.3), thus making the request identifiable at the big-step boundary.

Based on the STATEFLOW design models I have observed in practice[2], I define and translate a subset of the STATEFLOW syntax, described in Section 4.2. MATLAB's STATEFLOW stores the model's information in a file with a `.mdl` extension. My tool *mdl2smv* extracts the necessary information from the `.mdl` file and translates a model into the SMV modelling notation, creating a text file with extension `.smv`. I chose SMV because its notation can reflect precisely the semantics of the individual design models in STATEFLOW, and that of the composition of features, which is used during feature interaction detection. As a stand-alone tool, *mdl2smv* is more portable than a plug-in to MATLAB and it can be used without a license for MATLAB. *mdl2smv* is written in the C programming language.

Each STATEFLOW feature model becomes a separate module in SMV. The name of the SMV module is the name of the STATEFLOW subsystem within the SIMULINK model. Although one SMV model is generated from each STATEFLOW model by *mdl2smv*, each translated SMV model includes some elements that makes it "integration-ready". In Section 4.7, I indicate explicitly the elements in a SMV model that are included to aid integration. My tool *Alfie* (described in Chapters 5 and 6) creates the integrated model by taking two `.smv` files as input. During the integration, *Alfie*: *(1)* takes the SMV modules from the two `.smv` files, and *(2)* creates a `main` module, which coordinates the concurrent execution of the two features that are part of the integrated model. The relevant details on integration are given in Section 4.7.

[2]Observed during my visits to General Motors (GM) Research and Development as part of the requirements of my NSERC Industrial Postgraduate Scholarship.

An overview of how the elements of Stateflow are mapped to elements in the SMV notation is as follows:

1. Stateflow variables are declared as SMV variables of the appropriate type.
2. Stateflow states are declared as SMV variables of enumerated type. Hierarchically, the possible values of each superstate variable are the names of its substates.
3. To translate the behavioural description, a Stateflow small-step is mapped into an SMV step. The initial state of each superstate, and the initial values of variables are defined using an `init` operator. In each small-step, one transition is taken, which changes the values of states and variables. Conditional expressions in SMV on input, output and local variables are used to produce the updates to the state and output variables (using the operator `next`).
4. The behaviour on an ordered-composition is captured through a sequence of SMV steps, as several small-steps need to be taken to complete the big-step defined by an ordered-composition.
5. Feature models are integrated by synchronizing the times at which new inputs are generated and received by the models.

To illustrate the mapping from Stateflow into SMV, as well as the integration of SMV modules for feature interaction detection, I use two models: a simplified model of an air conditioning (AC) feature in Figure 4.2 and a simplified model of a heater (HEATER) feature in Figure 4.3. Both feature models share as inputs: the variable e that takes on the values enter and exit, and the variable t that indicates the current temperature, which ranges over the values 0..2. HEATER also includes the Boolean variables B_inc and B_dec to indicate if the button to increase and decrease the desired temperature is respectively pressed or depressed, and the local variable t_want that holds the desired temperature (ranging over 0..2). Both models can set the output variable set_therm to request that the thermostat make a change to the temperature, which ranges over the values 0..2. When AC and HEATER are integrated, a *feature interaction* is detected if contradictory output requests from these features are made to set_therm, which is their shared actuator controlling the room temperature.

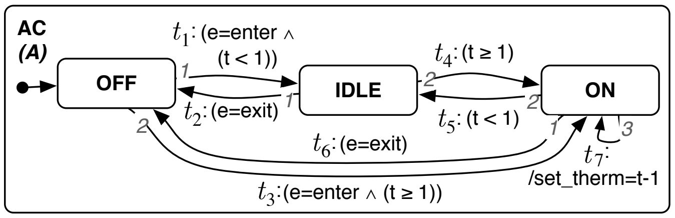


Figure 4.2: Simplified air conditioning (AC) model

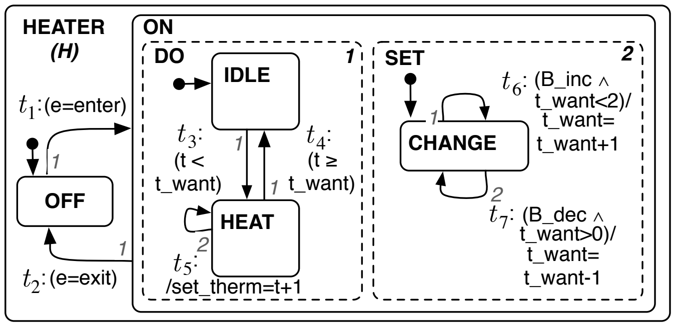


Figure 4.3: Simplified heater (HEATER) model

The translation process to create an SMV module and the process for integration of SMV modules for feature interaction detection are described in the rest of this chapter. The complete translated SMV model for AC is in Figure 4.14 on page 63, while the complete translated SMV model for HEATER appears in Figure 4.11 and Figure 4.12 on pages 58-59. Excerpts from the SMV translations are used in examples throughout this chapter.

4.2 Subset of STATEFLOW Syntax

I define and translate a subset of the STATEFLOW syntax, which matches the subset used in the operational semantics for STATEFLOW, defined separately by Hamon [88] and by Whalen [180], except that my subset does not include junctions and event broadcasting as they are not used when developing active safety systems [16]. There exists a report with design guidelines for modelling with MATLAB's STATEFLOW in the automotive industry [74], however, this report is outdated and refers to an old version of STATEFLOW. As I observed in practice, feature modellers in the automotive domain did not use the following STATEFLOW syntax:

- condition_action's in transitions,
- actions within states,
- connective junctions,
- graphical and MATLAB functions,
- In(state_name) condition function (which is evaluated as true when the state specified as the argument is active),
- temporal conditions using operators such as `after`, `at`, `every` (a type of temporal logic within STATEFLOW, which is different from the one described in Section 2.3.2),
- any notation that could allow event broadcasting.

mdl2smv supports all Stateflow features except the list above. These syntactic elements were not needed when modelling any of the non-proprietary automotive design features part of the "University of Waterloo Feature Model Set" (**UWFMS**), described in Appendix A. If they are desired while designing a feature model, in most cases, an equivalent design can be created without these syntactic elements.

In addition, *mdl2smv* assumes that Stateflow design models adhere to the following minor syntactic modelling rules:

- The type of data and range of values used must be set for all input data, output data, and local data. Thus, every variable has a finite type.
- Strings of any label must not contain quotes.
- Names must not include anything other than numbers, letters, or an underscore (spaces are not allowed as part of the name).
- Each action in a transition must have the form " x = y ; ".
- No two Stateflow models can have the same name (*i.e.,* "Chart", which is the default). The designer must provide a meaningful name.
- Label strings must appear on a single line[3].

4.3 Variable Declarations

One SMV module is created per Stateflow model. At each step in the execution of a Stateflow feature model, only one event occurs at a time. Therefore, the event inputs are modelled using one variable with an enumerated type containing the names of the possible events. The event inputs are kept separated per model, as some events are relevant to a particular feature, *e.g.,* an error signal. The formal parameters of the SMV module include this event variable, as well as all the data that is an external input or output to the Stateflow feature model. All data variables (inputs, outputs and local) are declared as type range of integers or Boolean in SMV, accordingly to the type they have in Stateflow. The declaration of an input variable in the SMV module is prefixed by the keyword `INPUT`, while the declaration of an output is prefixed by the keyword `OUTPUT`. Figure 4.4 illustrates the formal parameters for AC and HEATER, with line numbers referring to Figure 4.14 and Figure 4.11 for each respective model. Event variables names, which are part of the formal parameters, have prefix "A" for AC and "H" for HEATER.

[3]Parsing errors can occur due to newline characters such as \n and \r.

```
MODULE A (Ae, t, set_thermA, sys_stable) {
```

```
MODULE H
  (He,t,B_inc,B_dec,set_thermH,sys_stable) {
```

Figure 4.4: Formal parameters for AC and HEATER

Only local variables are initialized within each SMV feature module, using the `init` operator. *mdl2smv* uses the initialization information from the `.mdl` file, but if none is available, *mdl2smv* initializes the local variables to zero, which is the default value for variables of any type in STATEFLOW. Variables initialized with the `init` operator must be updated using an assignment with the `next` operator. Figure 4.5 illustrates initialization of local variables for HEATER, as there are no local variables initialized in AC.

```
init(t_want):=1;
```

Figure 4.5: Local variables initialization for HEATER

A feature that influences its environment, such as an air conditioning system or an active safety feature, contains multiple kinds of outputs. Some of these outputs are indications to the user, such as warnings or errors occurring in the system. Other outputs are meant to influence the environment by requesting changes to the actuators' values. The SMV model does not include a model of the dynamics of the car, therefore, when a feature produces an output to actuators, it is interpreted as a command to the environment. To model these actuator commands correctly, I use **parameterized events**, which are represented in SMV by *(1)* a *variable* that has the value associated with the command ("`set_`" followed by the actuator name) and *(2)* a *Boolean* that represents the presence or absence of the command (actuator name followed by "`_req`"). The Boolean variable is created during translation by *mdl2smv*. Its intended meaning is that when the "`_req`" Boolean is true, the "`set_`" variable contains an output from the feature. The process to set and reset the Boolean variable will be explained in Section 4.7. For AC and HEATER, the parameterized event for actuator therm is modelled by the output variable set_therm and the Boolean variable therm_req. Each output variable name associated with one of the parameterized events has suffix "A" or "H" (*i.e.,* set_thermA and set_thermH) to differentiate which one refers to AC or to HEATER respectively. In analysis, the environment is left unconstrained, *i.e.,* an output to an actuator is unrelated to the next sensor reading of the environment. The effects of this choice on analysis will be discussed in Chapter 7.

4.4 States

mdl2smv declares one state variable per level of state hierarchy in the Stateflow model. Each state variable has an enumerated type consisting of the names of all of its substates, prefixed with "`s`". For OR-states, the default transition information stored in the `.mdl` file is used to initialize the first active state at the corresponding level of the state hierarchy. For an ordered-composition, there is no default transition, but the state labelled with execution order of 1 is initialized to be the first active state. For example, HEATER contains the ordered-composition ON, which is initialized to DO. The declarations and initializations of state variables for AC and HEATER are shown in Figure 4.6.

```
sA: {sOFF, sIDLE, sON}; /* main superstate AC */
```

```
init(sA):=sOFF;
```

```
sH: {sON, sOFF};      /* main superstate HEATER */
sON: {sDO, sSET};     /* superstate ON */
sDO: {sIDLE, sHEAT}; /* superstate DO */
sSET: {sCHANGE};      /* superstate SET */
```

```
init(sH) :=sOFF;
init(sON) :=sDO;
init(sDO) :=sIDLE;
init(sSET) :=sCHANGE;
```

Figure 4.6: State variables declaration and initialization for AC and HEATER

4.5 Transitions

Transition variables are used to record and report concisely the transitions taken in a path. The name of a transition variable is prefixed by "`Tr`". The information recorded by a transition variable is the name of the transition taken at each step of the execution. Although the name of a transition is not explicit in the graphical view of a Stateflow model, Stateflow includes a number label for each transition in the `.mdl` textual description. Stateflow uses the name of the transition, which is simply a number greater than 0, when reporting syntactic errors found during parsing or simulation. Thus, *mdl2smv* uses this information to create the declarations of transition variables. *mdl2smv* takes the number that identifies a transition from the `.mdl` file, and prefixes it with "`t`" to denote that the identifier refers to a transition. I refer to these transitions as *progressing* transitions in later chapters.

A model with no ordered-compositions needs only to declare one transition variable, whose name is "`Tr`" followed by the main superstate's name and all the transition names in the model as its enumerated values, such as is done for model AC. In contrast, a model with ordered-compositions requires more than one transition variable to report all the transitions taken in a big-step[4]. It is sufficient to declare one transition variable for each sibling in an

[4]This information will be used to report feature interactions in Chapter 6.

ordered-composition as an enumerated type, with the names of the transitions contained in the sibling as values because each sibling is executed at most once in a big-step and, when executing, a sibling can take at most one transition. For example, for the ordered-composition ON in HEATER, two transition variables are declared: `TrDO` and `TrSET`. If the main superstate is not an ordered-composition, a transition variable for the main superstate must also be declared as some transitions are not within any of the ordered-compositions. For instance, for model HEATER, the transition variable `TrH` for its main superstate is declared. Transition variables are initialized to `t0`, meaning 'no transition taken'. There is also an implicit self-looping transition `tn` in each state of a Stateflow model that is considered to execute (and do nothing) when none of the other transitions exiting a state can be executed. I call `t0` and `tn` *non-progressing* transitions. The declarations and initializations of transition variables for AC and HEATER are shown in Figure 4.7.

```
TrA: {t0,tn,t1,t2,t3,t4,t5,t6}; /* Trans within AC */
```

```
init(TrA):=t0;
```

```
TrH: {t0,tn,t1,t2};     /* Trans within HEATER */
TrDO: {t0,tn,t3,t4,t5}; /* Trans within DO */
TrSET: {t0,tn,t6,t7};   /* Trans within SET */
```

```
init(TrH):=t0;
init(TrDO):=t0;
init(TrSET):=t0;
```

Figure 4.7: State variables declaration and initialization for AC and HEATER

As is common in translators to SMV (*e.g.,* [49]), the behaviour of transitions is modelled in SMV using `switch` statements. Nested `switch` statements follow the hierarchy of states in the Stateflow model and check whether the model is in the source state of a transition. Within the `switch`, there is a `case` statement for each state at the corresponding level of the hierarchy. Within each `case`, there is a series of `if-then-else` statements, one per transition, checking whether the event and/or condition of the transition are satisfied.

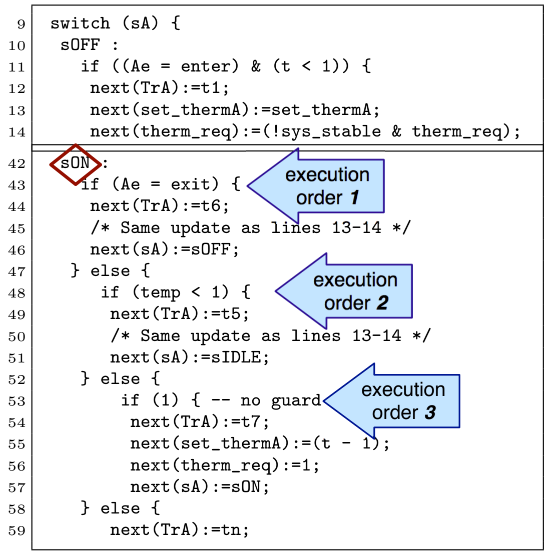
A simple state diagram of only one level of hierarchy would have one `switch` statement. Figure 4.8 shows the translation of an excerpt of AC, containing only OR-states and which has one hierarchy level. The order of the `if-then-else` statements within the `case` corresponds to the transitions' priority of execution, as illustrated by Figure 4.8. Within each `if` statement, there are assignments for the next state, transition and output variables as a result of a transition:

- For OR-states, the assignment corresponds to the state name that is the destination of the transition, *e.g.,* state sA is set to OFF by taking transition `t6`, shown in line 46 of Figure 4.8. For an ordered-composition, the updating process is discussed in Section 4.6.
- There is an assignment for *every* output variable. For variables that are directly changed by an assignment in the transition's action of the Stateflow design model,

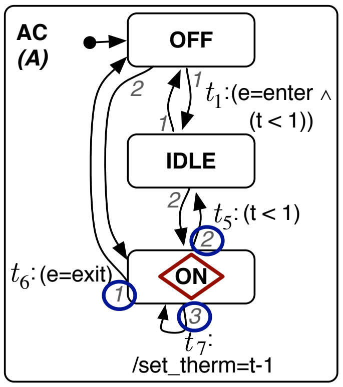
mdl2smv creates its corresponding assignment, *e.g.,* `set_thermA` in line 55 of Figure 4.8. For any output variable whose value is not explicitly defined in the transition's action of the Stateflow model, *mdl2smv* assigns it the value it currently holds, *e.g.,* `set_thermA` in line 13 of Figure 4.8.

- The transition variable is assigned the value of the identifier of the transition taken (prefixed by "`t`") for a model with only OR-states and one level of hierarchy, *e.g.,* `TrA` in line 12 of Figure 4.8.

The last `else` statement identifies the case where none of the transitions exiting the state can be executed, and therefore, a non-progressing transition is taken, *e.g.,* 58 of Figure 4.8. In this case, the state and output variables retain the value they currently hold, and the transition variable is updated to the value `tn`.



```
 9  switch (sA) {
10   sOFF :
11     if ((Ae = enter) & (t < 1)) {
12      next(TrA):=t1;
13      next(set_thermA):=set_thermA;
14      next(therm_req):=(!sys_stable & therm_req);
42   sON :
43     if (Ae = exit) {
44      next(TrA):=t6;
45      /* Same update as lines 13-14 */
46      next(sA):=sOFF;
47    } else {
48       if (temp < 1) {
49        next(TrA):=t5;
50        /* Same update as lines 13-14 */
51        next(sA):=sIDLE;
52     } else {
53         if (1) { -- no guard
54          next(TrA):=t7;
55          next(set_thermA):=(t - 1);
56          next(therm_req):=1;
57          next(sA):=sON;
58     } else {
59        next(TrA):=tn;
```

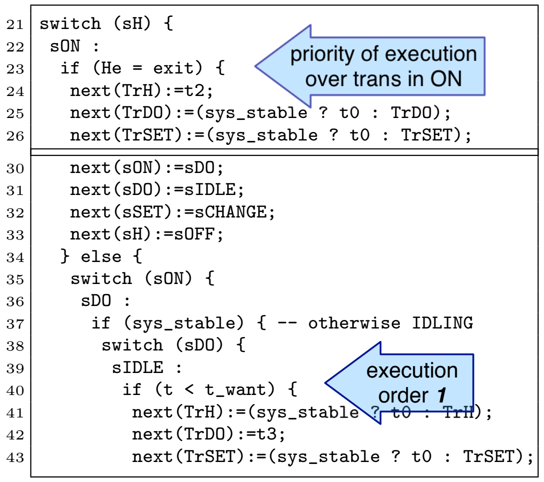


(Equivalent to Figure 4.2)

Figure 4.8: Translation of transitions for AC with OR-states and no hierarchy

Figure 4.9 shows the translation of an excerpt of HEATER, where state hierarchy and ordered-composition are present. As described in Section 4.1, outgoing transitions from a superstate must be listed before inner transitions, in order to follow Stateflow's priority of execution, which is done while creating transitions within a nested `switch` statement (*i.e.,* at a hierarchy level other than one), for instance, as is done for `t2` in line 24 of Figure 4.9. Then, internal transitions are listed; first, the transition with highest priority (*i.e.,* priority 1), and consecutively creating the rest in decreasing priority, as shown for transition `t3` in line 42 of Figure 4.9. Even when hierarchy and ordered-composition are present, the variables for states, outputs and transitions are still updated within each `if` statement as follows:

- For OR-states, the state variables are updated to the state name that is the destination of the transition, updating as well default states for transitions whose source is a superstate, and state's parents hierarchically for transitions whose destination is within a superstate (*e.g.,* lines 30-33 of Figure 4.9). For ordered-compositions, the updating process is discussed in Section 4.6.
- For output variables, the updating process is as described in the case of a state diagram with only one level of hierarchy.
- For the transition taken, its corresponding transition variable is assigned the identifier of the transition taken (prefixed by "`t`"), *e.g.,* `TrH` in line 24 of Figure 4.9. More than one transition variable is declared when a model contains ordered-compositions, but only one transition is taken in a small-step which is updated as explained above. The other transition variables, which are not directly updated in that small-step, must preserve their value, which will be either `t0` (no transition taken in that small-step), or the value of the transition taken in the big-step already. At a big-step boundary, transition variables are reset using an expression of the form `(sys_stable ? t0 : TrDO)`[5], as in line 25 of Figure 4.9 (*i.e.,* when `sys_stable`[6] is true).


```
21 switch (sH) {
22  sON :
23   if (He = exit) {
24    next(TrH):=t2;
25    next(TrDO):=(sys_stable ? t0 : TrDO);
26    next(TrSET):=(sys_stable ? t0 : TrSET);
30    next(sON):=sDO;
31    next(sDO):=sIDLE;
32    next(sSET):=sCHANGE;
33    next(sH):=sOFF;
34   } else {
35    switch (sON) {
36     sDO :
37      if (sys_stable) { -- otherwise IDLING
38       switch (sDO) {
39        sIDLE :
40         if (t < t_want) {
41          next(TrH):=(sys_stable ? t0 : TrH);
42          next(TrDO):=t3;
43          next(TrSET):=(sys_stable ? t0 : TrSET);
```
priority of execution over trans in ON
execution order *1*

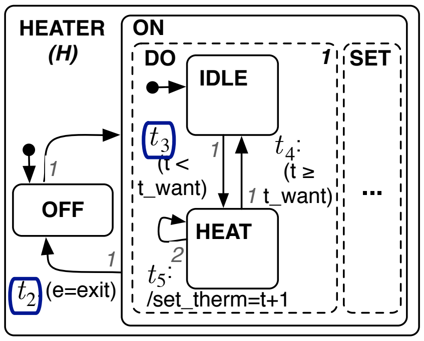


Figure 4.9: Translation of transitions HEATER with hierarchy and ordered-composition

Note that given the condition on `sys_stable` in line 37 of Figure 4.9[7], the reset expressions in lines 41 and 43 are redundant as the transition variables could simply have an

[5]`(a ? b : c)` is the SMV short version of `if (a) { b } else { c }`.

[6]Instead of using the `stable` macro that indicates the model's big-step boundary, *mdl2smv* uses `sys_stable` to make the model "integration-ready". However, if the model is going to be analyzed in isolation, one can simply define `sys_stable=stable` in the `main` module.

[7]This condition on `sys_stable` makes a feature to idle when integrated with another feature that requires more small-steps to complete its execution, as explained in detail in Section 4.7.

assignment to `t0`. However, for simplicity of implementation, these reset expressions are translated as any other reset expressions for transition variables, *e.g.,* the reset expressions in lines 25 and 26 of Figure 4.9.

4.6 Sequential Execution

To translate the behaviour of an ordered-composition in STATEFLOW, one must ensure that

1. the siblings in an ordered-composition execute sequentially in the order assigned to each sibling, and
2. all the siblings that are part of the ordered-composition use the same set of inputs when checking which transitions can be taken.

Thus, the execution of the behaviour of an ordered-composition will take several SMV steps, one SMV step per small-step. Only one transition is taken per small-step. Figure 4.10 shows an example of the execution of the ordered-composition labelled ON in model HEATER, consisting of siblings DO (numbered 1) and SET (numbered 2), therefore, taking two small-steps to execute. Figure 4.11 and Figure 4.12 show the translated SMV module for the feature model HEATER in Figure 4.3, and it will be referred to in the following paragraphs.

First, to model the sequential behaviour of the components of an ordered-composition in STATEFLOW appropriately in a sequence of SMV steps, in each step the parent state variable is updated to the next sibling in the ordered-composition (*e.g.,* line 48 in Figure 4.11 sets the ordered-composition ON to the sibling SET, which has execution order 2). This assignment is illustrated in Figure 4.10 by the change in SMV step i to step i+1. Also, the last component transfers the token for execution back to the first sibling of the ordered execution, which is captured in lines 124, 136 and 147, of Figure 4.12, where the ordered-composition ON is set to the sibling DO, which has execution order 1. This assignment is illustrated in Figure 4.10 by the change in SMV step i+1 to step i+2.

Second, inputs should stay the same for the execution of all siblings of an ordered-composition in STATEFLOW as illustrated by Figure 4.10. At SMV step i, new inputs are received, and DO takes its turn to check if, based on this set of inputs, any of its transitions can be taken. Following the example, transition `t3` was taken, and at SMV step i+1, SET checks if any transition can be taken with the same set of inputs. Finally, at SMV step i+2, new inputs are received. Therefore, in the SMV models, the input variables are semi-controlled by allowing the inputs to change only in the SMV step indicating that the execution of the big-step has completed. This behaviour is modelled in SMV using the

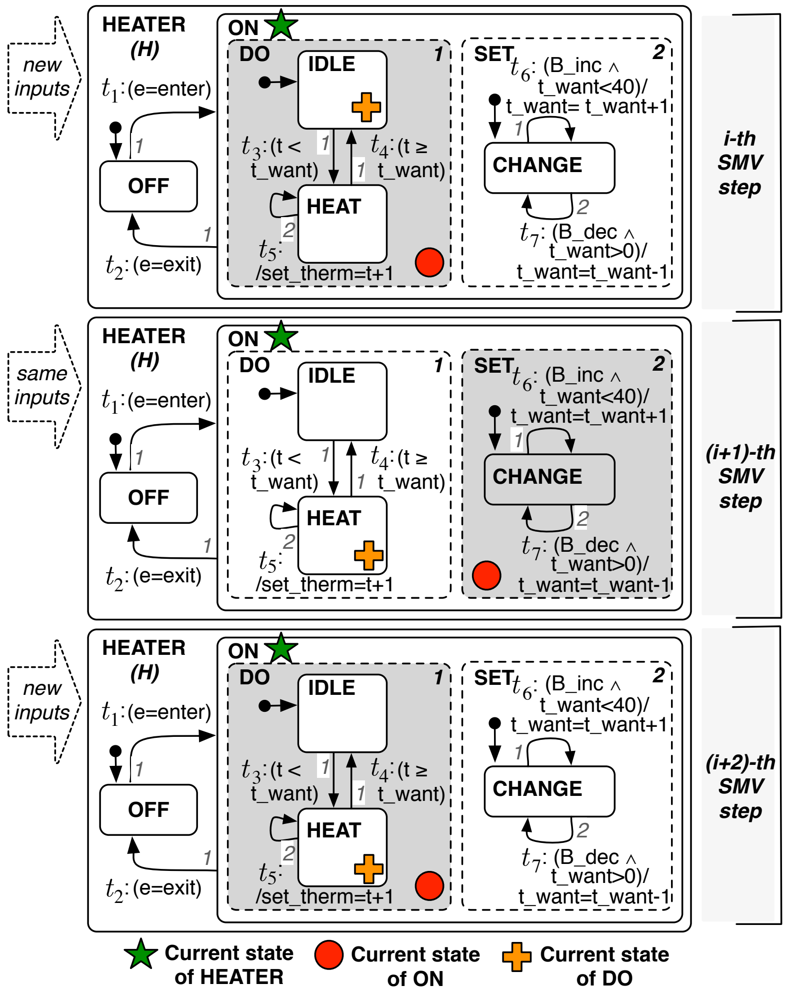


Figure 4.10: Illustration of sequential execution for HEATER

macro `stable`, which defines the boundaries of a big-step per model. By using a macro, this approach does not introduce any extra steps in the computation, and the number of variables in the SMV model does not increase.

At each step of the execution, a big-step varies in size, depending on whether the model includes ordered compositions or not, and the number of siblings in an ordered composition. Therefore, the macro `stable` per model, identifying a big-step boundary, is defined as:

1. *No ordered-compositions are present in a model*: In this case, a model has only OR-states and the boundary of the big-step is always reached after only one transition is taken, so the size of any big-step is one. Therefore, the macro `stable` is defined as 1 (*i.e., true*), as done in line 69 of Figure 4.14, which shows the SMV model for AC.

```
MODULE H
  (He, t, B_inc, B_dec, set_thermH, sys_stable) {
sH: {sON, sOFF};      /* main superstate HEATER */
sON: {sDO, sSET};     /* superstate ON */
sDO: {sIDLE, sHEAT};  /* superstate DO */
sSET: {sCHANGE};      /* superstate SET */
TrH: {t0,tn,t1,t2};     /* Trans within HEATER */
TrDO: {t0,tn,t3,t4,t5}; /* Trans within DO */
TrSET: {t0,tn,t6,t7};   /* Trans within SET */
t_want: 0..2;         /* temp wanted */
therm_req: boolean;  /* change in temp request */
init(sH):=sOFF;
init(sON):=sDO;
init(sDO):=sIDLE;
init(sSET):=sCHANGE;
init(TrH):=t0;
init(TrDO):=t0;
init(TrSET):=t0;
init(t_want):=1;
init(therm_req):=0;
switch (sH) {
 sON :
  if ((He) = (exit)) {
   next(TrH):=t2;
   next(TrDO):=(sys_stable ? t0 : TrDO);
   next(TrSET):=(sys_stable ? t0 : TrSET);
   next(set_thermH):=set_thermH;
   next(t_want):=t_want;
   next(therm_req):=(!sys_stable & therm_req);
   next(sON):=sDO;
   next(sDO):=sIDLE;
   next(sSET):=sCHANGE;
   next(sH):=sOFF;
  } else {
   switch (sON) {
    sDO :
     if (sys_stable) { -- otherwise IDLING
      switch (sDO) {
       sIDLE :
        if ((t) < (t_want)) {
         next(TrH):=(sys_stable ? t0 : TrH);
         next(TrDO):=t3;
         next(TrSET):=(sys_stable ? t0 : TrSET);
         next(set_thermH):=set_thermH;
         next(t_want):=t_want;
         next(therm_req):=(!sys_stable & therm_req);
         next(sDO):=sHEAT;
         next(sON):=sSET;
         next(sSET):=sSET;
         next(sH):=sON;
        } else {
         next(TrH):=(sys_stable ? t0 : TrH);
         next(TrDO):=tn;
         next(TrSET):=(sys_stable ? t0 : TrSET);
         next(t_want):=t_want;
         next(set_thermH):=set_thermH;
         next(therm_req):=(!sys_stable & therm_req);
         next(sDO):=sIDLE;
         next(sON):=sSET;
         next(sSET):=sSET;
         next(sH):=sON;
        }
       sHEAT :
        if ((t) >= (t_want)) {
         next(TrH):=(sys_stable ? t0 : TrH);
         next(TrDO):=t4;
         next(TrSET):=(sys_stable ? t0 : TrSET);
         next(set_thermH):=set_thermH;
         next(t_want):=t_want;
         next(therm_req):=(!sys_stable & therm_req);
         next(sDO):=sIDLE;
         next(sON):=sSET;
         next(sSET):=sSET;
         next(sH):=sON;
        } else {
         if (1) {
          next(TrH):=(sys_stable ? t0 : TrH);
          next(TrDO):=t5;
          next(TrSET):=(sys_stable ? t0 : TrSET);
          next(set_thermH):=(t + 1);
          next(t_want):=t_want;
          next(therm_req):=1;
          next(sDO):=sHEAT;
          next(sON):=sSET;
          next(sSET):=sSET;
          next(sH):=sON;
         } else {
          next(TrH):=(sys_stable ? t0 : TrH);
          next(TrDO):=tn;
          next(TrSET):=(sys_stable ? t0 : TrSET);
          next(t_want):=t_want;
          next(set_thermH):=set_thermH;
          next(therm_req):=(!sys_stable & therm_req);
          next(sDO):=sHEAT;
          next(sON):=sSET;
          next(sSET):=sSET;
          next(sH):=sON;
         }
      }}
     } else { -- IDLING
```

Figure 4.11: SMV model of feature HEATER – Part I

```
      next(TrH):=TrH;
      next(TrDO):=TrDO;
      next(TrSET):=TrSET;
      next(set_thermH):=set_thermH;
      next(t_want):=t_want;

      next(therm_req):=(!sys_stable & therm_req);
      next(sH):=sH;
      next(sON):=sON;
      next(sDO):=sDO;
      next(sSET):=sSET;
     }
    sSET :
     switch (sSET) {
      sCHANGE :
       if ((B_inc) & ((t_want) < (2))) {
               next(TrH):=(sys_stable ? t0 : TrH);
        next(TrDO):=(sys_stable ? t0 : TrDO);
        next(TrSET):=t6;
        next(set_thermH):=set_thermH;
        next(t_want):=(t_want) + (1);
        next(therm_req):=(!sys_stable & therm_req);
        next(sSET):=sCHANGE;
        next(sON):=sDO;
        next(sDO):=sDO;
        next(sH):=sON;
       } else {
        if ((B_dec) & ((t_want) > (0))) {
         next(TrH):=(sys_stable ? t0 : TrH);
         next(TrDO):=(sys_stable ? t0 : TrDO);
         next(TrSET):=t7;
         next(set_thermH):=set_thermH;
         next(t_want):=(t_want) - (1);
         next(therm_req):=(!sys_stable & therm_req);
         next(sSET):=sCHANGE;
         next(sON):=sDO;
         next(sDO):=sDO;
         next(sH):=sON;
        } else {
         next(TrH):=(sys_stable ? t0 : TrH);
         next(TrDO):=(sys_stable ? t0 : TrDO);
         next(TrH):=tn;
         next(t_want):=t_want;
         next(set_thermH):=set_thermH;
         next(therm_req):=(!sys_stable & therm_req);
         next(sSET):=sCHANGE;
         next(sON):=sDO;
         next(sDO):=sDO;
         next(sH):=sON;
       }}
    }}
  }
 sOFF :
  if ((He) = (enter)) {
   next(TrH):=t1;
   next(TrDO):=(sys_stable ? t0 : TrDO);
   next(TrSET):=(sys_stable ? t0 : TrSET);
   next(set_thermH):=set_thermH;
   next(t_want):=t_want;
   next(therm_req):=(!sys_stable & therm_req);
   next(sH):=sON;
  } else {
   next(TrH):=tn;
   next(TrDO):=(sys_stable ? t0 : TrDO);
   next(TrSET):=(sys_stable ? t0 : TrSET);
   next(t_want):=t_want;
   next(set_thermH):=set_thermH;
   next(therm_req):=(!sys_stable & therm_req);
   next(sH):=sOFF;
   next(sON):=sDO;
   next(sDO):=sIDLE;
  }
}
DEFINE stableOFF :=(sH = sOFF);
DEFINE stableON :=(sON = sDO) & (sH = sON);
DEFINE stable :=(((TrH = t1) | (TrH = t2))
                | (stableOFF) | (stableON));
}

MODULE main () {
 He :{enter, exit};
 t :0..2;
 B_inc :boolean;
 B_dec :boolean;
 set_thermH :0..2;
 init(set_thermH) :=0;
 DEFINE sys_stable :=(Hmodule.stable);
 if (next(sys_stable)) {
   next(Ae):={enter, exit};
   next(He):={enter, exit};
   next(t):=0..2;
   next(B_inc):={1, 0};
   next(B_dec):={1, 0};
 } else {
   next(Ae):=Ae;
   next(He):=He;
   next(t):=t;
   next(B_inc):=B_inc;
   next(B_dec):=B_dec;
 }

 Hmodule: H(He, t, B_inc, B_dec, set_thermH,
           sys_stable);
}
```

Figure 4.12: SMV model of feature HEATER – Part II

2. *Ordered-compositions are present in a model*: The definition is given by a recursive function that follows **Algorithm 1**.

An example of a model with both OR-states and ordered-compositions is HEATER, whose macro `stable` is defined by several macros, as shown in lines 174-177 of Figure 4.12. To define the macro `stable`, **Algorithm 1** is first called with the main superstate H and the name "`stable`". Then, following the `if` statement in line 1 of **Algorithm 1**, the macro `stable` for HEATER is defined as `(TrH=t1` $\vee$ `TrH=t2` $\vee$ `stableOFF` $\vee$ `stableON)`. Next, in lines 3-10 **Algorithm 1** checks each child of state H to create macros `stableOFF` and `stableON` appropriately as follows:

- For child OFF, the `if` statement in line 4 is followed as OFF is a basic state, thus defining the macro `stableOFF` as `(sH=sOFF)`.
- For child ON, the `else` statement in line 6 is followed because ON is a superstate, thus the function *define_stable* is called recursively with state ON and name `stableON`.

The call of function *define_stable* with ON follows the `else` statement in line 11 of **Algorithm 1** because the decomposition of state ON is ordered-composition. However, because there are no nested ordered-compositions, and the first sibling of ON, *i.e.,* DO, contains only basic states, lines 20-22 of **Algorithm 1** are followed, defining the macro `stableON` as `(sH=sON` $\wedge$ `sON=sDO)`. By using macros, this approach to defining big-step boundaries does not introduce any extra steps in the computation, and the number of variables in the SMV model does not increase.

4.7 Integrating Features

In the automotive domain, when several active safety features are integrated in a vehicle, in the most general case they may work concurrently, *i.e.,* they all receive the same set of inputs simultaneously. However, each feature reacts to the inputs independently, not communicating with each other directly. Therefore, even though all features receive their inputs synchronously, each feature must preserve its individual STATEFLOW semantics, *e.g.,* ordered-compositions execute their siblings sequentially. Figure 4.13 illustrates the concurrent execution of AC and HEATER, while Figure 4.14 shows the SMV model for the integrated model in Figure 4.13. Designers cannot use STATEFLOW to simulate the combined behaviour of features because there is no operator in STATEFLOW that corresponds to the concurrent behaviour of features integrated in a vehicle. But the integrated behaviour of features can be modelled in SMV, while preserving each feature model's STATEFLOW semantics.

Algorithm 1 – *define_stable*(state, name)

Input: state (state whose macro is defined), name (macro's name: for the main superstate the name is simply `stable`, otherwise, use "`stable`" followed by state's name)
Output: List of macros for model

```
if (state's decomposition is OR) then
    Create macro for state with name, defined as the disjunction of the list of transitions
     within state with the disjunction of the list of macros for the state's children
    for (each child in state) do
        if (child is a basic state) then
            Create macro for child named stable+(child_name), defined as (state=child)
        else ▷ child is a superstate
             ▷ Call function recursively with child
            define_stable(child, stable+(child_name))
        end if
    end for
else ▷ state's decomposition is ordered-composition
    if (nested ordered-compositions) then
        if (innermost ordered-composition's first sibling contains only basic states) then
            Create macro for state with name, defined as the condition, including the state's
            parent, that identifies when the token is in every first sibling hierarchically
        else
            Create macro for state with name, defined as the condition, including the state's
            parent, that identifies when the token is in every first sibling hierarchically, but
            a macro is included for the innermost first sibling instead
             ▷ Call function recursively with innermost first sibling
            define_stable(innermost first sibling, stable+(innermost_first_sibling_name))
        end if
    else ▷ there are no nested ordered-compositions
        if (first sibling contains only basic states) then
            Create macro for state with name, defined as a condition that includes the state's
            parent and (state=first sibling)
        else
            Create macro for state with name, defined as a condition that includes the state's
            parent and (state= macro first sibling)
             ▷ Call function recursively with first sibling
            define_stable(first sibling, stable+(first_sibling_name))
        end if
    end if
end if
```

My tool *Alfie* generates the integrated SMV model, taking as input two SMV models, translated with *mdl2smv*, and creating the `main` module that coordinates the concurrent execution of the SMV models. Inputs and outputs of the STATEFLOW feature model are declared in the `main` SMV module, removing the declarations prefixed with the keywords `INPUT` and `OUTPUT` in the two SMV feature modules[8], and placing them in the `main` SMV module (but without the keywords `INPUT` and `OUTPUT`). The declaration of input and outputs is shown in lines 77-83 of Figure 4.14. In the `main` SMV module, input variables are initialized by taking non-deterministically one of its allowed values given its declaration, so all the possible combinations of input values can be verified by SMV, while output variables are initialized to zero.

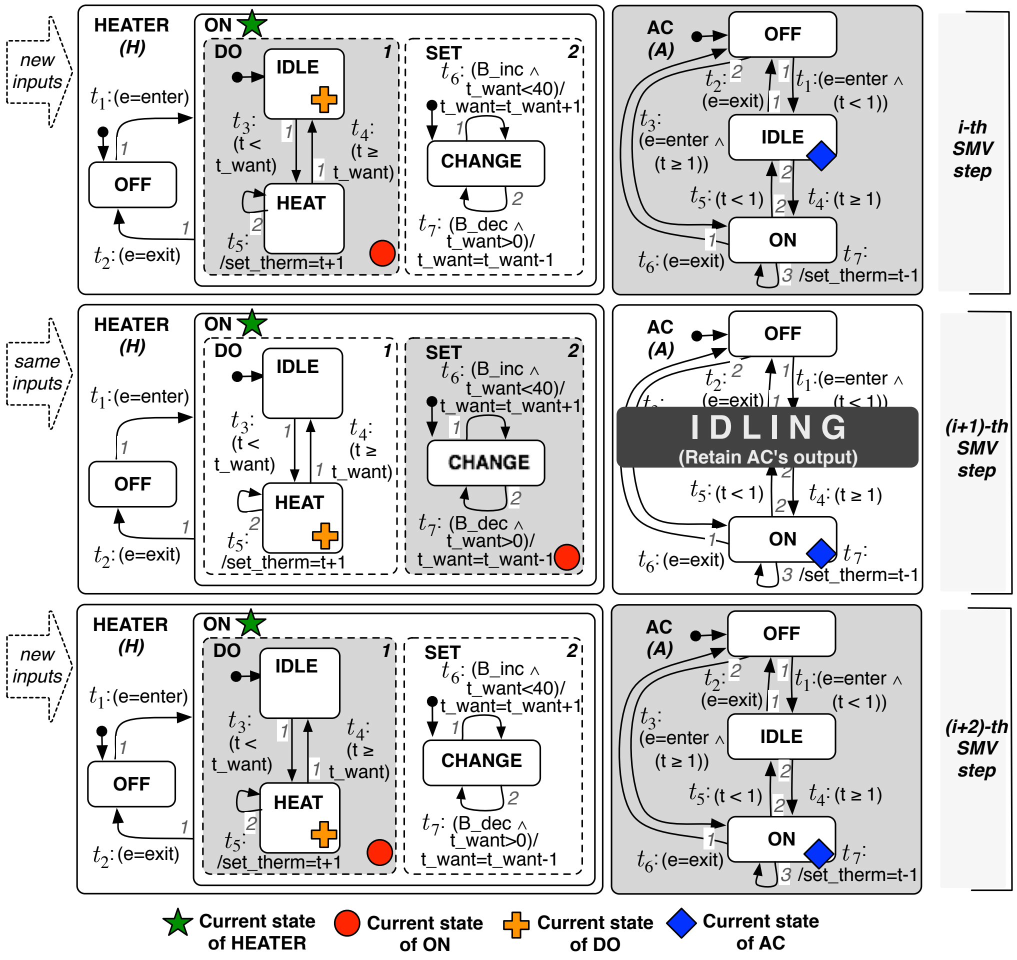


Figure 4.13: Illustration of concurrent execution for AC and HEATER

[8]The SMV features modules of AC and HEATER do not show the declarations with the keywords `INPUT` and `OUTPUT` because the SMV code is meant to show the already integrated model.

```
  1 MODULE A (Ae, t, set_thermA, sys_stable) {
  2 sA: {sOFF, sIDLE, sON}; /* main superstate AC */
  3 TrA: {t0,tn,t1,t2,t3,t4,t5,t6,t7};
  4 therm_req: boolean;   /* change in temp request */
  5 init(sA):=sOFF;
  6 init(TrA):=t0;
  7 init(therm_req):=0;
  8 if (sys_stable) { -- otherwise IDLING
  9  switch (sA) {
 10   sOFF :
 11     if ((Ae = enter) & (t < 1)) {
 12      next(TrA):=t1;
 13      next(set_thermA):=set_thermA;
 14      next(therm_req):=(!sys_stable & therm_req);
 15      next(sA):=sIDLE;
 16    } else {
 17       if ((Ae = enter) & (t >= 1)) {
 18        next(TrA):=t3;
 19        /* Same update as lines 13-14 */
 20        next(sA):=sON;
 21      } else {
 22        next(TrA):=tn;
 23        next(set_thermA):=set_thermA;
 24        next(therm_req):=(!sys_stable & therm_req);
 25        next(sA):=sOFF;
 26      }
 27    }
 28   sIDLE :
 29     if (Ae = exit) {
 30      next(TrA):=t2;
 31      /* Same update as lines 13-14 */
 32      next(sA):=sOFF;
 33    } else {
 34       if (t >= 1) {
 35        next(TrA):=t4;
 36        /* Same update as lines 13-14 */
 37        next(sA):=sON;
 38      } else {
 39        /* Same as tn transition in sOFF */
 40      }
 41    }
 42   sON :
 43     if (Ae = exit) {
 44      next(TrA):=t6;
 45      /* Same update as lines 13-14 */
 46      next(sA):=sOFF;
 47    } else {
 48       if (t < 1) {
 49        next(TrA):=t5;
 50        /* Same update as lines 13-14 */
 51        next(sA):=sIDLE;
 52     } else {
 53        if (t >= 1) {
 54         next(TrA):=t7;
 55         next(set_thermA):=(t - 1);
 56         next(therm_req):=1;
 57         next(sA):=sON;
 58       } else {
 59         /* Same as tn transition in sOFF */ }
 60     }
 61    }
 62   }
 63 } else { -- IDLING
 64   next(TrA):=TrA;
 65   next(set_thermA):=set_thermA;
 66   next(therm_req):=(!sys_stable & therm_req);
 67   next(sA):=sA;
 68 }
 69 DEFINE stable :=1;
 70 }
 71
 72 MODULE H
 73  (He, t, B_inc, B_dec, set_thermH, sys_stable) {
 74   /* Refer to Figures 4.6 and 4.7 */          }
 75
 76 MODULE main () {
 77  Ae :{enter, exit};
 78  He :{enter, exit};
 79  t :0..2;
 80  B_inc :boolean;
 81  B_dec :boolean;
 82  set_thermA :0..2;
 83  set_thermH :0..2;
 84  init(set_thermA) :=0;
 85  init(set_thermH) :=0;
 86
 87  DEFINE sys_stable:=Amodule.stable & Hmodule.stable;
 88  if (next(sys_stable)) {
 89    next(Cevents):={enter, exit};
 90    next(He):={enter, exit};
 91    next(t):=0..2;
 92    next(B_inc):={1, 0};
 93    next(B_dec):={1, 0};
 94  } else {
 95    next(Ae):=Ae;
 96    next(He):=He;
 97    next(t):=t;
 98    next(B_inc):=B_inc;
 99    next(B_dec):=B_dec;
100  }
101  Amodule:A(Ae, t, set_thermA, sys_stable);
102  Hmodule:H(He, t, B_inc, B_dec, set_thermH,
103                         sys_stable);
104 }
```

Figure 4.14: SMV model of integrated AC and HEATER

To define the big-step boundary for an integrated set of features, the macro `sys_stable` is created as the conjunction of the `stable` macro definitions for each feature. The macro `sys_stable` (line 87 of Figure 4.14) is true when all features are ready to begin with new inputs and have produced their outputs for the current set of inputs. To ensure that inputs remain constant when a big-step in the integrated model is executing, a conditional statement in the SMV `main` module is used. The `if-then-else` statement on lines 88–99 of Figure 4.14 forces inputs to stay the same while a big-step is executing and allows inputs to change once the big-step is completed, using the condition `next(sys_stable)`[9].

A feature interaction should be checked when the `sys_stable` macro is true, as this is the point where a summary of the outputs for the big-step in the integrated model is available. But to guarantee that the information about the outputs produced within the big-step is available at `sys_stable`, the feature models in the integrated model must do the following:

1. Whenever a feature model with an ordered-composition is combined with a feature model with fewer siblings in its ordered-composition or with no ordered-composition operators, the latter feature model must idle maintaining the values of its outputs while the former feature model finishes its sequential execution. For example, in Figure 4.13, where AC and HEATER execute concurrently, a set of inputs is received at step i, and HEATER will take two small-steps to process the input since it has to check for transitions that can be taken in all of ON's siblings. In contrast, AC will only take one small-step to process the same input. Therefore, AC must keep its outputs constant while waiting for HEATER to complete the steps of its ordered-composition. AC can only begin executing again at the next big-step, *i.e.,* when HEATER is back in state DO of ON. At step i+2, which is a new big-step, both features can process another set of inputs. Thus, the `sys_stable` macro is used as a condition to make a feature idle when another feature is still processing an input following its ordered-composition, such as taking a transition in the first component of the innermost ordered-composition operator in each feature (line 37 of Figure 4.11) or taking any transition in a feature with no ordered-composition (line 8 of Figure 4.14). If `sys_stable` is not true, the value of a feature's outputs are held constant (as in lines 63–68 of Figure 4.14).

2. Because feature interactions are detected as contradictory output requests to actuators, defined by parameterized events, from the features under consideration, the Boolean variables associated to parameterized events must follow **remainder semantics** [72]. Remainder semantics mean that the value of the event requested per-

[9]This aspect of our approach is similar to that used by Chan *et al.* [49] to model the semantics of RSML, except that the definition of when the inputs are allowed to change is quite different.

sists throughout the big-step, so its value can be available at the big-step boundary. The update of a parameterized event is as follows:

- If a request to an actuator is made as an action associated to the transition taken at the current small-step, then the variable's new value corresponds to the feature's request to the actuator, and the value of the Boolean is set to *true*, *i.e.,* set to 1. For instance, the Boolean variable `therm_req` is set to 1 in line 56 of Figure 4.14 because its corresponding variable `set_thermA` requests the temperature to decrease by one in line 55.
- If no request to an actuator is made in the current transition taken, the value of the variable retains its previous value, unless the current small-step is at `sys_stable`, where the Boolean's value is reset to *false*, *i.e.,* set to 0. This condition is illustrated in line 14 of Figure 4.14. Because the result of the condition is Boolean, it could be simply reduced to `(!sys_stable & therm_req)`, which is equivalent to the expression `(sys_stable ? 0 : therm_req)`.

4.8 Related Work

This section describes related work on translating Stateflow to the input language of formal analysis tools, as well as briefly describing other translators to SMV.

Banphawatthanarak and Krogh [18] translate Stateflow to SMV by creating an SMV module per OR- and AND-state, plus a module to coordinate the status of AND-states (*i.e.,* 'not-active', 'active-active', or 'active-wait'). The conditions for making transitions to each of these states are passed as parameters to the coordinator AND-state module. Our solution is much simpler. In addition, they only support Boolean variables, and do not support transition actions, needed in our features to model actuator request. In [18], each feature is analyzed only in isolation so they did not consider the semantics of the integration of features necessary to detect feature interactions.

Whalen *et al.* [181, 133] developed a framework to translate Stateflow into the input language of various formal methods tools, with NuSMV [54] being one of them. This translator framework, an in-house tool at Rockwell Collins [4], takes as input the specification language Lustre [85], and therefore, a Stateflow model is first imported into Lustre using the Esterel Technologies SCADE Suite [1] or the Reactis [3] tool. Once in Lustre, several transformation passes are performed until a specification is sufficiently close to the target language. However, there are no details as to how the elements of Stateflow are mapped into SMV, and a comparison with *mdl2smv* is not straightforward. Nevertheless, unlike this translator framework that depends on the SCADE suite or Reactis, *mdl2smv* is a stand-alone tool. Also, their translator framework intends to produce models that are analyzed in isolation, thus, not considering the semantics of the integration of models.

There have been several previous efforts to translate Stateflow to the input languages of other model checkers. These efforts either (1) did not include a translation for Stateflow AND-states (*e.g.,* Camera [46] – Stateflow to VHDL, Pingree and Mikk [145] – Stateflow to Spin); or (2) mapped ordered-composition to the Statecharts semantics of AND-states (*e.g.,* Kalita and Khargonekar [106] – Stateflow to STeP); or (3) analyzed features only in isolation and did not consider the semantics of the integration of features necessary to detect feature interactions (*e.g.,* Scaife *et al.* [157] – Stateflow to Lustre). Agrawal *et al.* [12] use a graph rewriting approach to convert a subset of Stateflow to Hybrid Automata. Their approach involves flattening to a much more primitive state machine.

There have been other efforts to translate big-step languages into SMV. However, the semantics of those big-step languages are different from the semantics of Stateflow. Chan *et al.* [49] present the translation of the Requirements State Machine Language (RSML) [123] into SMV. Lu *et al.* [127] describe how to use the semantic decomposition provided by template semantics [137] to parameterize the translation from a requirements notation to the input language of Cadence SMV and NuSMV.

4.9 Summary

This chapter described how to map models designed using a subset of Matlab's Stateflow to the modelling notation of SMV, preserving the semantics of individual Stateflow models, and also preserving the concurrent behaviour of an integrated set of Stateflow features. The key aspects of the translation explained in this chapter are:

- The sequential execution of an ordered-composition modelled in SMV ensures that **(1)** each sibling executes following a predefined order, one sibling per small-step, and **(2)** the inputs are semi-controlled, so that all the siblings in an ordered-composition react to the same set of inputs, using a macro `stable` per feature.
- The parallel execution of features in the integrated model modelled in SMV ensures that **(1)** each feature follows its own execution constraints, such as sequential execution for ordered-compositions, **(2)** the inputs are semi-controlled, so all features react to the same set of inputs using the macro `sys_stable` to define when the inputs can change, and **(3)** a feature that has fewer siblings in its ordered-composition or contains no ordered-composition has to idle and hold its outputs constant while the other features in the integrated model complete their execution, also with the help of the macro `sys_stable`.
- The use of parameterized events for output requests to actuators, thus, making the request available and recognizable at the big-step boundary, defined by `sys_stable`.

Chapter 5

Detecting and Representing all Different Counterexamples to an Invariant: *Alfie*

This chapter describes a novel method for generating a representation of the set of all counterexamples to an invariant for an extended finite state machine (EFSM) model by modifying the property being verified. Verification of invariants using model checking can find errors in a model by generating a counterexample. But because the set of all counterexamples is often too large to generate or comprehend, my method represents the complete set of counterexamples using one representative from each of a set of equivalence classes. These equivalence classes are based on the control states and transitions of the EFSM. I define four different definitions of equivalence. I call these *levels of counterexample equivalence classes*. This summarization of counterexamples is accomplished on-the-fly during iterations of the model checker.

The present chapter is organized as follows. Section 5.1 provides motivation and an overview of the process for finding all equivalence classes of counterexamples to an invariant for an EFSM. In Chapter 6, I extend this method to handle pairs of Stateflow models to detect feature interactions. Section 5.2 defines the generic EFSM formalism used and describes the paths generated by an EFSM. From these paths, the ones that fail an invariant, *i.e.,* the counterexamples, are defined in Section 5.3. A finite representation of the set of counterexamples generated by an EFSM is defined in Section 5.4. Section 5.5 describes the levels of counterexample equivalence classes, while Section 5.6 shows how the equivalence classes for each level can be represented in LTL on-the-fly. I demonstrate the use of my methodology on four individual automotive feature design models (an EFSM with hierarchy) in Section 5.7. Section 5.8 discusses related work.

5.1 Process Overview

An **invariant** is a property that must be true at all times during the execution of the model. When the invariant fails, the model checker generates a counterexample, which is a path showing a behaviour that fails the property. The traditional use of model checking follows a cycle of find bug - fix bug - re-run model checker, until no more counterexamples are found. It can be useful to find multiple or all bugs prior to fixing the model to help isolate the cause of the error [17, 60, 84, 51], as one counterexample by itself may not contain enough information to correctly fix the bug [60]. Moreover, seeing all bugs at once rather than the user iterating over this cycle can improve the user's experience of model checking in a similar way that a compiler returns all (or multiple) errors in one pass [17, 60, 51]. This process also likely reduces the amount of time it takes to create a correct model.

There is no single definition of what a distinct bug in a model is. Complex systems are often described using a form of state machine model because these languages match the internal conceptual models that people use to understand such systems [122]. State machines have explicit control states and transitions that manipulate data in triggers and actions. States and transitions are how a user has partitioned the description of the behaviour of a system. Therefore, the paths through a state machine provide a way of differentiating one bug from another, as all paths through the same control states and transitions are similar to a modeller despite data variations. I focus on extended finite state machine (EFSM) [52] models as many commonly-used modelling languages are based on EFSMs (*e.g.,* Statecharts [89], Specification and Description Language (SDL) [7]). This chapter presents a method to produce automatically one counterexample for each distinct path of an EFSM that fails the invariant, *i.e.,* has a bug.

Detecting multiple counterexamples requires either (1) a change to the model checking engine (*e.g.,* [93, 60, 98]) or (2) the creation of an automatic method that iteratively changes either (a) the model or (b) the property, until a sufficient set of counterexamples is generated. Some model checkers, such as SPIN [93], generate all counterexamples by having the model checking algorithm continue to search the state space after finding a counterexample until no more counterexamples exist. Yet, most of these counterexamples are slight data variations of each other. Moreover, it can take a long time to generate all counterexamples and the result is often too large to comprehend, providing little help in isolating the actual bugs.

Ball *et al.* [17] follow the approach of modifying the model: they use a method that, when a counterexample is generated, identifies a transition in the counterexample that does not appear in the set of correct paths computed by their method so far. Then, the identified transition is removed from the model and their method continues, searching for another counterexample. By changing the model, they eliminate the possibility of finding a different path that includes the removed transition and, therefore, the set of counterexamples generated is not complete, *i.e.,* it may not include a representative of all distinct paths that fail the invariant. If the bug is actually caused by a combination of factors along the path, an incorrect resolution may be chosen, thus, leaving the model still susceptible to failures.

I propose a solution to the problem of generating all counterexamples based on the third possible approach: automatically modifying the property. In contrast to modifying the model checking engine or the model, our method can work with any linear temporal logic (LTL) [128] model checker (explicit or symbolic) and it covers the complete set of counterexamples to an invariant because the model is never changed during the process. However, instead of generating all counterexamples (which can be a set too large to comprehend and generate), one counterexample per equivalence class is generated, returning to the user a representative counterexample in each equivalence class. These equivalence classes can be thought of as meaningful groupings of distinct errors in the model.

My method and tool, both called ***Alfie***, work on-the-fly iterating the model checker: after a counterexample describing a bug is produced, it forces the model checker to search for a distinct EFSM path with a bug, thus, greatly reducing the number of iterations of the model checker compared with approaches that generate all counterexamples (with all data variations) and then summarize the results (*e.g.,* [60]). It is fairly straightforward to create a property that disallows a previously seen counterexample (disjunct the invariant with a property describing the counterexample path seen), thus, the main contribution of my work is the way we ensure that the property we add disallows any counterexample in the same equivalence class to be generated. This "on-the-fly" summarization nature of my method dramatically reduces the time it takes to produce a useful set of counterexamples. The manageability and usefulness of the results generated by *Alfie* are discussed in Section 5.5, where the reduction achieved by my method is illustrated, and in Chapter 7, where my case study with automotive active safety features is described.

Alfie executes a cycle of (1) run model checker to find a counterexample, (2) find the equivalence class of the counterexample according to the chosen level and represent it in LTL, and (3) re-run model checker on the same model with the LTL property that rules out all counterexamples within the same equivalence class. The idea of grouping paths based on control states and transitions of the model's EFSM can be instantiated several ways. I define four different levels, each of which groups the complete set of counterexamples into equivalence classes on-the-fly based on their properties in the EFSM. For example, one level groups together all counterexamples that follow the same sequence of transitions in the EFSM; or if a modeller wants less detail, then another level groups all counterexamples that end at the same control state together. I describe how each level can be useful to isolate the error at different times during the analysis process.

The main contributions of this chapter are:

- Several definitions of equivalence classes of counterexamples. Each definition offers an idea of what is considered a distinct bug in a level, thus avoiding paths that are just slight data variations of each other.
- Representation of these equivalence classes as LTL properties. These representations were challenging to create because of the loops found both in Kripke structures and EFSMs as we consider the infinite paths of a finite EFSM.

5.2 Extended Finite State Machines (EFSMs)

An EFSM is a model with a finite set of control states and labelled transitions, but extended with variables [52]. These variables can be used in triggers or as part of the actions of the transitions. An EFSM is used to describe paths in a system's execution. I explain my method using a generic flat EFSM, but it will be generalized to models with hierarchical control states in Section 5.7. Figure 5.1 is an example of a simple EFSM. Graphically, control states are represented as nodes with transitions as edges, and an initial control state is designated with an edge that has no source control state.

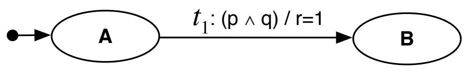


Figure 5.1: Example of simple EFSM

Definition 5.1 *The syntax of an* ***EFSM*** *consists of a tuple*

$$\langle\ CS,\ InitCS,\ V,\ InitV,\ T\ \rangle$$

where

- *CS is a finite set of control states.*
- *InitCS is a set of initial control states (InitCS $\subseteq$ CS).*
- *V is a finite set of typed variables with*
  - *$V = IV \cup OV \cup LV$, where IV is a set of input variables, OV is a set of output (controlled) variables and LV is a set of local variables;*
  - *The sets IV, OV and LV are disjoint.*
- *InitV is a set of sets of initial values for variables. Each set contains one pair for each variable. The first component of the pair is a variable in V, and the second component is the variable's initial assignment drawn from the variable's type.*

- *T is a finite set of **progressing** transitions. Each $t \in T$, with $t = n : s \xrightarrow{(c)/a} s'$, has*
    - *a name n, accessed by function $\mathsf{name}(t)$,*
    - *a source control state $s \in CS$, accessed by function $\mathsf{src}(t)$,*
    - *a destination control state $s' \in CS$, accessed by function $\mathsf{dst}(t)$,*
    - *a label of the form $(c)/a$, where (c) is an optional condition on the variables in V called a guard, accessed by function $\mathsf{guard}(t)$, and a is an optional set of assignments to variables in $(OV \cup LV)$ called actions, accessed by function $\mathsf{actions}(t)$. There are never two assignments to the same variable in a set of actions[1].*

Guards are specified in a language over variables in V that produces Boolean expressions, whereas actions are written in a language over variables that produces assignments to these variables. The particular selection of these languages is not further discussed as my method is independent of these languages. Events, which are often present in EFSMs, can be modelled as Boolean variables with values that do not persist.

Semantically, every control state implicitly has a single self-looping transition, which is taken when no guard on any other transition exiting the state is satisfied. These transitions are called **non-progressing** and have no actions associated with them, but input changes may occur. Non-progressing transitions have no effect on control states and output variables, and every control state has one such transition, thus, ***tn*** is used as the transition name for all non-progressing transitions in the EFSM. A self-looping progressing transition in T with no guard or actions implicitly has the guard *true*, unlike a non-progressing transition, whose guard is the conjunction of the negation of the guards of any other transition exiting the state. Non-progressing transitions ensure that at every configuration there is a next configuration. In this dissertation, I use the name of the transition when referring to the transition itself. Therefore, I use the phrase "the transition t_1" to mean "the transition with name t_1".

To define the behaviour of an EFSM formally, let's first define a configuration of an EFSM as follows to represent a moment in the execution of an EFSM.

Definition 5.2 *A **configuration** σ of an EFSM is a triple $\langle s, n, val\rangle$ that consists of*

- *s, a control state in CS,*
- *n, the name of the last transition taken, with value tn for non-progressing transitions or $\mathsf{name}(t)$ for progressing transition t in T,*
- *val, a set of pairs, where the first component of each pair is a variable in V, and the second component is the variable's assignment from the variable's finite type.*

[1]This restriction is to avoid race conditions in this simple model.

There is a finite number of possible configurations, although they might not all be reachable.

Definition 5.3 *The set of* ***initial configurations*** σ_{init} *of an EFSM consists of triples* $\langle s_{init}, n_{init}, val_{init}\rangle$*, where*

- $s_{init} \in$ *InitCS,*
- $n_{init} \in (T \cup \{tn\})$ *(no restriction on transitions taken),*
- $val_{init} \in$ *InitV, a set of initial assignments to variables.*

To define the next configuration during the execution of a model, the next step relation is introduced.

Definition 5.4 *The* ***step relation*** δ *defines the next step of the model's execution as* $\delta(\sigma_X, \sigma_Y)$*, with* $\sigma_X = \langle s_X, n_X, val_X\rangle$ *and* $\sigma_Y = \langle s_Y, n_Y, val_Y\rangle$*, where*

for a progressing transition, $t \in T$ |

- $n_Y =$ *name(t),*
- $s_X =$ *src(t),*
- $s_Y =$ *dst(t),*
- $val_X \in V$ *is a set of assignments to variables satisfying the conditions in* ***guard****(t),*
- val_Y *is a set of assignments to variables, where the assignment of local and output variables to values are defined by* ***actions****(t) using the values of* val_X*. Any local or output variable that is not assigned explicitly a value in* ***actions****(t) keeps its value in* val_X*. Inputs may change their values non-deterministically.*

or

for a non-progressing transition, $n_Y = tn$ |

- $s_X = s_Y$
- $\forall\ t \in T \bullet (s_X =$ *src(t))* $\Rightarrow val_X$ *does not satisfy conditions in* ***guard****(t),*
- val_Y *is a set of assignments to variables, where the assignment to local and output variables are the same as in* val_X*, while input variables may change their values non-deterministically.*

Example: For the model in Figure 5.1, a step of execution occurs when the model is in source control state **A** and the input variables p and q have the value *true*. Then, transition t_1 is taken, leading the model to control state **B**, and changing the value of the controlled variable r to 1.

Definition 5.5 *The set of paths of an EFSM is called* ***AllPaths*** *and defined as:*

$$AllPaths = \{ \langle \sigma_0, \sigma_1, \cdots, \sigma_n, \cdots \rangle \mid \sigma_0 \in \sigma_{init} \wedge \forall i \geq 0 \bullet \delta(\sigma_i, \sigma_{i+1}) \}$$

Even though there is a finite number of configurations (because the number of states and transitions is finite and the domains from which the variables take values are finite), a single path is infinite because every configuration always has a next configuration. Moreover, the set of all paths in the EFSM can have an infinite number of elements since there can be an infinite number of iterations of the loops that are part of the model.

For the flawed model of an air conditioning (AC) system in Figure 5.2 (introduced in Chapter 1 and repeated here for convenience), some examples of paths in *AllPaths* are shown in Table 5.1, with loops indicated by a bar at the left.

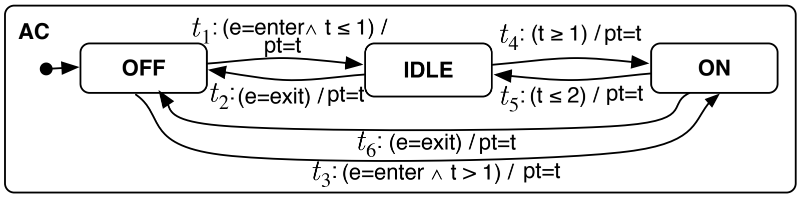


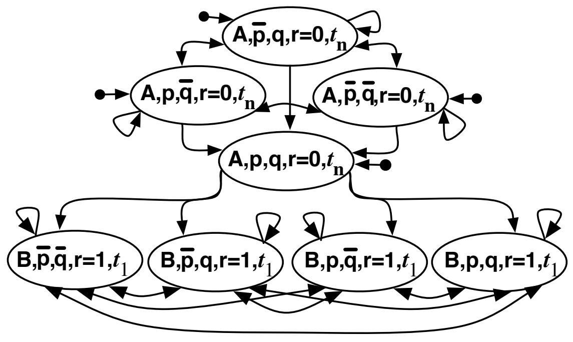
Figure 5.2: Example EFSM of a flawed air conditioning (AC) model

path **p_1**	path **p_2**	path **p_3**
⟨(OFF, tn, e=exit, t=0, pt=0), (OFF, tn, e=enter, t=1, pt=0), (IDLE, t_1, e=exit, t=0, pt=1), (OFF, t_2, e=enter, t=1, pt=0), $\cdots$⟩,	⟨(OFF, tn, e=enter, t=0, pt=0), (IDLE, t_1, e=exit, t=1, pt=0), (OFF, t_2, e=enter, t=0, pt=1), (IDLE, t_1, e=exit, t=1, pt=0), (OFF, t_2, e=exit, t=0, pt=1), $\cdots$⟩,	⟨(OFF, tn, e=enter, t=1, pt=0), (IDLE, t_1, e=enter, t=1, pt=1), (ON, t_4, e=exit, t=1, pt=1), (OFF, t_6, e=enter, t=0, pt=1), $\cdots$⟩,
path **p_4**	path **p_5**	path **p_6**
⟨(OFF, tn, e=enter, t=1, pt=0), (IDLE, t_1, e=enter, t=1, pt=1), (ON, t_4, e=enter, t=1, pt=1), (IDLE, t_5, e=enter, t=1, pt=1), (ON, t_4, e=enter, t=1, pt=1), $\cdots$⟩,	⟨(OFF, tn, e=enter, t=2, pt=0), (ON, t_3, e=enter, t=2, pt=2), (IDLE, t_5, e=enter, t=1, pt=2), (ON, t_4, e=exit, t=0, pt=1), $\cdots$⟩,	⟨(OFF, tn, e=enter, t=2, pt=0), (ON, t_3, e=enter, t=2, pt=2), (IDLE, t_5, e=enter, t=2, pt=2), (ON, t_4, e=enter, t=1, pt=2), $\cdots$⟩,
path **p_7**	path **p_8**	
⟨(OFF, tn, e=enter, t=0, pt=0), (IDLE, t_1, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=2, pt=2), (IDLE, t_5, e=exit, t=1, pt=2), (OFF, t_2, e=enter, t=0, pt=1), $\cdots$⟩,	⟨(OFF, tn, e=enter, t=2, pt=0), (ON, t_3, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, e=enter, t=0, pt=1), (IDLE, t_5, e=enter, t=1, pt=0), $\cdots$⟩,	

Table 5.1: Example of paths in *AllPaths* for AC shown in Figure 5.2

5.2.1 EFSM as Kripke Structure (KS)

The meaning of an EFSM can be described in a Kripke structure (KS). A KS is the model usually used to represent a system for a model checking algorithm. Each KS state represents a configuration of the model, as a σ in Definition 5.2. In a KS, the control states of an EFSM are typically modelled as a variable with the control state names as values. The other EFSM variables are modelled as KS variables of appropriate finite types. Figure 5.3[2] shows the KS for the EFSM in Figure 5.1, repeated here for convenience. In this example, the EFSM has two control states, whereas the Kripke structure has 8 KS states. Transition t_1 in Figure 5.1 corresponds to four transitions in the KS. One can see that the control states of the EFSM are abstractions created by the modeller to group together a set of past behaviours that have the same set of possible future behaviours. The KS state space is the reachable set of KS states within the cross product of the possible values of all the KS variables. The KS state space is usually significantly larger than the set of control states of the EFSM.



```
MODULE main() {
 tr : {tn, t1};
 cs : {A, B};
 p, q, r : boolean;
 init(tr) := tn;
 init(cs) := A;
 init(r) := 0;

switch(cs){
 A: if (p & q) {
     next(tr):=t1;
     next(cs):=B;
     next(r)=1;
   } else {
     next(tr):=tn;
     next(cs):=cs;
     next(r)=r;
   }

 B: {
     next(tr):=tn;
     next(cs):=cs;
     next(r)=r;
   }

}
```

Figure 5.3: Kripke structure and SMV model for EFSM in Figure 5.1

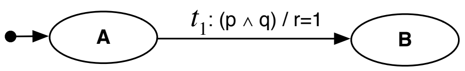


Figure 5.1: Example of simple EFSM (from page 70)

A KS is the underlying structure of a SMV model. Figure 5.3 also shows an SMV model of the KS of Figure 5.1. In the SMV model, the variable name ***cs*** has the value of the current control state of the configuration. The variable ***tr*** has the value of the name of

[2] I use $\bar{\mathsf{p}}$ to mean p=0, *i.e.,* p is false.

the last transition taken in the model[3]. To facilitate the representation of configurations and to make LTL properties easier to understand in this dissertation, in the text I use directly the values of the control states and the names of the transitions instead of relating them to their variable names *cs* and *tr*, *i.e.,* instead of $tr = t_1$, I will write t_1. In the SMV model of Figure 5.3, the variables p and q are input to the model and they are allowed to change non-deterministically at each step in the model's execution (*i.e.,* there are no next statements for these variables), whereas r is a controlled variable. The step relation δ is implemented in SMV by the `next` statements that are part of the model. LTL properties for a model checker are written in terms of KS variables. The counterexamples generated by SMV are in terms of configurations (*i.e.,* KS states).

5.3 Counterexample Paths

From all the paths of an EFSM, I am interested in the paths in which an invariant property fails. An invariant (**inv**) is a predicate on configurations, which should be true in all configurations of all paths. If an invariant fails, then an error must have occurred. A counterexample is any path of the model that contains a configuration σ_k, for some k, in which the invariant fails. The set ***CE*** is defined as:

$$CE = \{\ \langle \sigma_0, \sigma_1, \cdots, \sigma_k, \cdots \rangle \mid \langle \sigma_0, \sigma_1, \cdots, \sigma_k, \cdots \rangle \in AllPaths \wedge \neg \mathbf{inv}(\sigma_k)\}$$

As explained in Section 2.3, a model checker can return as a counterexample: 1) a finite prefix of a path in *CE* or 2) a finite prefix with a cycle at the end.

Example: For model AC in Figure 5.2, an invariant property to check is that 'the previous temperature has to be less than or equal to 1 for AC to be in IDLE, unless AC is in control state OFF', formally described in LTL as: $\mathbf{inv} = (((\mathsf{pt} \leq 1) \leftrightarrow \mathsf{IDLE}) \vee \mathsf{OFF})$. However, for the property G(**inv**), several counterexamples can be generated by a model checker, which the modeller can think of as distinct errors. Two of these counterexamples for AC are shown next, which refer to the paths shown in Figure 5.1 and are marked in the AC model in Figure 5.4. In Section 5.5, I will introduce a method to generate and represent all the distinct errors in a model.

(1) Path **p_3** has $\neg\mathbf{inv}(\sigma_2)$ where σ_2=(ON,t_4,e=exit,t=1,pt=1), thus, it reports an error that occurs when reaching ON while previous temperature is 1 because the condition on transition t_4 is $\geq$ instead of $>$;

[3] The transition name may not be necessary for the computation of the counterexamples. However, because I make use of them in the on-the-fly grouping method presented in Section 5.6, I include them in the model. They cannot be implemented as a macro because, in Cadence SMV, the macro may not appear in counterexamples due to cone of influence reduction.

(2) Path **p_5** has $\neg\mathbf{inv}(\sigma_3)$ where σ_3=(IDLE,t_5,e=enter,t=1,pt=2), thus, it shows an error that occurs when reaching IDLE while previous temperature is 2 because the condition on the transition t_5 is set to 2 instead of 1.

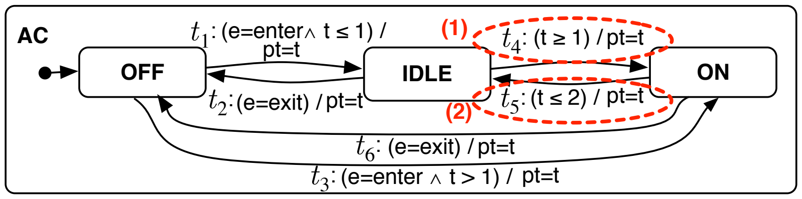


Figure 5.4: Example EFSM of an air conditioning (AC) model

5.4 Failed Invariant Paths (FIPaths)

A counterexample returned by the model checker is a finite path (represented as a finite prefix with a cycle at the end, as described in Chapter 2). However, for generality, I consider the whole set *CE* of infinite paths when defining my equivalence classes because, when asking for another counterexample, the model checker can generate one seen before with the same loop iterated. Therefore, I define the levels of equivalence classes in terms of the set ***FIPaths*** (failed invariant paths), a finite representation of *CE*. *FIPaths* is a finite set of finite paths. *FIPaths* is defined as follows, with every function of the definition illustrated in Figure 5.5:

$$FIPaths = \{q \mid \exists c \in CE \bullet \\ q = reduce\_vals(reduce\_init\_config(reduce\_config\_loops(trunc(progress(c)))))\}^4$$

where:

- *progress*(c): Removes all non-progressing transitions from c to avoid stuttering, except for a constant loop of non-progressing transitions that might appear at the end of the path if the model reaches a final control state. The inputs in the last configuration of a sequence of non-progressing transitions are the only ones that might cause the error and these are copied back to the configuration with the progressing transition just before the sequence begins. For example, the configurations σ_{10} and σ_{11} in path c of Figure 5.5 contain non-progressing transitions, and therefore, they are eliminated from the path by *progress*, and the inputs of σ_{11} are copied to configuration σ_9.

[4] It might appear easier to swap functions *trunc* and *progress* in the definition to avoid dealing with loops at the end. However, as it will be seen in Section 5.6, this order is more convenient for implementation.

- *trunc*(c): Creates the subpath of c that ends in the first configuration that fails the invariant. My analysis concentrates on paths to the first configuration where the invariant fails because any other configuration that fails the invariant after the first one could have been caused by the first error. For example, path c in Figure 5.5 is truncated at configuration σ_8, which is the first configuration in c that fails the invariant.

- *reduce_config_loops*(c): Removes all configuration loops from c. A **configuration loop** is one that reaches the same configuration more than once in c. These loops are unnecessary because the path without these loops has all the steps that cause the error. For example, the configuration loop $\langle\sigma_1,\sigma_2,\sigma_3\rangle$ is the same as the configuration loop $\langle\sigma_3,\sigma_4,\sigma_5\rangle$, therefore, the loops are removed as the path does not lose any information to reach a failed invariant.

- *reduce_init_config*(c): Removes from c a loop that starts at an initial configuration and reaches another initial configuration, where an initial configuration is described in Definition 5.3. This loop contains excessive information because the bug can be reached without traversing this loop. For example, reaching a second initial configuration like σ_2 does not contribute to finding a bug in the model, and therefore, the initial configuration loop $\langle\sigma_1,\sigma_2\rangle$ is eliminated.

- *reduce_vals*(c): Removes from c: (1) the value of the transition name in the first configuration and (2) the input variables in the final configuration of the path. The transition name contains the last transition taken and the inputs to the model are used to calculate the next configuration (not the current configuration) and therefore neither values contribute to the computation that fails the invariant.

I also use *FIPaths* as a function that takes an element of *CE* and maps it to an element of *FIPaths*, *i.e., FIPaths*(c) produces the element of *FIPaths* that c maps to.

More concretely, I illustrate the use of the definition with the following counterexample for model AC in Figure 5.2:

$$\begin{array}{rl}
 & \langle(\mathsf{OFF},\ tn,\ \mathsf{e{=}enter},\ \mathsf{t{=}1},\ \mathsf{pt{=}0}),\\
\big| & (\mathsf{IDLE},\ t_1,\ \mathsf{e{=}enter},\ \mathsf{t{=}0},\ \mathsf{pt{=}1}),\\
\big| & (\mathsf{IDLE},\ tn,\ \mathsf{e{=}enter},\ \mathsf{t{=}0},\ \mathsf{pt{=}0}),\\
\big| & (\mathsf{IDLE},\ tn,\ \mathsf{e{=}enter},\ \mathsf{t{=}1},\ \mathsf{pt{=}0}),\\
 & (\boldsymbol{\mathsf{ON}},\ t_4,\ \boldsymbol{\mathsf{e{=}exit}},\ \boldsymbol{\mathsf{t{=}2}},\ \boldsymbol{\mathsf{pt{=}1}}),\\
 & (\mathsf{OFF},\ t_6,\ \mathsf{e{=}enter},\ \mathsf{t{=}1},\ \mathsf{pt{=}2}),\\
 & \cdots\rangle
\end{array}$$

with (a) a loop of non-progressing transitions indicated by a bar at the left, eliminated by *progress*, (b) the configuration that fails the invariant in bold, with any configuration after that one truncated by *trunc*, and (c) the application of *reduce_vals* to the resulting subpath, therefore generating the following element of *FIPaths* (where *reduce_config_loops* and *reduce_init_config* had no effect on this path):

$$\begin{aligned}\langle(&\mathsf{OFF}, && \text{e=enter, t=1, pt=0}),\\ (&\mathsf{IDLE}, t_1, && \text{e=enter, t=1, pt=0}),\\ (&\mathbf{ON}, t_4, && \mathbf{pt=1})\rangle.\end{aligned}$$

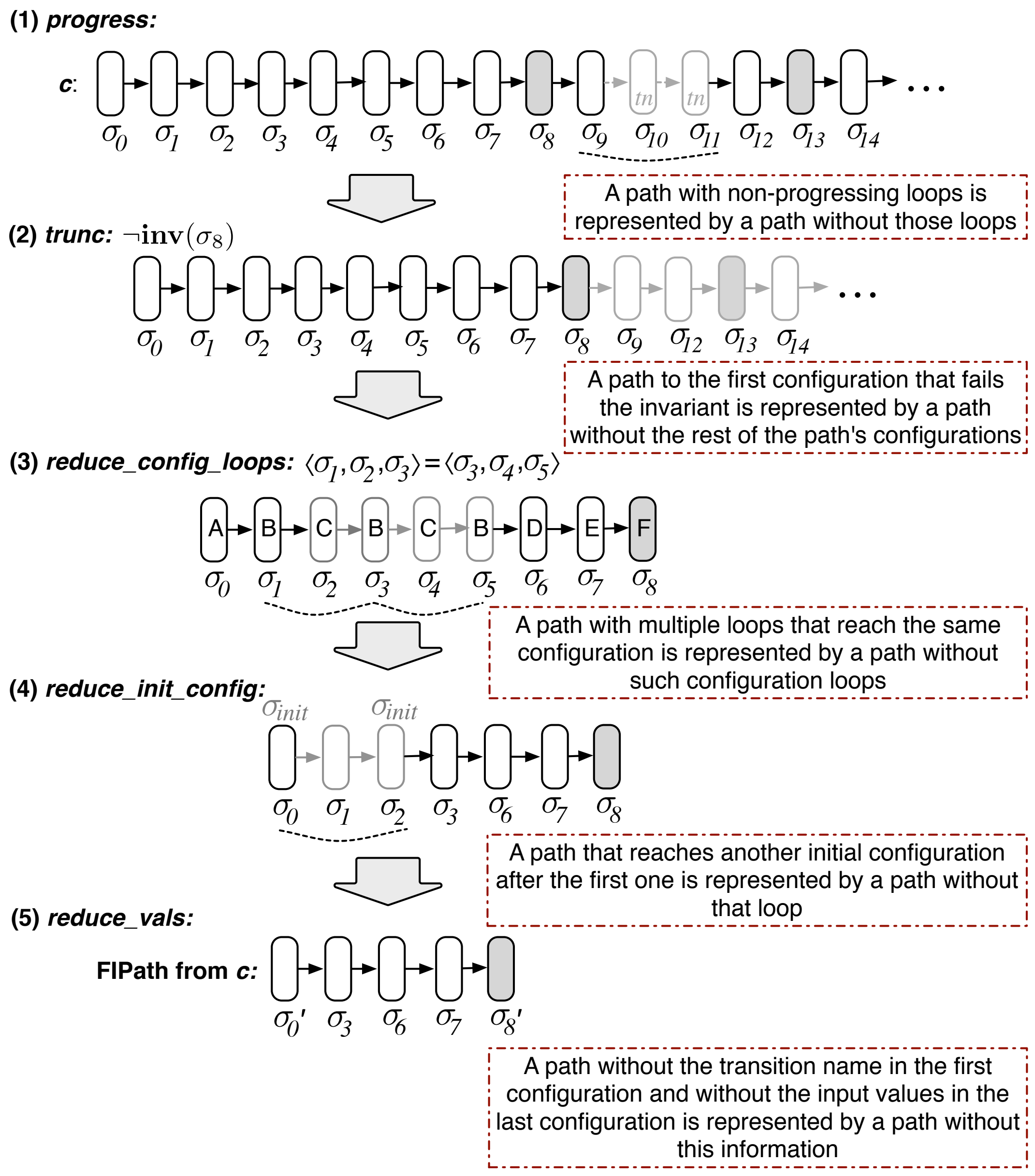
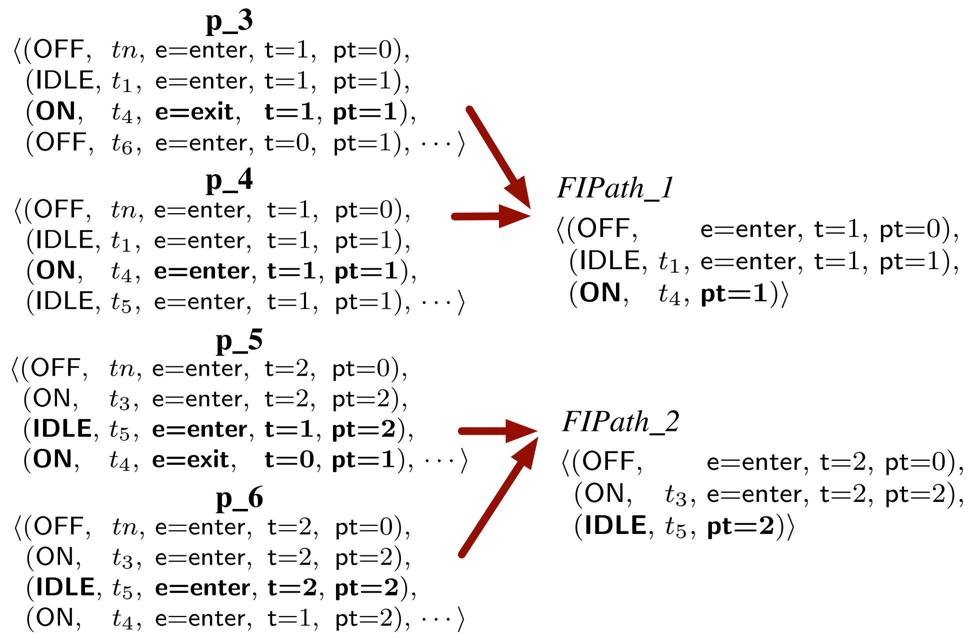

(1) progress:
c:
tn
tn
σ0 σ1 σ2 σ3 σ4 σ5 σ6 σ7 σ8 σ9 σ10 σ11 σ12 σ13 σ14
A path with non-progressing loops is represented by a path without those loops
(2) trunc: ¬inv(σ8)
σ0 σ1 σ2 σ3 σ4 σ5 σ6 σ7 σ8 σ9 σ12 σ13 σ14
A path to the first configuration that fails the invariant is represented by a path without the rest of the path's configurations
(3) reduce_config_loops: ⟨σ1,σ2,σ3⟩=⟨σ3,σ4,σ5⟩
A B C B C B D E F
σ0 σ1 σ2 σ3 σ4 σ5 σ6 σ7 σ8
A path with multiple loops that reach the same configuration is represented by a path without such configuration loops
(4) reduce_init_config:
σinit σinit
σ0 σ1 σ2 σ3 σ6 σ7 σ8
A path that reaches another initial configuration after the first one is represented by a path without that loop
(5) reduce_vals:
FIPath from c:
σ0' σ3 σ6 σ7 σ8'
A path without the transition name in the first configuration and without the input values in the last configuration is represented by a path without this information

Figure 5.5: Paths in *CE* are represented by elements of *FIPaths*

The set of configurations is finite because the number of states and transitions is finite plus the domains from which the variables can take values are also finite. The elements of *FIPaths* are finite sequences of configurations because paths end at the first configuration that fails the invariant and contain only one instance of any configuration loop. The set *FIPaths* is finite because a path is truncated and any instance of a configuration loop is removed.

One element of *FIPaths* is generated from every path in *CE*. Figure 5.6 illustrates the mapping of elements of *CE* to elements of *FIPaths* with respect to the paths shown in Figure 5.1, where configurations that fail the invariant are in bold.



p_3
⟨(OFF, tn, e=enter, t=1, pt=0),
(IDLE, t_1, e=enter, t=1, pt=1),
(**ON**, t_4, **e=exit**, **t=1**, **pt=1**),
(OFF, t_6, e=enter, t=0, pt=1), ⋯⟩
p_4
⟨(OFF, tn, e=enter, t=1, pt=0),
(IDLE, t_1, e=enter, t=1, pt=1),
(**ON**, t_4, **e=enter**, **t=1**, **pt=1**),
(IDLE, t_5, e=enter, t=1, pt=1), ⋯⟩
p_5
⟨(OFF, tn, e=enter, t=2, pt=0),
(ON, t_3, e=enter, t=2, pt=2),
(**IDLE**, t_5, **e=enter**, **t=1**, **pt=2**),
(**ON**, t_4, **e=exit**, **t=0**, **pt=1**), ⋯⟩
p_6
⟨(OFF, tn, e=enter, t=2, pt=0),
(ON, t_3, e=enter, t=2, pt=2),
(**IDLE**, t_5, **e=enter**, **t=2**, **pt=2**),
(ON, t_4, e=enter, t=1, pt=2), ⋯⟩
FIPath_1
⟨(OFF, e=enter, t=1, pt=0),
(IDLE, t_1, e=enter, t=1, pt=1),
(**ON**, t_4, **pt=1**)⟩
FIPath_2
⟨(OFF, e=enter, t=2, pt=0),
(ON, t_3, e=enter, t=2, pt=2),
(**IDLE**, t_5, **pt=2**)⟩

Figure 5.6: Paths in *CE* that become elements of *FIPaths*

5.5 Counterexample Equivalence Classes

Within the general idea of grouping counterexamples based on control states and transitions, there are more precise groupings that may be useful at different times during the analysis process. These groupings define what one can consider to be distinct bugs. I chose these levels of counterexample equivalence classes based on what is deemed as relevant and useful in the literature and my case studies, but other levels could be defined.

The following notation is used to describe the levels of equivalence classes, where by definition, if $p = \langle \sigma_0, \cdots, \sigma_k \rangle \in \mathit{FIPaths}$ then it is always holds that $\neg\mathbf{inv}(\sigma_k)$:

- *FICS*: The set of control states that are in a reachable configuration in which the invariant fails.

$FICS$={ $s_k \mid \exists\ p = \langle\sigma_0,\cdots,\sigma_k\rangle \in FIPaths \wedge \sigma_k = \langle s_k, n_k, val_k\rangle$ }

- *FIT*: The set of transitions that are in a reachable configuration in which the invariant fails, which are the transitions that lead to a state in *FICS*.

  FIT={ $n_k \mid \exists\ p = \langle\sigma_0,\cdots,\sigma_k\rangle \in FIPaths \wedge \sigma_k = \langle s_k, n_k, val_k\rangle$ }

- *fst_cs*(p): The control state of the first configuration in path p.

  $fst\_cs(p)$=s_0 if $p = \langle\sigma_0,\cdots,\sigma_k\rangle \in FIPaths \wedge \sigma_0 = \langle s_0, n_0, val_0\rangle$

- *lst_cs*(p): The control state of the last configuration in path p.

  $lst\_cs(p)$=s_k if $p = \langle\sigma_0,\cdots,\sigma_k\rangle \in FIPaths \wedge \sigma_k = \langle s_k, n_k, val_k\rangle$

  By definition, $lst\_cs(p) \in FICS$.

- *lst_trans*(p): The last transition taken in path p.

  $lst\_trans(p)$=n_k if $p = \langle\sigma_0,\cdots,\sigma_k\rangle \in FIPaths \wedge \sigma_k$=$\langle s_k, n_k, val_k\rangle$

  By definition, $lst\_trans(p) \in FIT$.

- *trans_seq*(p): The finite sequence of transitions taken in path p.

  $trans\_seq(p)$=n_0,..,n_k if $p = \langle\sigma_0,\cdots,\sigma_k\rangle \in FIPaths \wedge \sigma_i$=$\langle s_i, n_i, val_i\rangle|_{0\leq i\leq k}$

- *all_but_last*(p): The finite sequence of configurations in path p except for the last one.

  $all\_but\_last(p)$=σ_0,$\cdots$,σ_{k-1} if $p = \langle\sigma_0,\cdots,\sigma_k\rangle \in FIPaths$

- *reduceEFSM*(p): Removes EFSM loops from path p. An **EFSM loop** is one that reaches the same control state in p more than once.

  $$reduceEFSM(p)=\langle\sigma_0,\cdots,\sigma_i,\cdots,\sigma_k\rangle \text{ if } p = \langle\sigma_0,\cdots,\sigma_i,\cdots,\sigma_j,\cdots,\sigma_k\rangle \in FIPaths \mid \\ \forall i,j \bullet \sigma_i=\langle s_i, n_i, val_i\rangle \wedge \sigma_j=\langle s_j, n_j, val_j\rangle \wedge s_i = s_j \wedge i \neq j$$

The notation $[x]$ is used for the equivalence class of x, which consists of the set of equivalent elements of *FIPaths* in the class x. x may be a control state, a path, or a transition, *etc.*

I present my levels of equivalence classes defined over the set *FIPaths* in order from the most detailed to the least detailed. I use the model AC in Figure 5.2 to present the counterexample equivalence classes created by each of the levels, which are shown in Table 5.2. The number beside an equivalence class in Table 5.2 indicates the elements of *FIPaths* in the class, illustrating the reduction achieved by my method since only one representative counterexample is presented to the user for each equivalence class. The total number of elements of *FIPaths* for the model is 45, with some of these elements shown in Table 5.3. The calculation of the number of elements of *FIPaths* is described in Section 5.6.5.

Level 1	Level 2	Level 3	Level 4
$[\langle t_1, t_4\rangle]$ – 18	$[t_4]$ – 30	[OFF,ON] – 30	[ON] – 30
$[\langle t_3, t_5, t_4\rangle]$ – 12			
$[\langle t_3, t_5\rangle]$ – 9	$[t_5]$ – 15	[OFF,IDLE] – 15	[IDLE] – 15
$[\langle t_1, t_4, t_5\rangle]$ – 6			

Table 5.2: Counterexample equivalence classes for Figure 5.2. The total number of elements of *FIPaths* is 45.

5.5.1 Level 1: Distinct Paths

I expect Level 1 to be the most commonly chosen level for analysis as it captures one representative of each *distinct* path through the EFSM that contains an error. My definition groups paths with multiple iterations of an EFSM loop together because they may seem the same from the user's perspective, not adding information as to the cause of the error. For example, the path of the model AC in Figure 5.2, with ON in *FICS*

$$\langle \mathsf{OFF}\text{-}t_1\text{-}\mathsf{IDLE}\text{-}t_2\text{-}\mathsf{OFF}\text{-}t_1\text{-}\mathsf{IDLE}\text{-}t_4\text{-}\mathsf{ON}\rangle$$

shows the same error information, from the user's perspective, as the EFSM path

$$\langle \mathsf{OFF}\text{-}t_1\text{-}\mathsf{IDLE}\text{-}t_2\text{-}\mathsf{OFF}\text{-}t_1\text{-}\mathsf{IDLE}\text{-}t_2\text{-}\mathsf{OFF}\text{-}t_1\text{-}\mathsf{IDLE}\text{-}t_4\text{-}\mathsf{ON}\rangle$$

and also the same information regarding the error as the path without EFSM loops

$$\langle \mathsf{OFF}\text{-}t_1\text{-}\mathsf{IDLE}\text{-}t_4\text{-}\mathsf{ON}\rangle.$$

In this case, the user would rather see another path that shows a different kind of error! Thus, I consider all paths with iterations of an EFSM loop to be equivalent to one without such loops, using *reduceEFSM* in the definition of Level 1.

However, EFSM loops that end in the configuration that fails the invariant cannot be eliminated without losing too much information because the actions on the transition leading to the failed configuration might be needed to help identify the cause of the error. Therefore, I make Level 1 differentiate paths by their last transition because these paths share the same potential error cause regardless of any EFSM loops in the rest of the path. The immediate cause of the failure (though not necessarily the bug in the model) can then be found by analyzing the guard or the actions of this transition. In Figure 5.7, bugs are differentiated by their last transition, otherwise, path $\langle t_1, t_2, t_2\rangle$ and path $\langle t_1, t_3, t_3\rangle$ would both be reduced to $\langle t_1\rangle$ by *reduceEFSM* with respect to C. Instead, Level 1 generates the equivalence classes $[\langle t_1, t_2\rangle]$ and $[\langle t_1, t_3\rangle]$.

⟨(OFF, e=enter, t=2, pt=0), (ON, t_3, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩	⟨(OFF, e=enter, t=1, pt=0), (IDLE, t_1, e=enter, t=1, pt=1), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=0, pt=0), (IDLE, t_1, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩
⟨(OFF, e=enter, t=2, pt=0), (ON, t_3, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=1, pt=0), (IDLE, t_1, e=enter, t=2, pt=1), (ON, t_4, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩	⟨(OFF, e=enter, t=0, pt=0), (IDLE, t_1, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=0, pt=2), (IDLE, t_5, pt=2)⟩
⟨(OFF, e=enter, t=2, pt=0), (ON, t_3, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩	⟨(OFF, e=enter, t=1, pt=0), (IDLE, t_1, e=enter, t=2, pt=1), (ON, t_4, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=0, pt=0), (IDLE, t_1, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩
⟨(OFF, e=enter, t=2, pt=0), (IDLE, t_1, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=1, pt=0), (ON, t_3, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩	⟨(OFF, e=enter, t=0, pt=0), (ON, t_3, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩
⟨(OFF, e=enter, t=2, pt=0), (IDLE, t_1, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩	⟨(OFF, e=enter, t=1, pt=0), (ON, t_3, e=enter, t=1, pt=2), (IDLE, t_5, e=enter, t=1, pt=1), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=0, pt=0), (IDLE, t_1, e=enter, t=2, pt=1), (ON, t_4, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩
⟨(OFF, e=enter, t=2, pt=0), (ON, t_3, e=enter, t=1, pt=2), (IDLE, t_5, e=enter, t=2, pt=1), (ON, t_4, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩	⟨(OFF, e=enter, t=1, pt=0), (ON, t_3, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩	⟨(OFF, e=enter, t=0, pt=0), (ON, t_3, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=2, pt=2), (IDLE, t_5, pt=2)⟩
⟨(OFF, e=enter, t=2, pt=0), (ON, t_3, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=1, pt=2), (IDLE, t_5, e=enter, t=1, pt=1), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=1, pt=0), (ON, t_3, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=1, pt=2), (IDLE, t_5, e=enter, t=1, pt=1), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=0, pt=0), (ON, t_3, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=1, pt=2), (IDLE, t_5, e=enter, t=1, pt=1), (ON, t_4, pt=1)⟩
⟨(OFF, e=enter, t=2, pt=0), (IDLE, t_1, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=1, pt=0), (IDLE, t_1, e=enter, t=2, pt=0), (ON, t_4, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=0, pt=0), (IDLE, t_1, e=enter, t=2, pt=1), (ON, t_4, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩
⟨(OFF, e=enter, t=2, pt=0), (ON, t_3, e=enter, t=1, pt=2), (IDLE, t_5, e=enter, t=2, pt=1), (ON, t_4, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=1, pt=0), (ON, t_3, e=enter, t=1, pt=2), (IDLE, t_5, e=enter, t=2, pt=1), (ON, t_4, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩	⟨(OFF, e=enter, t=0, pt=0), (ON, t_3, e=enter, t=1, pt=2), (IDLE, t_5, e=enter, t=2, pt=1), (ON, t_4, e=enter, t=0, pt=2), (IDLE, t_5, e=enter, t=1, pt=0), (ON, t_4, pt=1)⟩

Table 5.3: Example of elements in *FIPaths* for model AC shown in Figure 5.2. The total number of elements of *FIPaths* is 45.

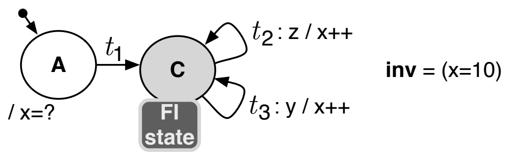


Figure 5.7: Counterexample differentiated by last transition

Note that a representative counterexample is presented to the user for each equivalence class. Thus, even though EFSM loops are removed to define the equivalence class, if an EFSM loop is needed to fail the invariant (because of data changes, such as an EFSM loop for a counter), this loop will be present in what the user receives. For instance, in the model of Figure 5.8 **(a)**, transition t_4 needs to be taken at least once for the invariant to fail, therefore, the EFSM loop is necessary. Some paths containing necessary EFSM loops with t_2 and t_4 are illustrated in Figure 5.8 **(b)**. The equivalence class generated for these paths is $[\langle t_1, t_3 \rangle]$ because it summarizes the information to recognize the error, differentiated by its last transition, without the need to record and look at all paths with various combinations of the EFSM loops.

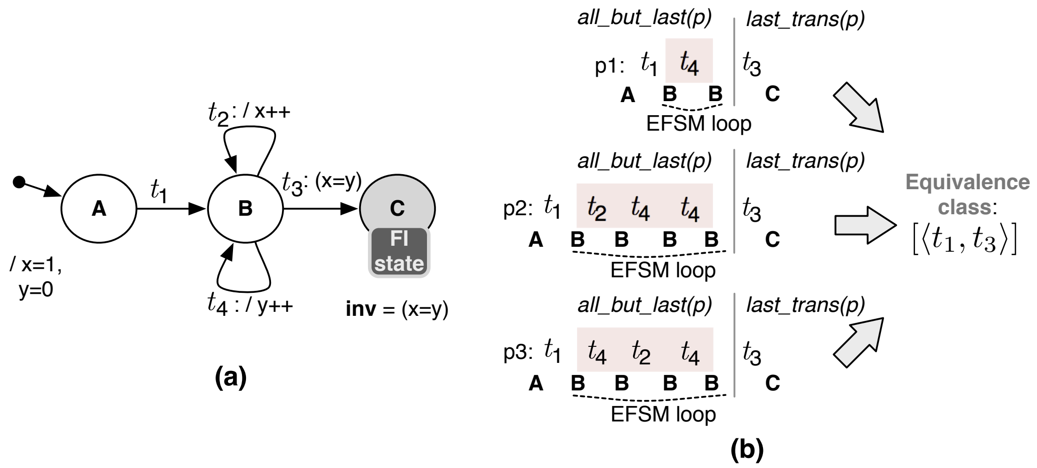


Figure 5.8: EFSM loops due to counters are eliminated by *reduceEFSM*

A limitation of using the reduction of EFSM loops in Level 1 is that *Alfie* could miss some distinct bugs, for instance, in the case of two self-looping transitions that both have errors, such as self-looping transitions t_2 and t_4 in Figure 5.8. However, I made the design decision of reducing EFSM loops to make the verification more efficient because, even with the reduction of EFSM loops, some models can generate a large number of equivalence classes for Level 1. Including even one instance of each different EFSM loop would make the problem of generating all equivalence classes of counterexamples for Level 1 even harder, based on my observation while experimenting with this idea.

Level 1 considers as equivalent all the paths that end in the same transition and that have the same sequence of transitions after removing EFSM loops in the rest of the path[5].

Definition 1: $\forall p \in \mathit{FIPaths} \bullet$

$$[p] = \{q \in \mathit{FIPaths} \mid \begin{array}{l} \mathit{lst\_trans}(q) = \mathit{lst\_trans}(p) \\ \wedge\ \mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{all\_but\_last}(p))) = \mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{all\_but\_last}(q))) \end{array}\}$$

There are four equivalence classes at Level 1 for the EFSM AC in Figure 5.2. For example, the data variations of path ⟨OFF-t_1-IDLE-t_4-ON-t_5-IDLE-t_4-ON⟩ are all part of the equivalence class $[\langle t_1, t_4\rangle]$ after removing the EFSM loop with respect to control state IDLE. Four distinct counterexamples represent these four equivalence classes. Two of them were described in Section 5.3 ($\langle t_1, t_4\rangle$ and $\langle t_3, t_5\rangle$) and the other two counterexamples are the following:

(3) The counterexample path **p_7**

```
⟨(OFF,  tn, e=enter, t=0, pt=0),
 (IDLE, t1, e=enter, t=2, pt=0),
 (ON,   t4, e=enter, t=2, pt=2),
 (IDLE, t5, e=exit,  t=1, pt=2),
 ···⟩,
```

is an instance of the EFSM path ⟨OFF-t_1-IDLE-t_4-ON-t_5-IDLE⟩;

(4) The counterexample path **p_8**

```
⟨(OFF,  tn, e=enter, t=2, pt=0),
 (ON,   t3, e=enter, t=0, pt=2),
 (IDLE, t5, e=enter, t=1, pt=0),
 (ON,   t4, e=exit,  t=0, pt=1),
 ···⟩,
```

is an instance of the EFSM path ⟨OFF-t_3-ON-t_5-IDLE-t_4-ON⟩.

These last two are distinct EFMS paths, but the bugs in the model illustrated by these counterexamples are the same as the ones found by the first two distinct counterexamples. In another model, it might not be the case that two distinct EFSM paths have the same error cause.

5.5.2 Level 2: Distinct Last Transitions

All the paths that have the same last transition are considered equivalent.

Definition 2: $\forall t \in \mathit{FIT} \bullet [t] = \{p \in \mathit{FIPaths} \mid \mathit{lst\_trans}(p) = t\}$

There are two equivalence classes at Level 2 for the EFSM AC in Figure 5.2. For example, the path ⟨OFF-t_1-IDLE-t_4-ON-t_5-IDLE-t_4-ON⟩ is part of the equivalence class $[t_4]$.

[5]For Levels 1-2, the counterexample path must be of length at least one.

5.5.3 Level 3: Distinct Initial and Final States

All the paths that have the same initial control state and final control state are considered equivalent.

Definition 3: $\forall i \in \mathit{InitCS}, \forall s \in \mathit{FICS} \bullet$
$[i, s] = \{p \in \mathit{FIPaths} \mid \mathit{fst\_cs}(p) = i \wedge \mathit{lst\_cs}(p) = s\}$

An equivalence class is empty if an initial control state is not the first state on a path in *FIPaths*. There are two non-empty equivalence classes at Level 3 for the EFSM AC in Figure 5.2. For example, the path $\langle$OFF-t_1-IDLE-t_4-ON-t_5-IDLE-t_4-ON$\rangle$ is part of the equivalence class [OFF,ON].

Level 3 can be used as a preliminary check to examine conditions on the initial control states and variable values that lead to an error in order to find bugs in the specification of possible initial values.

5.5.4 Level 4: Distinct Final States

All the paths that lead to the same final control state are considered equivalent.

Definition 4: $\forall s \in \mathit{FICS} \bullet$
$[s] = \{p \in \mathit{FIPaths} \mid \mathit{lst\_cs}(p) = s\}$

There are two equivalence classes at Level 4 for the EFSM AC in Figure 5.2. For example, the path $\langle$OFF-t_1-IDLE-t_4-ON-t_5-IDLE-t_4-ON$\rangle$ is part of the equivalence class [ON].

Chechik and Gurfinkel say that a level like this one is important to quickly find errors in initial states when the very first state fails the invariant [51]. This level might also be useful in choosing a resolution: *e.g.,* a sink/error state could be added to the model from these states upon discovery by a counterexample.

5.5.5 Discussion

A counterexample in which a second error in the model can *only* be reached by going through an earlier configuration that failed the invariant falls into the equivalence class of the path to the first error, and therefore, it is not distinguished from it, as illustrated in Figure 5.9 where the equivalence class is chosen with respect to control state Y. My method distinguishes counterexamples with respect to the first error on the path only. However, if there is a path that reaches the second error passing through the control state where a previous bug was recognized but without failing the invariant there this time (*e.g.,* control state Y in Figure 5.9), this path would be in a different equivalence class (*e.g.,* a bug with respect to control state Z).

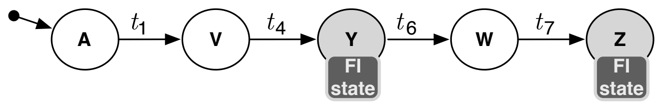


Figure 5.9: Counterexample path with multiple failed invariants

5.6 On-the-fly LTL Counterexample Grouping

This section describes how my method uses a model checker to implement the definitions introduced in Section 5.5. My method and tool, both called *Alfie*, represent the levels of equivalence classes on-the-fly using LTL properties. In related work [60], the user first generates all counterexamples by iterating the model checker and then groups them into equivalence classes. In my method, by ruling out all equivalent counterexamples on-the-fly, there is no need to generate all counterexamples, which substantially reduces the number of iterations of the model checker. An LTL property describes all paths in an equivalence class of a counterexample rather than the complete list of counterexamples.

My on-the-fly grouping method and tool *Alfie* is illustrated in Figure 5.10. My tool *Alfie* iteratively (1) asks SMV to generate a counterexample, (2) creates the equivalence class from the the counterexample for the desired level, (3) represents this equivalence class as an LTL expression, (4) creates a new property that is the disjunction of this LTL expression with the invariant and the LTL expressions representing previously generated counterexamples, and (5) repeats the process by re-running the model checker on the same model with the new property. By disjuncting an LTL expression of the equivalence class with the property, *Alfie* disallows the generation of any more counterexamples in that equivalence class. The output of my method is one representative counterexample per equivalence class. I call this set of representative counterexamples ***CErep***. This iterative process runs automatically via scripts, and it is repeated until no more counterexamples are found. *Alfie* does not rely on the order in which the counterexamples are generated (such as generation of the shortest one first). This process terminates because the set of *FIPaths* is finite and a different element is generated at each iteration of my method.

The model input to *Alfie* must contain the following:

inv: Macro specifying the criteria to identify an error in the model, *i.e.,* the invariant predicate **inv**, defined over the KS variables.

progress: Macro specifying $(\mathrm{X}(\neg(tn)) \vee \mathsf{final\_states})$. This condition forces the model checker to only produce counterexamples with progressing transitions unless the model reaches a final control state (*i.e.,* a state that is not a source of any progressing transition). If not explicitly defined, *Alfie* can identify the final states in the model and include this information in the macro progress.

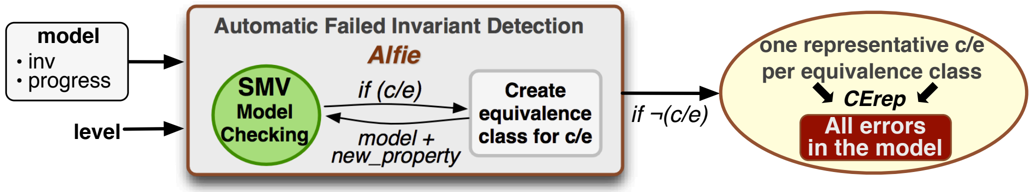


Figure 5.10: On-the-fly grouping level process

To generate the equivalence class of a counterexample, c, my method first calculates the element of *FIPaths*, q, associated with c. But by incorporating parts of the definition of *FIPaths* into the LTL property to be checked, I can limit the model checking exploration. Therefore, for every level of equivalence classes, my method begins by checking the property

$$\textbf{prop}\text{: } \mathrm{G}(\mathsf{progress}) \rightarrow \mathrm{G}(\mathsf{inv})$$

to generate the first counterexample. The macro progress included in the property implements the definition of the function *progress* in Section 5.4, ensuring that the counterexample c returned by the model checker only contains progressing transitions. Then, my method applies to c the rest of the definition of *FIPaths*, described in Section 5.4, thus generating the element q. First, the function *trunc* creates a subpath of c ending in the first configuration that fails the invariant, followed by the application of the functions *reduce_config_loops*, *reduce_init_config* and *reduce_vals* to the subpath returned by *trunc*. For the element q of *FIPaths*, *Alfie* creates an LTL expression L according to the desired level that is added to the invariant as a disjunction, creating the property to check next

$$\textbf{prop\_L}\text{: } \mathrm{G}(\mathsf{progress}) \rightarrow ((\mathrm{G}(\mathsf{inv})) \vee L).$$

The iterative process that *Alfie* follows is summarized by **Algorithm 2**, where the function *mk_ltl_expr* creates the LTL property, and comments are preceded by the symbol ▷. Figure 5.11 illustrates the relationship between counterexamples, elements of *FIPaths*, and LTL properties representing equivalence classes.

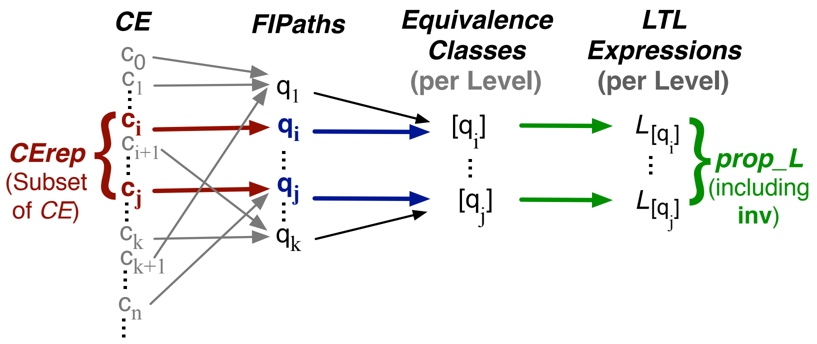


Figure 5.11: Relationship between *CE*, *CErep*, *FIPaths* and LTL properties representing equivalence classes

Algorithm 2 – *alfie*(model, level)

Input: model (containing macros inv, and progress), level
Output: *CErep* (set of representative counterexamples, one per equivalence class)

```
1: $CErep \leftarrow \emptyset$
2: prop_L ← prop
3: RUN_SMV(model, prop_L)
4: while (counterexample $c$ generated) do
5:     ▷ Create LTL expression for $c$ according to desired level
6:     $L \leftarrow mk\_ltl\_expr(c,$ level$)$
7:     prop_L ← prop_L + $L$  ▷ Add $L$ as disjunction with the invariant
8:     $CErep \leftarrow CErep \cup c$  ▷ Add counterexample $c$ to set $CErep$
9:     RUN_SMV(model, prop_L)
10: end while
11: return $CErep$
```

The loop invariant that holds after each iteration of *Alfie* is:

Alfie loop-invariant: $\forall p \in CE \bullet (p \models \textbf{prop\_L} \Leftrightarrow \exists c \in CErep \bullet p \in [c])$

meaning that **prop_L** exactly matches the equivalence classes for the counterexamples seen so far in the process. Note that $p \in (AllPaths - CE)$ also satisfies **prop_L** because these paths are not counterexamples. From this loop-invariant, it can be concluded that the number of iterations of *Alfie* is always equal to the number of equivalence classes. Because the LTL expression representing an equivalence class is always added as a disjunction, the set of possible counterexamples in the next iteration is strictly reduced, *i.e.,* the change in the property cannot result in any additions to the set *CE*. In the following sections, I justify per level that the property exactly represents the counterexamples equivalent to those in *CErep*.

A corollary of the loop-invariant is:

CErep_uniqueness: $\forall c_1, c_2 \in CErep \bullet c_1 \neq c_2 \Rightarrow [c_1] \neq [c_2]$

***Justification of corollary of Alfie loop-invariant*:**

The loop-invariant says that the LTL property exactly represents the set of equivalence classes seen so far in the process. If $c_1 \neq c_2$, with $c_1, c_2 \in CErep$, then there would have been an iteration of *Alfie* in which c_1 was already part of *CErep* and c_2 was generated by *Alfie*. So, for c_2 to be generated at that iteration, c_2 could not have been part of an equivalence class already represented in *CErep*, and therefore, $[c_1] \neq [c_2]$. □

The incorporation of *progress*, which is part of the definition of *FIPaths*, to the property checked by my method is beneficial to reduce the model checking effort. However, the property to check, *e.g.,* either **prop** or **prop_L**, then becomes an implication, and care must be taken that this property is not vacuously satisfied when the antecedent of the implication is false. Therefore, before attempting to generate results for any level of counterexample equivalence classes, *Alfie* verifies the property EG progress to ensure that it is possible for the model to take progressing transitions, and thus, that the antecedent of the property is not trivially satisfied.

Next, I explain in order from least complex to most complex (reverse order of Section 5.5) the LTL expression that represents an equivalence class of elements of *FIPaths* according to the desired level. The LTL expressions for a counterexample from model AC of Figure 5.2, are summarized in Table 5.4, where the LTL expression L added per level is highlighted. For each level of equivalence classes, the loop invariant that must hold for each iteration of *Alfie* will be briefly justified. In the rest of this section, I will use only control states and transitions to describe a path instead of listing all the configurations forming the path, thus simplifying the explanations.

Level	**LTL property**
4	$(\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \ (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \mathsf{ON}))\ )$
3	$(\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \ (\mathsf{OFF} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \mathsf{ON})))\ )$
2	$(\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \ (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge t_4))\ )$
1	$(\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \ (\mathsf{OFF} \wedge (\mathsf{inv}\ \mathrm{U}\ (t_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge t_4)))))\ )$

Table 5.4: LTL properties per level of equivalence classes for counterexample path **p_3**, $\langle$OFF-t_1-IDLE-t_4-ON$\rangle$, from model AC of Figure 5.2

5.6.1 Level 4: Distinct Final States

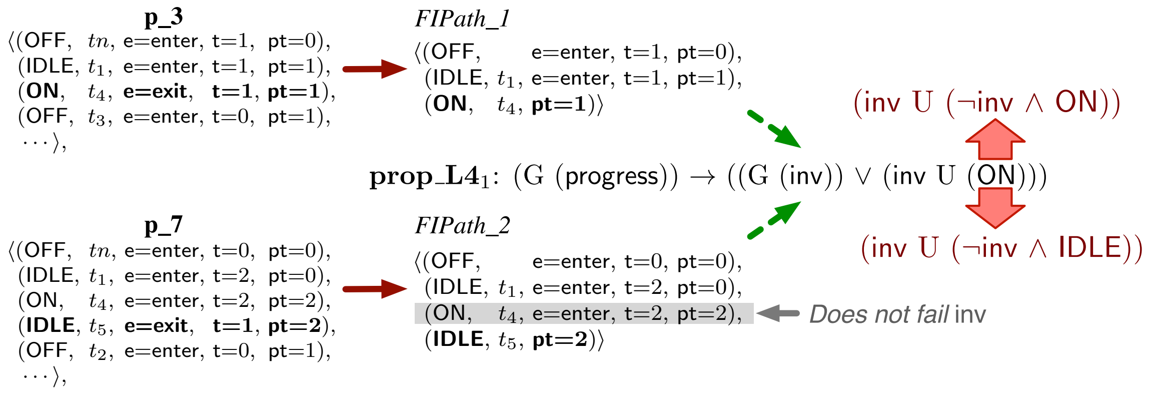
For a path $q \in$ *FIPaths*, in Level 4 *Alfie* adds to the invariant a disjunction with an LTL expression L that has the value of the control state in the last configuration of q (which is an element of *FICS*), generating the following property with L highlighted:

$$\textbf{prop\_L4}\text{: } (\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \ (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \mathit{lst\_cs}(q)))\ )^{6}.$$

prop_L4 forces the model checker to find a counterexample in which the first control state to fail inv is different from $\mathit{lst\_cs}(q)$.

[6]Recall that $\mathit{lst\_cs}(q)$ is shorthand for $\mathit{cs}{=}\mathit{lst\_cs}(q)$

Expression L in **prop_L4** includes $\neg$inv as a conjunction with *lst_cs*(q) because another counterexample may have a prefix with *lst_cs*(q) in it, but the invariant does not fail in that instance of *lst_cs*(q), and this counterexample is in a distinct equivalence class, as illustrated by Figure 5.12. In Figure 5.12, path **p_3** maps to *FIPath_1*, so property **prop_L4**$_1$ would be generated by *Alfie* for Level 4 if $\neg$inv was excluded from the LTL expression L. However, the element of *FIPaths* generated from **p_7** would also satisfy property **prop_L4**$_1$, and **p_7** would not be produced by SMV in another iteration. Therefore, the inclusion of $\neg$inv to expression L in property **prop_L4**$_1$ is necessary, as indicated in Figure 5.12. A similar justification of the need of $\neg$inv in expression L follows for Level 3 and for Level 2.



p_3
⟨(OFF, tn, e=enter, t=1, pt=0),
(IDLE, t_1, e=enter, t=1, pt=1),
(ON, t_4, e=exit, t=1, pt=1),
(OFF, t_3, e=enter, t=0, pt=1),
···⟩,
FIPath_1
⟨(OFF, e=enter, t=1, pt=0),
(IDLE, t_1, e=enter, t=1, pt=1),
(ON, t_4, pt=1)⟩
(inv U (¬inv ∧ ON))
prop_L4$_1$: (G (progress)) → ((G (inv)) ∨ (inv U (ON)))
(inv U (¬inv ∧ IDLE))
p_7
⟨(OFF, tn, e=enter, t=0, pt=0),
(IDLE, t_1, e=enter, t=2, pt=0),
(ON, t_4, e=enter, t=2, pt=2),
(IDLE, t_5, e=exit, t=1, pt=2),
(OFF, t_2, e=enter, t=0, pt=1),
···⟩,
FIPath_2
⟨(OFF, e=enter, t=0, pt=0),
(IDLE, t_1, e=enter, t=2, pt=0),
(ON, t_4, e=enter, t=2, pt=2),
(IDLE, t_5, pt=2)⟩
Does not fail inv

Figure 5.12: Counterexample path missed by property **prop_L4** if $\neg$inv is excluded

In the rest of the section, I will justify that the loop invariant for *Alfie* holds with respect to property **prop_L4** for Level 4.

***Justification of Alfie loop-invariant*:**

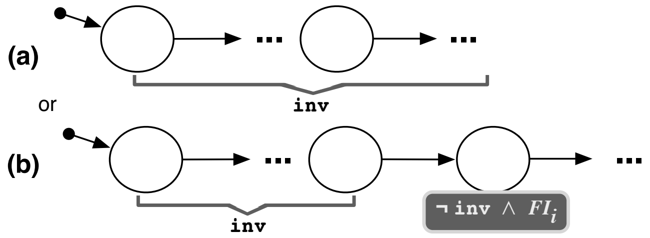
(⇒) $\forall p \in CE \bullet (p \models \textbf{prop\_L4} \Rightarrow \exists c \in CErep \bullet p \in [c])$

For Level 4, property **prop_L4** has the form

$$(\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge FI_1))$$
$$\cdots$$
$$\vee\ (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge FI_k)))$$

where $\{FI_1,\cdots,FI_k\} = \{lst\_cs(FIPaths(c_1)),\cdots,lst\_cs(FIPaths(c_k)) \mid c_1,..,c_k \in CErep\}$.
In the justification, because the antecedent G(progress) does not change, I concentrate only on the consequent.

If $p \in \mathit{AllPaths}$ satisfies **prop_L4**, then p must be of one of two forms:

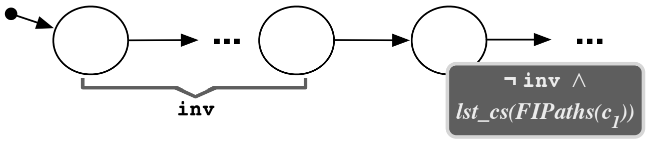


For case **(a)**, p is not in CE. For case **(b)**, a path p has as the first control state that fails the invariant an element FI_i, returned by $\mathit{lst\_cs}(\mathit{FIPaths}(c_i))$. Therefore, $p \in [c_i]$ for a $c_i \in \mathit{CErep}$. □

(⇐) $\forall p \in \mathit{CE} \bullet ((\exists c \in \mathit{CErep} \bullet p \in [c]) \Rightarrow p \models$ **prop_L4**$)$

Let path p be a member of an equivalence class of c_1 in CErep. By **Algorithm 2** for Level 4, when c_1 was generated by the model checker, property **prop_L4** included the LTL expression $(\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \mathit{lst\_cs}(\mathit{FIPaths}(c_1))))$, disjuncted with the invariant.

If $p \in [c_1]$, p must have the form:



Therefore, path p satisfies property **prop_L4**.

□

5.6.2 Level 3: Distinct Initial and Final States

For a path $q \in \mathit{FIPaths}$, in Level 3 *Alfie* adds to the invariant a disjunction with an LTL expression L describing the initial control state of q (which must be an element of InitCS), and the last control state in q (which must be an element of FICS), generating the following property with L highlighted:

$$\textbf{prop\_L3}\text{: } (\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee (\mathit{fst\_cs}(q) \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \mathit{lst\_cs}(q))))\ ).$$

prop_L3 forces the model checker to search for a counterexample that either starts with the same initial control state, but ends at a different control state that fails the invariant, or starts with a different initial control state and ends at a control state where the invariant fails. Over multiple paths, all final control states are grouped with the same initial control state together in a disjunction with the invariant for brevity of the LTL expression. In the rest of the section, I will justify the loop invariant for *Alfie* with respect to Level 3.

***Justification of Alfie loop-invariant*:**

(⇒) $\forall p \in \mathit{CE} \bullet (p \models \textbf{prop\_L3} \Rightarrow \exists c \in \mathit{CErep} \bullet p \in [c])$

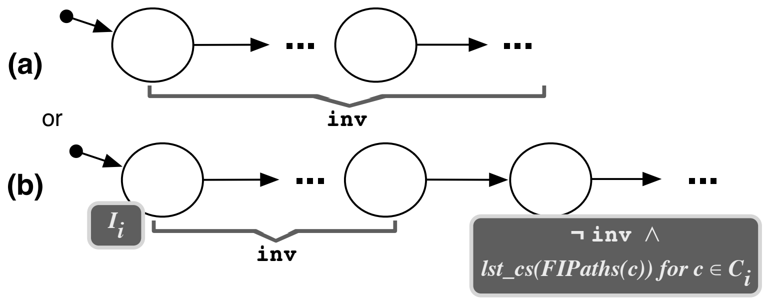
Let *CErep* be partitioned by the set $I = \{I_1,...,I_k\}$ of initial control states into subsets such that $\mathit{CErep} = \mathrm{C}_1 \cup \mathrm{C}_2 \cup \cdots \cup \mathrm{C}_k$, where $\forall i : 0..k,\ \forall c \in \mathrm{C}_i \bullet \mathit{fst\_cs}(c){=}\mathrm{I}_i$.

For Level 3, property **prop_L3** has the form

$$(\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \\ \vee\ (I_1 \wedge (\mathsf{inv}\ U\ (\neg\mathsf{inv} \wedge (\bigvee_{c\in C_1} \mathit{lst\_cs}(c))))) \\ \ldots \\ \vee\ (I_k \wedge (\mathsf{inv}\ U\ (\neg\mathsf{inv} \wedge (\bigvee_{c\in C_k} \mathit{lst\_cs}(c)))))).$$

In the justification, because the antecedent G(progress) does not change, I concentrate only on the consequent.

If $p \in \mathit{AllPaths}$ satisfies **prop_L3**, then p must be of one of two forms:



For case **(a)**, p is not in *CE*. For case **(b)**, a path p has as its initial control state I_i, returned by $\mathit{fst\_cs}(\mathit{FIPaths}(c))$, $\forall c \in \mathrm{C}_i$ (because all counterexamples in partition C_i start with the same initial state), and path p has as the first control state that fails the invariant $\mathit{lst\_cs}(\mathit{FIPaths}(c))$ for some $c \in \mathrm{C}_i$ (because each counterexample in partition C_i ends with a different final state). Therefore, $p \in [c]$ for $c \in \mathrm{C}_i$ and $\mathrm{C}_i \subseteq \mathit{CErep}$. □

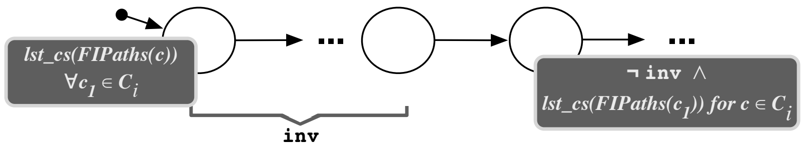
(⇐) $\forall p \in \mathit{CE} \bullet (\exists c \in \mathit{CErep} \bullet p \in [c] \Rightarrow p \models \textbf{prop\_L3})$

Let path p be a member of the equivalence class of $c_1 \in \mathrm{C}_i$ and $\mathrm{C}_i \subseteq \mathit{CErep}$. By **Algorithm 2** for Level 3, when c_1 was generated by the model checker, property **prop_L3** included the LTL expression

$$(\mathit{fst\_cs}(\mathit{FIPaths}(c_1)) \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge (\ \cdots \vee \mathit{lst\_cs}(\mathit{FIPaths}(c_1)) \vee \cdots\ ))),$$

disjuncted with the invariant.

If $p \in [c_1]$, p must have the form:



Therefore, path p satisfies property **prop_L3**. □

5.6.3 Level 2: Distinct Last Transitions

For a path $q \in \mathit{FIPaths}$, in Level 2 *Alfie* adds to the invariant a disjunction with an LTL expression L that has the value of the last transition taken in q (which must lead to a control state in *FICS* and therefore be part of *FIT*), generating the following property with L highlighted:

$$\textbf{prop\_L2}\text{: (G (progress))} \rightarrow \text{((G (inv))} \vee \text{ (inv U } (\neg\text{inv} \wedge \mathit{lst\_trans}(q)))\text{),}$$

prop_L2 forces the model checker to find a counterexample in which the transition that leads to the first control state that fails the invariant is different from the one described by $\mathit{lst\_trans}(q)$. In the rest of the section, I will justify the loop invariant for *Alfie* regarding Level 2.

***Justification of Alfie loop-invariant*:**

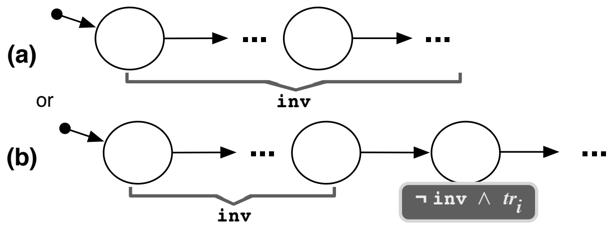
$(\Rightarrow)$ $\forall p \in \mathit{CE} \bullet (p \models \textbf{prop\_L2} \Rightarrow \exists c \in \mathit{CErep} \bullet p \in [c])$

For Level 2, property **prop_L2** has the form

$$\begin{aligned}&\text{(G (progress))} \rightarrow \text{((G (inv))} \vee \text{(inv U } (\neg\text{inv} \wedge tr_1))\\ &\qquad\cdots\\ &\qquad\vee \text{(inv U } (\neg\text{inv} \wedge tr_k)))\end{aligned}$$

where $\{tr_1,\cdots,tr_k\} = \{\mathit{lst\_trans}(\mathit{FIPaths}(c_1)),\cdots,\mathit{lst\_cs}(\mathit{FIPaths}(c_k)) \mid c_1,..,c_k \in \mathit{CErep}\}$.

If $p \in \mathit{AllPaths}$ satisfies **prop_L2**, then p must be of one of two forms:

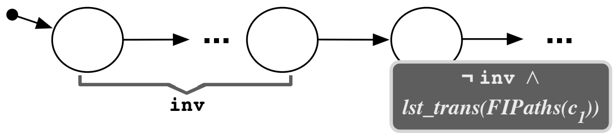


For case **(a)**, p is not in *CE*. For case **(b)**, a path p has tr_i as the transition that leads to the first control state that fails the invariant, which is equal to $\mathit{lst\_trans}(\mathit{FIPaths}(c_i))$. Therefore, $p \in [c_i]$ for $c_i \in \mathit{CErep}$. □

($\Leftarrow$) $\forall p \in \mathit{CE} \bullet (\exists c \in \mathit{CErep} \bullet p \in [c] \Rightarrow p \models$ **prop_L2**)

Let path p be a member of the equivalence class of c_1 in *CErep*. By **Algorithm 2** for Level 2, when c_1 was generated by the model checker, property **prop_L2** included the LTL expression (inv U ($\neg\mathsf{inv} \wedge \mathit{lst\_trans}(\mathit{FIPaths}(c_1))$)), disjuncted with the invariant.

If $p \in [c_1]$, p must have the form:



Therefore, path p satisfies property **prop_L2**. □

5.6.4 Level 1: Distinct Paths

For a path $q \in \mathit{FIPaths}$, in Level 1 *Alfie* adds to the property a disjunction with an LTL expression L that makes the model checker accept any path with the same sequence of transitions as q, and all EFSM looping variations of this path, except that the transition entering the last state where the invariant fails must be the same on all paths. A looping variant is any path that reaches that same control state two or more times in the path as illustrated in Figure 5.13. By including the looping variations, the model checker will not report them separately. The EFSM loops in the looping variations of a path must not contain states that fail the invariant. My method forces counterexamples with different final transitions to be in distinct equivalence classes because the actions of the last transition might have caused the invariant to fail.

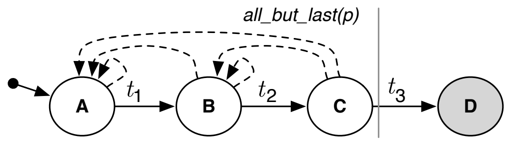


Figure 5.13: Looping variations of path $\langle t_1, t_2, t_3 \rangle$

Level 1 was the most difficult level to express correctly in LTL. For the benefit of the reader, next I will describe step by step the various options that lead me to the correct LTL expression L for Level 1, using the AC model in Figure 5.2 with the counterexample path **p_3**, $\langle \mathsf{OFF}\text{-}t_1\text{-}\mathsf{IDLE}\text{-}t_4\text{-}\mathsf{ON} \rangle$, to illustrate each option. Then, I will justify that the loop invariant for *Alfie* holds after each iteration for Level 1.

Option 1: If a model checker returns as a counterexample the path **p_3** for model AC, the element of *FIPaths* generated from it is q_1: $\langle t_1, t_4 \rangle$. The most natural LTL expression to describe a sequence including looping variants, and to be used as a disjunction with the invariant is (where A is the initial state in the path):

$$\mathsf{A} \wedge (\mathrm{F}\ t_1 \wedge (\mathrm{F}\ t_4)) \qquad (L\_a)$$

However, expression (L_a) allows looping variants of path q_1 that contain configurations that fail the invariant to be part of the same equivalence class, which is incorrect because the process must only group together paths that have the same sequence of transitions and whose last transition leads to the *first* configuration in the path that fails the invariant.

Option 2: The reasoning above lead me to consider an expression with the operator Until. Through the use of the Until operator, the LTL expression allows paths with EFSM loops whose states all satisfy the invariant to be included in the same equivalence class. Moreover, the expression with the Until operator fails when an EFSM loop contains states that fail the invariant. The new LTL expression for path q_1 in model AC is expressed as:

$$\mathsf{A} \wedge (\mathsf{inv}\ \mathrm{U}\ (t_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (t_4)))) \qquad (L\_b)$$

Expression (L_b) is not yet entirely correct, as it would allow paths that have the sequence described as a subpath of another path. For instance, given expression (L_b), the equivalence class $[\langle t_1, t_4, t_5\rangle]$ from counterexample path **p_7** would not be generated. In this case, expression (L_b) is describing the counterexample path **p_3**, with sequence of transitions t_1-t_4, which happens to be a subpath from the element of *FIPaths* q_2: $\langle t_1, t_4, t_5\rangle$, generated from counterexample path **p_7**. However, in path **p_7**, t_4 does not reach a configuration that fails the invariant as it did in path **p_3**, but instead, t_5 does.

Option 3: The LTL expression for q_1 must explicitly include the condition that indicates that the invariant does not hold when taking the transition to the failed invariant, which is the last transition of the path, such as t_4 in q_1 *i.e.,* $\underline{(\neg\mathsf{inv} \wedge t_4)}$ The new LTL expression for q_1 (generated from path **p_3**) is:

$$\mathsf{A} \wedge (\mathsf{inv}\ \mathrm{U}\ (t_1 \wedge (\mathsf{inv}\ \mathrm{U}\ \underline{(\neg\mathsf{inv} \wedge t_4)}))) \qquad (L\_c)$$

In contrast, the LTL expression for q_2 (generated from path **p_7**) is:

$$\mathsf{A} \wedge (\mathsf{inv}\ \mathrm{U}\ (t_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (t_4 \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge t_5)))))) \qquad (L\_d)$$

Therefore, the new LTL formalization shown by (L_c) and (L_d) allows my method to follow the definition of Level 1, thus generating the following property with L highlighted:

prop_L1: $(\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee$
$(\mathsf{I} \wedge (\mathsf{inv}\ \mathrm{U}\ (t_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (t_2 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (t_{k-1} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge t_k)))))))))\ )$,

where t_k is in *FIT*. In the rest of the section, I will justify that the loop invariant for *Alfie* holds with respect to Level 1.

***Justification of Alfie loop-invariant*:**

(⇒) $\forall p \in CE \bullet (p \models \textbf{prop\_L1} \Rightarrow \exists c \in CErep \bullet p \in [c])$

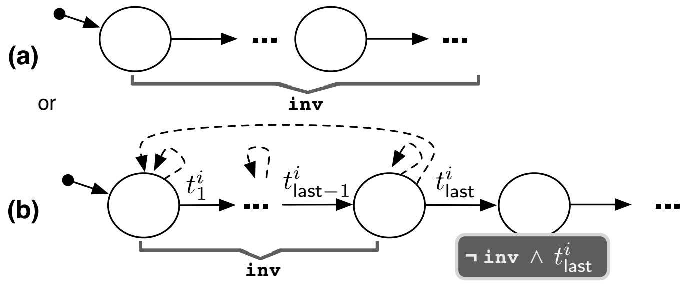
For Level 1, property **prop_L1** has the form:

$$
\begin{aligned}
&(\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv}))\\
&\qquad\qquad \vee (\mathsf{I}_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (t^1_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (t^1_{\mathsf{last}-1} \wedge\\
&\qquad\qquad\quad (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge t^1_{\mathsf{last}})))))))\\
&\qquad\qquad\qquad \cdots\\
&\qquad\qquad \vee (\mathsf{I}_k \wedge (\mathsf{inv}\ \mathrm{U}\ (t^k_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (t^k_{\mathsf{last}-1} \wedge\\
&\qquad\qquad\quad (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge t^k_{\mathsf{last}}))))))))
\end{aligned}
$$

where

$$
\begin{aligned}
&\{\ \langle t^i_1, \cdots, t^i_{\mathsf{last}}\rangle \mid \exists\ c_i \in CErep\ \bullet\\
&\qquad \langle t^i_1, \cdots, t^i_{\mathsf{last}-1}\rangle = \mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{all\_but\_last}(\mathit{FIPaths}(c_i)))) \text{ and}\\
&\qquad\qquad t^i_{\mathsf{last}} = \mathit{lst\_trans}(\mathit{FIPaths}(c_i))\ \}.
\end{aligned}
$$

If $p \in \mathit{AllPaths}$ satisfies **prop_L1**, then p must be of one of two forms:



For case **(a)**, p is not in CE. For case **(b)**, path p has $\langle t^i_1, ..., t^i_{\mathsf{last}-1}\rangle$ as the sequence of transitions resulting after removing any EFSM loops from the element of *FIPaths* of p, returned by $\mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{all\_but\_last}(\mathit{FIPaths}(c_i))))$, and t^i_{last} as the transition that leads to the first control state that fails the invariant, returned by $\mathit{lst\_trans}(\mathit{FIPaths}(c_i))$. Therefore, $p \in [c_i]$ for $c_i \in CErep$. □

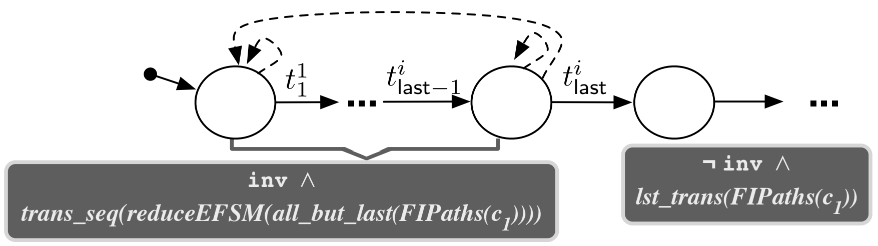
(⇐) $\forall p \in CE \bullet (\exists c \in CErep \bullet p \in [c] \Rightarrow p \models \textbf{prop\_L1})$

Let path p be a member of the equivalence class of c_1 in *CErep*. By **Algorithm 2** for Level 1, when c_1 was generated by the model checker, property **prop_L1** included the LTL expression

$$(\mathsf{I}_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (t^1_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (t^1_2 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \mathit{lst\_trans}(\mathit{FIPaths}(c_1)))))))))$$

disjuncted with the invariant.

If $p \in [c_1]$, p must have the form:



Therefore, path p satisfies property **prop_L1**. $\square$

5.6.5 FIPaths

There are a variety of methods for generating the set of all counterexamples (*e.g.,* use of SPIN [93], proof strategies to guide the search [51]). However, the set *FIPaths* is often too large to generate and comprehend given the data variations. To give a measure of the reduction achieved by my method, I used *Alfie* to generate the elements of *FIPaths* for the model AC in Figure 5.2 via an approach similar to that of Level 1, but with configurations rather than just control states and transitions. *Alfie* iteratively disjuncts the invariant with an LTL expression representing the element of *FIPaths*. My method, however, never expects the user to ask for the elements of *FIPaths*. The generation of all elements of *FIPaths* is practical only for tiny examples because the complexity of LTL model checking depends on the size of the property, as well as on the size of the model [57].

Table 5.5 shows the number of cycle iterations, the number of equivalence classes per level, the maximum BDD size for all cycles, and the total time for all iterations of the analysis of model AC in Figure 5.2. The BDD nodes reported is a useful measure of the size of the problem that usually goes up as complexity of the problem increases, but not necessarily. The number of equivalence classes matches Table 5.2. In this dissertation, the model checking verification runs were performed on a 2.8 GHz AMD Opteron CPU with 32 GB of RAM, using the Cadence SMV options `-f` (to compute the reachable states by forward search, restricting model checking to these states) and `-sift` (to use sifting, attempting to improve the variable order).

5.7 Case study with Automotive Features

This section shows how my method and tool *Alfie* is used to check for errors in the functional requirements of four of my non-proprietary automotive feature models, *Collision Avoidance (CA)*, *Emergency Vehicle Avoidance (EVA)*, *Parking Space Centering (PSC)*,

Level	**Iterations**	**Equiv. Classes**	**BDD Nodes**	**Total Time**
4	2	2	1025	2.28s
3	2	2	1030	2.27s
2	2	2	1099	2.27s
1	4	4	10002	2.57s
FIPaths	**# Elements:** 45		136628	27.34s

Table 5.5: Statistics for the analysis of model AC in Figure 5.2

and *Reversing Assistance (RA)* [102]. These features are known as "Active Safety Systems" because they use sensors, cameras, and radar to help the driver control the vehicle. CA helps to prevent or mitigate collisions when driving forward. EVA pulls the vehicle over when an emergency vehicle needs the road to be cleared. PSC assists during perpendicular parking. RA helps prevent or mitigate collisions when reversing. These automotive features are representative in type and complexity of models that I have seen developed in industrial practice [99], but do not include failure modes (*e.g.,* fail-safe states for degraded modes of operation).

Chapter 4 describes how to translate these feature models created in MATLAB's STATEFLOW to SMV. In STATEFLOW, features are hierarchical state machines. Thus, my translation creates one state name variable per hierarchy level of the EFSM in the SMV model. In this case, the constraints on *cs* are expressed over the values of control states at all levels in the hierarchy. No other changes in the definitions for *Alfie* are required. Also, automotive features have only one initial state, because STATEFLOW does not allow multiple initial states. Therefore, Level 3 has the same result as Level 4. Table 5.6 contains information on the size of the translated models in SMV, including history variables needed to verify the invariant property of interest.

	Reachable State Space	**# Trans.**	**# Vars.**	**Max. Vars. range**	**# Basic Control States**
CA	8.24241e+07	26	25	100	9
EVA	1.64848e+08	19	34	100	8
PSC	2.74266e+10	18	34	100	12
RA	6.18181e+07	18	24	100	8

Table 5.6: Information on the size of CA, EVA, PSC and RA

For CA, EVA, PSC and RA, the property checked is that a feature remains disengaged when intended, as specified in each feature's functional requirements. For this analysis, I seeded errors in the feature models [7]. The results of the analysis are summarized in Table 5.7, showing the number of iterations of the model checker, the maximum BDD nodes for all iterations and the time taken to complete all iterations of the analysis. The number of equivalence classes equals the number of iterations for all levels, as *Alfie* generates one equivalence class per iteration.

Level	**CA**			**EVA**		
	Iterations / Equiv. Classes	**BDD Nodes**	**Total Time**	**Iterations / Equiv. Classes**	**BDD Nodes**	**Total Time**
4	2 / 2	452147	4.4s	2 / 2	36053	3.4s
3	2 / 2	452147	4.4s	2 / 2	36053	3.4s
2	2 / 2	453186	4.6s	2 / 2	39991	3.5s
1	2 / 2	465399	5.1s	2 / 2	17857	3.8s

Level	**PSC**			**RA**		
	Iterations / Equiv. Classes	**BDD Nodes**	**Total Time**	**Iterations / Equiv. Classes**	**BDD Nodes**	**Total Time**
4	2 / 2	24220	3.4s	2 / 2	452178	5.0s
3	2 / 2	24220	3.4s	2 / 2	452178	5.0s
2	2 / 2	39329	3.5s	2 / 2	453022	4.5s
1	2 / 2	31507	4.6s	2 / 2	465797	4.9s

Table 5.7: Case study results per level of counterexample equivalence classes

The results from the analysis by level are described in detail for CA, where the invariant checked is that CA remains disengaged when intended. Figure 5.14 illustrates the two errors in the model uncovered by *Alfie*, which generated two distinct counterexamples at all levels exactly matching these two bugs. Consider the information provided per level:

- Level 4 (Distinct Final States), and Level 3 (Distinct Initial and Final States) both generate two distinct counterexamples (two equivalence classes) for CA because there is only one initial state in models generated from STATEFLOW features. Considering the distinct final control states found, one can learn:
  - [(sCA=sENABLED, sENABLED=sENGAGED,sENGAGED=sIDLE)]:
    This counterexample lets the modeller observe that CA becomes engaged when it is meant to be disengaged. Because CA has only one transition leading to the state ENGAGED, the modeller can identify that the error occurs because the condition on t_{16} checks ($\mathsf{Speed} \geq 25$) when it should be ($\mathsf{Speed} > 25$).

[7] The models used in my case study, reported in Chapter 7, contain no errors.

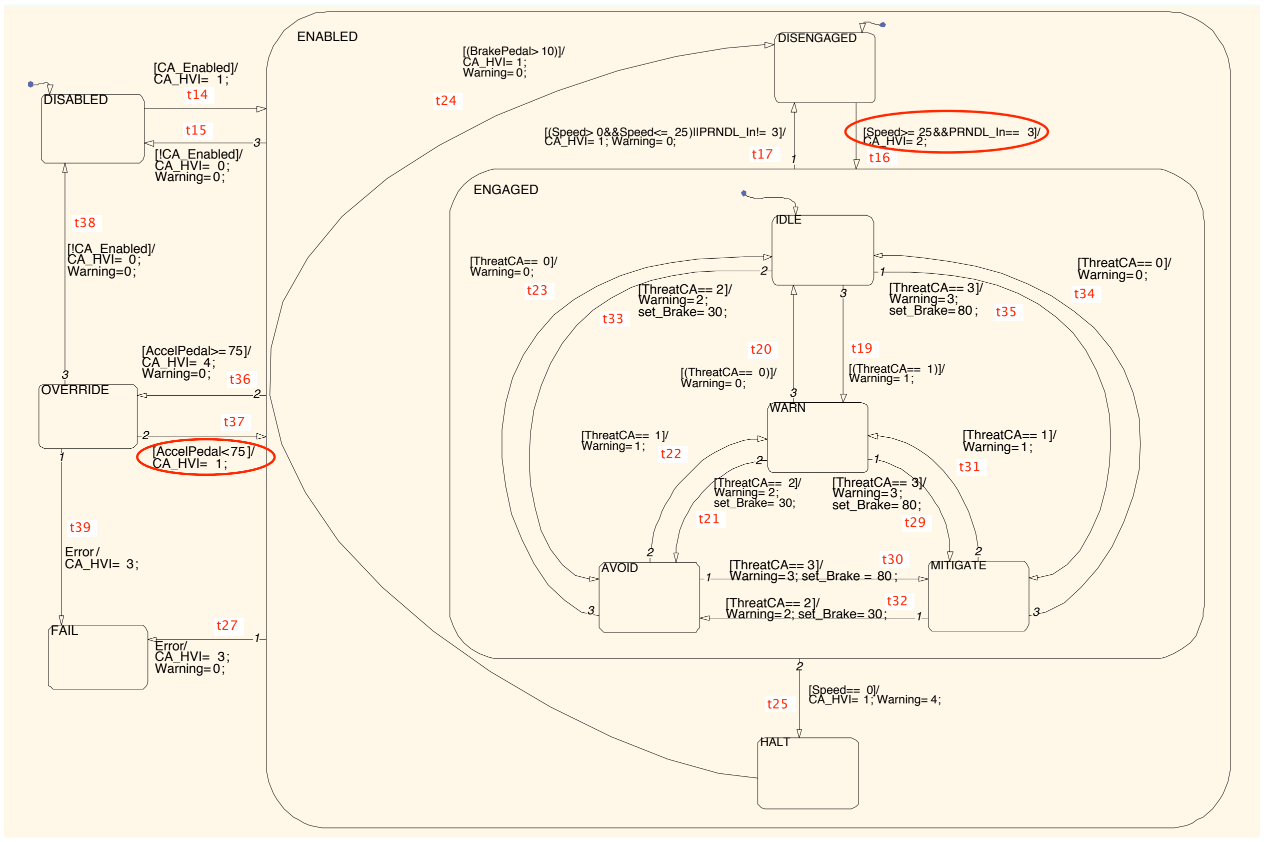

ENABLED
DISABLED
[CA_Enabled]/ CA_HVI= 1;
t14
t15
[!CA_Enabled]/ CA_HVI= 0; Warning=0;
t38
[!CA_Enabled]/ CA_HVI= 0; Warning=0;
OVERRIDE
[AccelPedal>=75]/ CA_HVI= 4; Warning=0;
t36
t37
[AccelPedal<75]/ CA_HVI= 1;
t39
Error/ CA_HVI= 3;
FAIL
t27
Error/ CA_HVI= 3; Warning=0;
[(BrakePedal> 10)]/ CA_HVI= 1; Warning=0;
t24
DISENGAGED
[(Speed> 0&&Speed<= 25)||PRNDL_In!= 3]/ CA_HVI= 1; Warning= 0;
t17
[Speed>= 25&&PRNDL_In== 3]/ CA_HVI= 2;
t16
ENGAGED
IDLE
[ThreatCA== 0]/ Warning=0;
t23
[ThreatCA== 2]/ Warning=2; set_Brake= 30;
t33
[ThreatCA== 3]/ Warning=3; set_Brake=80;
t35
[ThreatCA== 0]/ Warning=0;
t34
t20
[(ThreatCA== 0)]/ Warning= 0;
t19
[(ThreatCA== 1)]/ Warning= 1;
WARN
[ThreatCA== 1]/ Warning= 1;
t22
[ThreatCA== 1]/ Warning= 1;
t31
[ThreatCA== 2]/ Warning= 2; set_Brake= 30;
t21
[ThreatCA== 3]/ Warning=3; set_Brake= 80;
t29
AVOID
[ThreatCA== 3]/ Warning=3; set_Brake = 80;
t30
MITIGATE
[ThreatCA== 2]/ Warning=2; set_Brake= 30;
t32
[Speed== 0]/ CA_HVI= 1; Warning= 4;
t25
HALT

Figure 5.14: Errors in feature CA uncovered by **Alfie**

- [(sCA=sENABLED, sENABLED=sDISENGAGED,sENGAGED=sIDLE)]:

  This counterexample lets the modeller observe that CA must be disengaged but with the wrong conditions, although it is not straightforward to isolate the cause because there are several transitions that lead to state ENABLE while being disengaged. Close inspection and analysis of the counterexample could let the modeller identify that the error occurs when transition t_{37} is taken. The error occurs because transition t_{37} does not check if variable CA_Enabled is *true*, which is necessary for CA to be disengaged.

• Level 2 (Distinct Last Transitions), reports 2 distinct counterexamples for CA. Considering the distinct last transitions, one can learn:

  - [t_{16}]:

    This counterexample shows the last transition taken for CA to become engaged when it is meant to be disengaged. The error occurs because of the condition on t_{16}, as explained above.

  - $[t_{37}]$:
    This counterexample shows the last transition taken for CA to be disengaged but with the wrong conditions. Same error found for counterexample c_2 above.

- Level 1 (Distinct Paths), reports 2 distinct counterexamples for CA. Consider the information provided by Level 1:
  - $[\langle t_{14}, t_{16}\rangle]$:
    This counterexample lets the modeller observe the sequence of transitions that lead CA become engaged when it should be disengaged. The error occurs because of the condition on t_{16}, as explained above.
  - $[\langle t_{14}, t_{36}, t_{37}\rangle]$:
    This counterexample lets the modeller observe the sequence of transitions that lead CA to be disengaged when it should not be. The error occurs because transition t_{36} can be taken regardless of the enabledness of CA. However, t_{37} should check if variable CA_Enabled is *true*, which is a necessary condition for CA to be disengaged.

While all the levels provided the same number of distinct counterexamples in this case study, Level 1 gives the highest confidence that we have isolated the distinct bugs in the model. Level 1 is the only level that differentiates errors that are not in the last transition. For example, for the bug isolated by the second counterexample above, an alternative correction might be to change t_{36}. If there were multiple transitions besides t_{36} entering state OVERRIDE, there could be multiple distinct bugs to isolate, which would be captured by Level 1. We anticipate that a user is likely to either focus on Level 1 to gain as much information as possible about the complete set of bugs or incrementally progress through the levels to see if a higher level produces any additional useful information.

In general, it is not possible to generate a finite set of *FIPaths* because this set is too big and the size of the LTL property becomes so large that the model checker would not terminate. My method allows users to have useful information about the complete set of counterexamples without having to generate all counterexamples, a process that may not be possible, or if possible, take a long time. For example, to know in which control states an error occurs, one consults the results of Level 4; while to know which paths lead to the error, one checks for the results of Level 1. More concretely, to show the reduction that my method accomplishes, I explain how many data variant paths are represented by the equivalence class $p{=}\langle t_{14}, t_{16}\rangle$ reported by Level 1 for model CA, and illustrated in Figure 5.15. The input variables used as triggering conditions in transitions restrict the values that these variables can take. However, the input variables in CA have very large range values, *e.g.,* Speed, ranging from 0 to 100 as one can expect (in a slow car). Therefore, the number of ways the input values can vary is really large, as shown in the numbers on top of the circles in Figure 5.15. These numbers are the data variations allowed at each step of path p. The total number of data variant counterexample paths for the equivalence class p is 7.822×10^{21}, which does not include looping variants of path p. Even though some model checkers are able to detect irrelevant variables using techniques such as cone of influence reduction, having variables like Speed, as shown in this example, would still produce a great number of counterexamples.

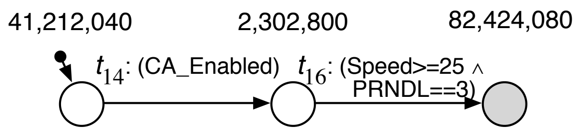


Figure 5.15: Number of data variant paths in equivalence class $[\langle t_{14}, t_{16}\rangle]$

5.8 Related Work

To the best of my knowledge, my approach is the first to generate and summarize the set of all counterexamples on-the-fly by modifying the property. Some approaches use a modified version of a model checking algorithm to generate all counterexamples. In addition to SPIN [93], which can generate all counterexamples by continuing the state space search after the property fails, Copty *et al.* create a model checking engine that generates a BDD representing all counterexamples of a given length, and in a post-processing step, annotate these counterexamples to help diagnose and fix a reported failure [60].

Many approaches do not necessarily generate all counterexamples while attempting to isolate the cause of an error. Jin *et al.* created a model checking algorithm variation that creates annotated counterexamples with events describing fate (inevitability towards the error) or free will (attempt to avoid the error) [98]. Groce and Visser describe an algorithm to find traces that are data variations of a counterexample for Java programs, then process this set of traces to find differences between counterexamples and traces with no error [84]. Sharygina and Peled use a testing tool to generate traces that are related to a counterexample for a software program, where the generated traces or *neighbourhood* of a counterexample might help understand the cause of the error [159]. The neighbourhood of a counterexample may contain traces with no error as well as other counterexamples, but no automatic analysis is done to group or classify counterexamples (if more than one exist). Chechik and Gurfinkel use a modified model checker that generates multiple counterexamples to a property and then in post-processing create a proof-like tree to summarize the data variations from the counterexamples generated [51].

Ball *et al.* modify the model by removing the transition in the counterexample that does not appear in any correct trace so far and then look for another counterexample [17]. This method makes the assumption that the cause of a failure is a single transition. Therefore, all counterexamples that include this transition are in the same group even ones that result in a different failure and the equivalence class of a counterexample is not precisely defined because it depends on the order the traces are explored. By changing the model, they eliminate the possibility of finding a different counterexample that includes the removed transition and therefore, the set of counterexamples generated is not complete.

My work bares some resemblance to the use of model checking to generate test cases of a model that satisfy certain coverage criteria (*e.g.,* [14], [90], [71], [95]). However, in these approaches, one witness (test case) is found for each property and then a new property is created to generate another witness until the coverage criteria is satisfied. some testing approaches use a structural coverage criteria for EFSM-based models. For example, Geist, Hartman *et al.* developed the tool GOTCHA for state and transition coverage [20], [76], while Gargantini and Heitmeyer construct properties from an SCR specification for structural coverage based on guards of a transition [79].

My work might remind the reader of work by Kroening and Weissenbacher to reduce the number of spurious counterexamples (ones that are not real paths in the concrete model) generated as a result of model checking an abstract model using predicate abstraction on loop conditions [117]. Potential looping paths in the counterexample from the abstract model are symbolically unrolled until a "real" path in the concrete model is detected. In my Level 2 (distinct paths), I deem equivalent all EFSM looping variants of a counterexample and do not allow these to be generated again via an LTL property. Because I deal with concrete models, all counterexamples generated by the model checker are real. I do not use predicate abstraction and therefore no refinement/simulation of counterexamples is needed. The grouping I perform is for the sake of eliminating related, real counterexamples from the set of all real counterexamples that are returned to the modeller.

5.9 Summary

In this chapter, I defined a series of levels of equivalence classes that represent the complete set of counterexamples to an invariant in a representative set of counterexamples. This reduced set of equivalence classes is easier to generate and comprehend than the whole set of counterexamples. I have shown how to represent these equivalence classes on-the-fly as LTL properties to be used by a model checker. The equivalence classes I defined are based on the control states and transitions of an extended finite state machine (EFSM). These equivalence classes of counterexamples and their representation as LTL properties can be used for any model that is expressed as an EFSM and that would benefit from a more abstract representation of counterexamples. I demonstrated the reduction produced by my proposed levels of equivalence classes of counterexamples in the verification of an invariant for four automotive feature design models. The strengths of my approach are that it can be used with any LTL model checker, it generates a representation of the complete set of counterexamples, and the summarization of counterexamples occurs on-the-fly meaning that all counterexamples are never generated. The weakness of my approach is that the model checker will repeat work as it analyzes the model again in each iteration of my method, however the number of cycles is often quite small because each of the LTL properties represents a set of counterexamples. Also, my method only reports paths that end in the first configuration that fails the invariant, thus ignoring other potential bugs after the first one. However, this might be enough information to recognize all distinct bugs in a model represented as an EFSM, since the bugs after the first one in the path might be repeats from the ones already recognized, for instance, in cases where the model is allowed to go back to its initial state.

Chapter 6

Detecting and Representing all Different Feature Interactions in Concurrent Features: Generalization of *Alfie*

This chapter describes how to generalize the levels of counterexample equivalence classes as well as the on-the-fly LTL grouping method, presented in Chapter 5, to find a set of counterexamples that is representative of the set of all feature interactions for a pair of concurrent automotive active safety features, without flattening the model. These automotive features are STATEFLOW models, which, unlike an EFSM, can contain composite states (*i.e.,* ordered-compositions). These composite states make a STATEFLOW model respond to an input as a big-step, that is, a sequence of transitions within the components of the feature. Moreover, the pair of STATEFLOW models representing the features must execute concurrently. Therefore, a feature interaction must be detected within the big-step of the combined model, and my definitions and method described in Chapter 5 need to be generalized to achieve this goal.

The present chapter is organized as follows. Section 6.1 provides an overview of the process for finding a set of counterexamples that is representative of the set of all feature interactions for each pair of STATEFLOW models. In this section, I explain both, (1) how to set up the model checking verification to detect a feature interaction when two STATEFLOW models run concurrently, and (2) the changes required to generalize my method and tool *Alfie* so that a set of counterexamples that is representative of the set of all feature interactions for pairs of STATEFLOW models can be automatically generated. Section 6.2 defines the formalism of a STATEFLOW model. The details of the generalization of my method to detect all equivalence classes of feature interactions is described in Section 6.3, while the representation in LTL of the equivalence classes of counterexamples for two concurrent STATEFLOW models (without calculating the flattened cross product of the two models) is given Section 6.4. Section 6.5 considers related work.

6.1 Overview of Generalization

This section starts by reminding the reader of the elements of STATEFLOW models that are different from EFSMs. Then, I re-introduce two STATEFLOW models that help illustrate the concepts related to detection of feature interactions when two models run concurrently: the STATEFLOW model of an air conditioning (AC) system and the STATEFLOW model of a heater (HEATER) system. The rest of the section describes the changes required for my method and tool *Alfie* to be generalized in order to generate a representative set of equivalence classes of feature interactions for a pair of STATEFLOW models.

6.1.1 From EFSM to STATEFLOW Models

Figure 6.1 shows the elements of a STATEFLOW model that are different from an EFSM, and therefore, require changes to my method for EFSMs. The elements of STATEFLOW, and their translation to the language of SMV, were explained in detail in Chapter 4.

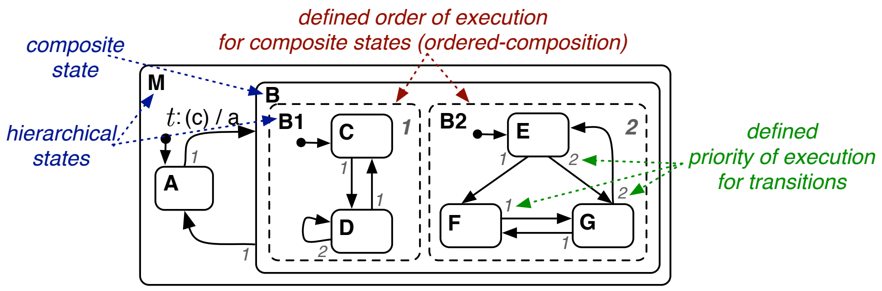


Figure 6.1: Elements of a STATEFLOW model that differ from an EFSM

Beyond an EFSM, a STATEFLOW model can include hierarchical states, such as the main superstate M in Figure 6.1, and composite states, such as ordered-composition B in Figure 6.1. A STATEFLOW model runs in a single thread, and its execution is completely deterministic because *(1)* each sibling in an ordered-composition has an assigned execution order, thus an ordered-composition runs sequentially (*e.g.,* in Figure 6.1, sibling B1 executes before sibling B2 in B), and *(2)* all the transitions in a STATEFLOW model have a defined priority of execution, so non-determinism is disallowed.

When a Stateflow model includes an ordered-composition, its sequential execution is completed in a big-step, with each sibling executing in a small-step. All of the siblings of an ordered-composition respond to the same set of inputs, and there is at most one progressing transition taken in a small-step. In contrast, an EFSM takes only one transition in response to a set of inputs. *mdl2smv* creates a condition called stable (a macro in SMV), which is true when a big-step is concluded for an individual feature.

A feature interaction is detected during the integration of features, when two Stateflow models are running concurrently. Because each Stateflow model can include ordered-compositions, the feature interaction can occur between transitions taken in different small-steps of a big-step. To illustrate the integration of features and detection of feature interactions, consider the Stateflow model of a heater (HEATER) system and the Stateflow model of an air conditioning (AC) system, first introduced in Section 4, and shown here in Figure 6.2 and in Figure 6.3 respectively. Note that, although Chapter 5 showed a flawed AC system, the model AC in Figure 6.2 describes the corrected and intended functionality of the system.

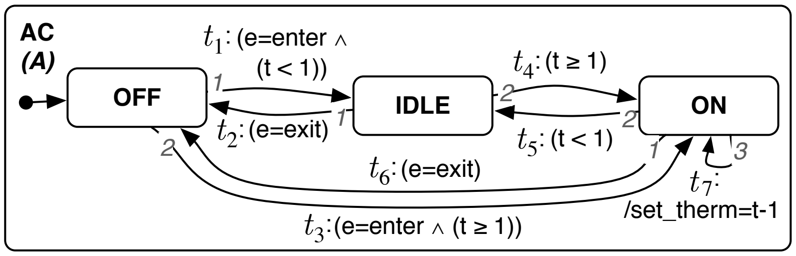


Figure 6.2: Simplified air conditioning AC model

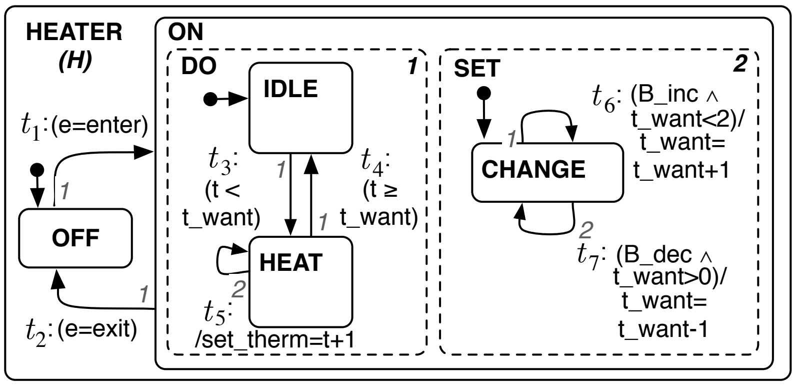


Figure 6.3: Simplified heater HEATER model

Models AC and HEATER share as inputs: the variable e that takes on the values enter and exit, and the variable t that indicates the current temperature and ranges over the values 0..2. HEATER also includes the Boolean variables B_inc and B_dec to indicate if the buttons to increase and decrease the desired temperature are respectively pressed or depressed, and the local variable t_want that holds the desired temperature (ranging over 0..2 and initialized to 1). Both models can set the output variable set_therm to request that the thermostat changes the temperature. The shared actuator of these two features is set_therm, which ranges over the values 0..2.

The rest of this section summarizes the changes needed for my method and tool *Alfie* to detect and represent all equivalence classes of feature interactions.

6.1.2 Detect Feature Interactions in a big-step

A feature interaction is detected when two STATEFLOW models run concurrently. The invariant to check is $\neg$FI because the main goal is to have a system of integrated features with no feature interactions. If a feature interaction exists, then a counterexample is generated.

A feature interaction for active safety features is detected by SMV as conflicting requests to the actuators made via parameterized events, when two models are executing using parallel composition. Parameterized events consist of (1) a variable that has the value associated with the request (prefixed by "`set_`") and (2) a Boolean that represents the presence or absence of the request (with suffix "`_req`"). For instance, a feature interaction should be detected if AC and HEATER make contradictory output requests to set_therm. However, when an ordered-composition is present, it might not be possible to isolate one small-step where the feature interaction occurs because conflicting requests to actuators can occur in different small-steps. Therefore, the detection of feature interactions must be made at the big-step boundary of the concurrent features, *i.e.,* when the two features have generated all their outputs for a given set of inputs. The big-step boundary for the integrated pair of features is marked by the condition sys_stable, which is the conjunction of the stable macros per feature.

Because a feature interaction can only be detected at sys_stable, the Boolean variables associated with parameterized events use remainder semantics to guarantee that the values of the requests to actuators persist throughout the big-step. Then, the definition of the invariant **inv** to detect a feature interaction becomes

$$\textbf{inv} = (\mathsf{sys\_stable} \rightarrow \neg\mathsf{FI}).$$

Example 6.1: To illustrate the detection of a feature interaction when HEATER and AC are running concurrently, suppose that t_want is set to 2 and that the initial temperature t is 1. The LTL property of interest is an *Immediate Same Actuator* FIDP, as defined in Chapter 3, with the value threshold equal to one:

$$\mathrm{G}\ \neg(|\ \mathsf{assign}_{\mathsf{thermo}_\mathsf{A}} - \mathsf{assign}_{\mathsf{thermo}_\mathsf{H}}\ | > 1)$$

meaning that a feature interaction is detected if HEATER and AC request changes to the value of the shared actuator set_therm that differ by 1. Because the request to an actuator is modelled by a parameterized event, the statement of the FIDP must include the Boolean variables that indicate the requests (*i.e.,* variable ending in *i.e.,* _req), as well as the difference between the variables values. By using the FIDP, checked at sys_stable, the property to detect feature interactions becomes:

$$\mathrm{G}(\mathsf{sys\_stable} \rightarrow \neg(\mathsf{A.therm\_req} \wedge \mathsf{H.therm\_req} \wedge |\ \mathsf{set\_therm}_\mathsf{A} - \mathsf{set\_therm}_\mathsf{H}\ | > 1)).$$

Using this invariant, a counterexample trace such as the one in Figure 6.4 is returned by the model checker. This counterexample path shows a feature interaction that occurs when AC requests $\mathsf{set\_therm}_\mathsf{A}$ to be set to 0 as HEATER requests $\mathsf{set\_therm}_\mathsf{H}$ to be set to 2 in *big-step*$_3$. Note that, once set by t_7 and t_5, the values of $\mathsf{set\_therm}_\mathsf{A}$ and $\mathsf{set\_therm}_\mathsf{H}$ remain unchanged and their associated Boolean event requests A.therm_req and H.therm_req remain *true* throughout the big-step thanks to the remainder semantics. In Figure 6.4, the values of transitions taken are in bold and underlined to be easily recognizable.

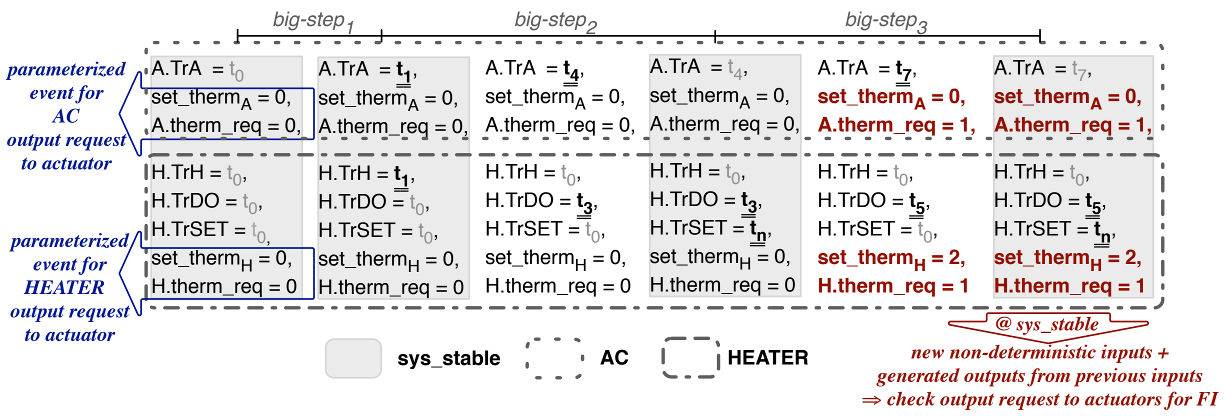


Figure 6.4: Counterexample path for HEATER and AC

The size of a big-step in the integrated model can change dynamically, depending on if any of the features includes an ordered-composition and the number of siblings in an ordered-composition.

Example 6.2: When HEATER and AC run concurrently, the size of the big-step is 1 if AC takes any of its transitions while HEATER takes t_1 or t_2, as in ***big-step***$_1$ in Figure 6.4. However, when the ordered-composition ON in model HEATER is executing, the size of the big-step is 2 because ON takes two transitions to complete its execution, one transition taken in each sibling, as shown in Figure 6.4 for ***big-step***$_2$ or ***big-step***$_3$.

6.1.3 Report Set of Transitions Taken in a big-step

Because of the presence of ordered-compositions, multiple transitions can be taken in a big-step. Because the feature interaction can occur between transitions taken in different small-steps of a big-step, the set of transitions taken in a big-step is needed to understand the feature interaction. Recall that a model that includes an ordered-composition requires more than one transition variable to report all the transitions taken in a big-step, so multiple transition variables are included in the model by *mdl2smv*, as described in detail in Chapter 4. The summary of transitions taken in a big-step must be observed at sys_stable because, at any other small-step, only a partial history of the transitions taken in the big-step is available, or repeated information would be gathered when a feature idles while the other feature completes its execution. These cases are illustrated in the following example.

Example 6.3: When HEATER is executing its ordered-composition ON, it takes two small-steps for HEATER to process the input while AC processes the same input in one small-step. Therefore, AC must hold its information constant by (a) keeping its local variables values unchanged, including transitions and control states, and (b) keeping the value of its actuator variables unchanged. Figure 6.4 shows that AC holds its information constant while idling in the second small-step of ***big-step***$_2$ and the second small-step of ***big-step***$_3$. My process observes the control states reached and transitions taken in a big-step at sys_stable, where all the information about the completed execution of an ordered-composition is available (*e.g.,* ON in HEATER), and where it can recognize that each of t_4 and t_7 were taken only once.

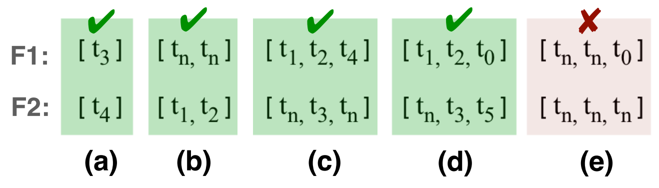
For concurrent components, function *lst_trans_big_step* produces of all transitions taken in a big-step (at sys_stable). All the transitions taken in the big-step must be reported, as the actions of a couple of these transitions are the cause of the feature interaction.

6.1.4 Update Definition of *FIPaths* for Concurrent Models

FIPaths defines a finite representation of the set *CE*, as introduced in Chapter 5. To be able to deal with concurrent models, I need to modify function *progress* and function *reduce_init_config*, which are part of the definition of *FIPaths*, as described next. Note that the function *trunc* always truncates the path at the big-step boundary because the feature interaction is identified at sys_stable.

First, for Stateflow features running concurrently, at least one of the features must be making progress by taking a progressing transition during a big-step to avoid stuttering. Recall that, for a model that contains ordered-compositions, more than one transition variable is necessary to report all the transitions taken in a big-step. For instance, in model HEATER, three transition variables are declared: `TrH`, `TrDO` and `TrSET` by *mdl2smv*, as explained in Chapter 4. Therefore, the definition of *progress* must account for all the transition variables that are part of the features running concurrently. But requesting that all transition variables take a progressing transition is too strong of a condition as, it is valid that one feature idles by taking a *tn* whenever at that step in the execution there is nothing that the feature can do. Moreover, not all transition variables are updated in a big-step, as the parent of some transitions are not active at that point in the execution, *i.e.,* in HEATER, when ON is executing, the only transition variables updated are `TrDO` and `TrSET`.

The macro progress that implements the definition of *progress* should state that it is never the case that, in the same small-step, all transition variables have value *tn* (which is the value for self-looping non-progressing transitions), or the value *t0* (which is the value meaning 'no transition taken' in a sibling). Examples of accepted and rejected big-steps by the macro progress are illustrated in Figure 6.5.



F1:
F2:
✓ $[t_3]$ $[t_4]$ (a)
✓ $[t_n, t_n]$ $[t_1, t_2]$ (b)
✓ $[t_1, t_2, t_4]$ $[t_n, t_3, t_n]$ (c)
✓ $[t_1, t_2, t_0]$ $[t_n, t_3, t_5]$ (d)
✗ $[t_n, t_n, t_0]$ $[t_n, t_n, t_n]$ (e)

Figure 6.5: Example of accepted and rejected steps by macro progress

Second, any counterexample that returns to an initial configuration should be considered equivalent to one that starts from the last initial configuration in the counterexample because the same failed invariant can be reached without this initial loop, where an initial configuration is described in Definition 6.4. Therefore, the definition of *reduce_init_config* must check both triples (one per model) in each configuration to identify an initial configuration loop.

6.1.5 Update Definition of Equivalence Classes

For concurrent models, it would be possible to take the cross-product of the combined models and use my definitions from Chapter 5. However, during my experiments I found that this method does not achieve the same value in summarization for Level 1. Figure 6.6 illustrates the problem with features that do not contain ordered-compositions. If the combined path of two models running concurrently is taken, the sequence of transitions in path

p1 is distinct from the sequence of transitions in path **p2** because of non-progressing transitions taken in between progressing ones. But when considering the sequence of progressing transitions per model, as shown at the bottom of Figure 6.6, both paths are equivalent. Therefore, I refine the notion of equivalence for Level 1 by gathering the sequence of transitions per model and then removing EFSM loops, and creating the conjunction of the resulting information. For Level 1, projection is required to distinguish the transitions that correspond to each model before the equivalence class is generated.

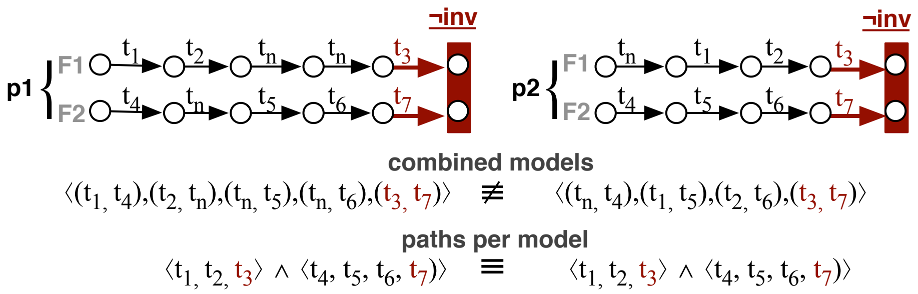

¬inv
p1
F1
F2
p2
combined models
⟨(t1, t4),(t2, tn),(tn, t5),(tn, t6),(t3, t7)⟩ ≢ ⟨(tn, t4),(t1, t5),(t2, t6),(t3, t7)⟩
paths per model
⟨t1, t2, t3⟩ ∧ ⟨t4, t5, t6, t7)⟩ ≡ ⟨t1, t2, t3⟩ ∧ ⟨t4, t5, t6, t7)⟩

Figure 6.6: Equivalence of paths per model, but not in the combined models

For Levels 2-4, the equivalence class is defined on the combined models because these levels concentrate only on a particular point in the path (such as the common initial configuration or the configuration where the invariant fails at sys_stable). These points in the execution can only be defined by both model's concurrent execution, so looking at only one model's execution in isolation would be incorrect.

6.1.6 Remove EFSM loops with respect to big-step boundaries

A path generated from the concurrent execution of two STATEFLOW feature models can be thought of as a sequence of big-steps. The function *reduceEFSM* must check for and eliminate EFSM loops only at sys_stable, *i.e.,* at the big-step boundary, because relevant information about a path in the model, and particularly about the feature interaction, could be eliminated otherwise. These ideas are illustrated in the following example.

Example 6.4: Consider the concurrent execution of models A and B in Figure 6.7, in which a feature interaction occurs when taking transition t_2 along with transition t_5. Recall that in Section 6.1.5, the definition of Level 1 is refined to be applied per feature, and thus, *reduceEFSM* is only applied per feature. If *reduceEFSM* was applied to model B considering small-steps, an EFSM loop from *small-step*$_2$ to *small-step*$_5$ would be detected and eliminated. However, the eliminated EFSM loop contains information about the transition in model B that contributed to the feature interaction in the last big-step, and thus, that information would not be reported. This problem does not occur when applying *reduceEFSM* at sys_stable.

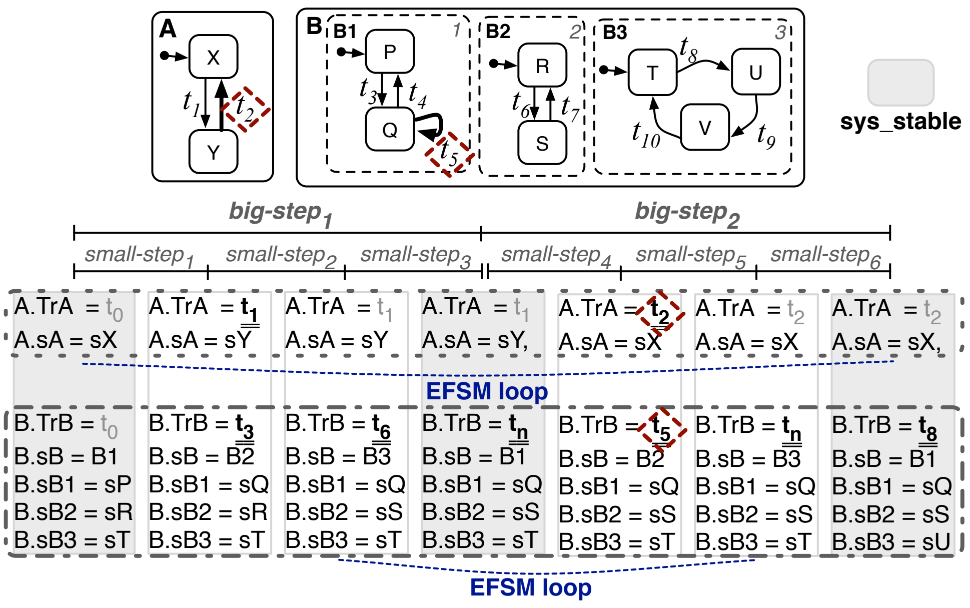


Figure 6.7: Equivalence of paths per model, but not in the combined models

6.1.7 No Environmental Constraints

Automotive active safety features react to the same set of inputs (*i.e.*, same environmental conditions), but there are no other shared variables among features. These features are normally developed by different vendors, or by different teams within a company, in isolation. However, indirect communication between features occurs when the outputs from the features change the environment, affecting the input of the features at a later time. Such dependencies can be modelled by an environment model or constraints. However, I do not use environmental constraints because of the difficulty of modelling accurately an automotive hybrid system. As a result, my method can result in some false positives caused by the unconstrained environment. I view this as better than missing a real counterexample describing a feature interaction because of constraints that are too strict or based on incorrect assumptions, particularly when these assumptions come from different vendors. Analysis of the counterexamples can help construct and refine environmental constraints, as also done by Whalen *et al.* [181]. In Chapter 7, more insight will be given as to the consequences of the lack of environmental constraints in my case study.

6.2 STATEFLOW Model

The syntax of a Stateflow model, which is a hierarchical state machine containing OR-states, AND-states (*i.e.,* ordered-compositions) and labelled transitions, is defined next.

Definition 6.1 *The syntax of a* Stateflow *model consists of a tuple*

$$\langle\ S,\ H,\ CS,\ InitCS,\ V,\ InitV,\ T\ \rangle$$

where

- *S is a finite set of control states.*
- *H is a tree that defines the AND/OR hierarchy of control states, where AND means ordered-composition, basic states are leaves and that includes defaults.*
- *CS is a finite set of trees, each of which is a subtree of H with markings, such that for each AND-state, only one child is marked, and for each OR-state, only one child is present in the tree.*
- *$InitCS$ is a set of trees of initial control states ($InitCS \subseteq CS$).*
- *V is a finite set of typed variables with*
  - *$V = IV \cup OV \cup LV$, where IV is a set of input variables, OV is a set of output (controlled) variables and LV is a set of local variables;*
  - *The sets IV, OV and LV are disjoint.*
- *$InitV$ is a set of sets of initial values for variables. Each set contains one pair for each variable. The first component of the pair is a variable in V, and the second component is the variable's initial assignment drawn from the variable's type.*
- *T is a finite set of* ***progressing*** *transitions. Each $t \in T$, with $t = n : s \xrightarrow{(c)/a} s'$, has*
  - *a name n, accessed by function* **name**(*t*),
  - *a source control state $s \in S$, accessed by function* **src**(*t*),
  - *a destination control state $s' \in S$, accessed by function* **dst**(*t*),
  - *a label of the form $(c)/a$, where (c) is an optional condition on the variables in V called a guard, accessed by function* **guard**(*t*), *and a is an optional set of assignments to variables in $(OV \cup LV)$ called actions, accessed by function* **actions**(*t*). *There are never two assignments to the same variable in a set of actions*[1].

[1]This restriction is to avoid race conditions. I consider these type of assignments to be a design error in Stateflow models.

Next, I provide an overview of the semantics of STATEFLOW. As in the case of EFSM models, every control state implicitly has a single self-looping transition named `tn`, which is taken when no guard on any other transition exiting the state is satisfied. These transitions are called **non-progressing**.

Definition 6.2 *A control state s is* ***active*** *in $cs \in$* ***CS*** *if s is a child of an OR-state and it is present in cs, if s is a child of an AND-state and all of its ancestor AND-states are marked in cs.*

A configuration represents a moment in the execution of two STATEFLOW models running in parallel.

Definition 6.3 *A* ***configuration*** *σ of a pair of* STATEFLOW *features running in parallel is a pair of two triples (one for model A and one for model B, with the syntactical elements superscripted with A or B), where triple $\langle cs^{\mathsf{A}}, n^{\mathsf{A}}, val^{\mathsf{A}} \rangle$ consists of*

- *$cs^{\mathsf{A}} \in CS^{\mathsf{A}}$,*
- *$n^{\mathsf{A}} \subseteq T^{\mathsf{A}} \cup \{\boldsymbol{tn}\}$, a set of names of transitions; several transitions can be taken in a big-step because of ordered-compositions,*
- *val^{A}, a set of pairs, where the first component of each pair is a variable in V^{A}, and the second component is the variable's assignment from the variable's finite type.*

and similarly for model B.

Definition 6.4 *The set of* ***initial configurations*** *σ_{init} of a pair of* STATEFLOW *features running in parallel contains pairs of two triples (one for model A and one for model B), where triple $\langle cs^{\mathsf{A}}_{init}, n^{\mathsf{A}}_{init}, val^{\mathsf{A}}_{init} \rangle$ consists of*

- *$cs^{\mathsf{A}}_{init} \in InitCS^{\mathsf{A}}$,*
- *$n^{\mathsf{A}}_{init}=\emptyset$ as no transitions have been taken,*
- *$val^{\mathsf{A}}_{init} \in InitV^{\mathsf{A}}$, a set of initial assignments to variables.*

and similarly for model B.

A *step* during the execution of two STATEFLOW models running in parallel leads from one configuration to the next following the ideas of small-steps and big-steps introduced in Chapter 4, where a big-step consist of a series of small-steps. There is at most one progressing transition taken in a small-step. All small-steps in a big-step respond to the same set of inputs.

Definition 6.5 *For a* Stateflow *model, a triple $\langle cs, n, val \rangle$ is* **stable** *when either*

- *For all active nested AND-states in cs, only the children that are the first siblings in each ordered-composition are marked as active, or*
- *any transition is taken in a state whose parent is not an ordered-composition.*

Definition 6.6 *For two* Stateflow *models A and B running in parallel, a configuration $(\langle cs^{\mathsf{A}}, n^{\mathsf{A}}, val^{\mathsf{A}} \rangle, \langle cs^{\mathsf{B}}, n^{\mathsf{B}}, val^{\mathsf{B}} \rangle)$ is* **sys_stable** *when the triple of each of the models is stable.*

Definition 6.7 *During the execution of two* Stateflow *models A and B running in parallel,* ***big-step****$(I, (\langle cs_{\mathsf{A}}, n_{\mathsf{A}}, val_{\mathsf{A}} \rangle, \langle cs_{\mathsf{B}}, n_{\mathsf{B}}, val_{\mathsf{B}} \rangle))$ is the result of a series of small-steps, starting from the configuration $(\langle cs_{\mathsf{A}}, \emptyset, val_{\mathsf{A}} \mid_I \rangle, \langle cs_{\mathsf{B}}, \emptyset, val_{\mathsf{B}} \mid_I \rangle)$ where $val_{\mathsf{A}} \mid_I$ means the set of assignments to input variables is input I in val_{A} (and similarly for model B), and ends in a* sys_stable *configuration. The* ***small-step*** *relation of the concurrent model is defined as in Chapter 4 and consists of:*

- *if a model A is not in a stable configuration, it takes a transition such that*
  - *the set of active control states and the assignments to variables in the triple for A of this configuration satisfy the conditions in the transition's guard,*
  - *in the next configuration, the set of active control states is modified to leave the source state and enter the destination of the transition (respecting the hierarchy H following default states), with AND-states nodes marking as active the child that is next sibling in the ordered-composition, the name of the transition taken is added to the set of transition names taken, the set of assignments to local and controlled variables are modified by the transition's actions, and the set of assignments to input variables is the same as the initial set of assignments to variables from inputs I, i.e., $val_{\mathsf{A}} \mid_I$.*

  and similarly for model B.

- *if a model A is in a stable configuration, but model B is not, A idles such that*
  - *in the next configuration, the set of control states, the set of transition names taken, and the set of assignments to variables is maintained the same.*

  and similarly for model B.

The set of transitions taken defined above are implemented in an SMV model using transition variables. The transition name `t0`, which was first introduced in Section 4.5, is used in an SMV model generated by *mdl2smv* to model the empty set assignment to the set of transition names in the initial configuration as well as in the starting configuration of a big-step.

6.3 Counterexample Equivalence Classes for a Pair of Concurrent Components

The process to create the set of all feature interactions in a reduced set of equivalence classes is defined in terms of the set *FIPaths*, as described in Section 5.4 but with the changes to the definition of *progress* and *reduce_init_config* explained in Section 6.1.4.

The following notation is used to describe the levels of counterexample equivalence classes for models running concurrently. These models can contain hierarchical and composite states, therefore, my definitions need to consider sets of states and sets of transitions, unlike the definitions in Chapter 5, where a step in the execution of an EFSM contains one control state and one transition.

- *InitCS:* A set where each element is a set of control states found in the initial configuration. Each element is the union of combinations of sets of initial control states from each model.
- *FICS:* A set where each element is a set of control states found in a reachable configuration in which the invariant fails at sys_stable.
- *FIT*: A set of sets of progressing transitions, where each set of transitions is part of a reachable configuration in which the invariant fails. From each set of transitions, two of the transitions in the set contain actions whose requests to actuators have caused the feature interaction reflected at sys_stable.
- *fst_cs(p):* The set of control states found in the first configuration in path p.
- *lst_cs(p):* The set of control states found in the last configuration in path p.
- *lst_trans_big_step(p):* The set of progressing transitions taken in the last big-step in path p.
- *all_but_last_big_step(p):* The sequence of configurations in path p except for the ones in the last big-step.
- *trans_seq(p):* The sequence of sets of progressing transitions taken in path p at the boundary of each big-step, *i.e.,* at sys_stable.
- $proj_F(p)$: The set of triples, each triple containing the set of control states, the set of transitions and the set of assignments to variables corresponding to feature F in path p, *i.e.,* the information of one of the concurrent features.

- *reduceEFSM(p):* Removes EFSM loops from path p, as explained in Section 6.1.6. An **EFSM loop** is one that reaches the same control states at sys_stable in p more than once.

The notation $[x]$ is used for the equivalence class of x, which consists of the set of equivalent elements of *FIPaths* in the class x. x may be a control state, a path, or a transition, *etc.*

I present my levels of equivalence classes defined over the set *FIPaths*, in order from the most detailed to the least detailed. Table 6.1 shows the equivalence classes of feature interactions created by each of the levels for the integrated model of AC and HEATER.

Level 1	**Level 2**
$[\langle$ ((A.TrA=t_1),(A.TrA=t_4),(A.TrA=t_7)), ((H.TrH=t_1),(H.TrDO=t_3),(H.TrDO=t_5)) $\rangle]$	[A.TrA=t_7, H.TrDO=t_5]
$[\langle$ ((A.TrA=t_3),(A.TrA=t_7)), ((H.TrH=t_1),(H.TrDO=t_3),(H.TrDO=t_5)) $\rangle]$	
$[\langle$ ((A.TrA=t_3),(A.TrA=t_7)), ((H.TrH=t_1),(H.TrDO=t_3,H.TrSET=t_6),(H.TrDO=t_5)) $\rangle]$	

Level 3	**Level 4**
[((A.sA=OFF,H.sH=OFF, H.sON=DO,H.sDO=IDLE,H.sSET=CHANGE), (A.sA=ON,H.sH=ON H.sON=DO,H.sDO=HEAT,H.sSET=CHANGE)]	[(A.sA=ON,H.sH=ON, H.sON=DO,H.sDO=HEAT,H.sSET=CHANGE)]

Table 6.1: Levels of equivalence classes of feature interactions for the integrated model of AC and HEATER

6.3.1 Level 1: Distinct Paths

I expect Level 1 to be the most commonly chosen level for analysis as it captures one representative of each distinct path that contains a feature interaction. However, as explained in detail in Section 6.1.5, for Level 1, I use the sequence of transitions per model after removing EFSM loops, and create the conjunction of the information per model to define the equivalence class for the combined path. For Level 1, projection is required to distinguish the transitions that correspond to each model, and the removal of EFSM loops must be done at the big-step boundary, *i.e.,* at sys_stable. Thus, all the paths that have the same transitions in its last big-step and that have the same sequence of progressing transitions after removing EFSM loops in each feature's path are considered equivalent[2].

[2]For Levels 1-2, the counterexample path must be of length at least one.

Definition 1: For a pair of features A and B, $\forall p \in \mathit{FIPaths} \bullet$

$$[p] = \{q \in \mathit{FIPaths} \mid$$
$$\mathit{lst\_trans\_big\_step}(q) = \mathit{lst\_trans\_big\_step}(p)$$
$$\wedge\ ((\mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{proj}_A(\mathit{all\_but\_last\_big\_step}(q))))) = (\mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{proj}_A(\mathit{all\_but\_last\_big\_step}(p))))))$$
$$\wedge\ ((\mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{proj}_B(\mathit{all\_but\_last\_big\_step}(q))))) = (\mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{proj}_B(\mathit{all\_but\_last\_big\_step}(p))))))\}$$

There are three equivalence classes at Level 1 for the integrated model of AC and HEATER. For example, the counterexample path illustrated by Figure 6.4 is part of the first equivalence class in Table 6.1 after performing the projections per model and taking the respective sequence of progressing transitions. There were no EFSM loops that needed to be removed in this example. For the second and third equivalence classes, a feature interaction is detected when model AC reaches state ON via transition t_3, while the execution of HEATER is similar to the one shown in Figure 6.4, except that the second configuration in the path of the third equivalence class describes that a progressing transition was taken in sibling SET, requesting the desired temperature to increase. Also, for the last two equivalence classes, EFSM loops had to be removed to arrive at the description shown in Table 6.1.

6.3.2 Level 2: Distinct Last Transitions

All the paths that have the same set of progressing transitions in the last big-step are considered equivalent. Note that t in this definition is a set of progressing transitions.

Definition 2: $\forall t \in \mathit{FIT} \bullet [t] = \{p \in \mathit{FIPaths} \mid \mathit{lst\_trans\_big\_step}(p) = t\}$

There is one equivalence class at Level 2 for the integrated model of AC and HEATER. The counterexample path in Figure 6.4 on page 109 is part of the equivalence class [A.TrA=t_7,H.TrDO=$t5$], which is in fact the suffix in all paths shown as equivalence classes of Level 1.

6.3.3 Level 3: Distinct Initial and Final States

All the paths that have the same set of initial control states and set of final control states are considered equivalent.

Definition 3: $\forall i \in \mathit{InitCS}, \forall s \in \mathit{FICS} \bullet$

$$[i, s] = \{p \in \mathit{FIPaths} \mid \mathit{fst\_cs}(p) = i \wedge \mathit{lst\_cs}(p) = s\}$$

An equivalence class is empty if a set of initial control states in *InitCS* is not the first on a path that leads to a set of control states in *FICS*. There is one non-empty equivalence class at Level 3 for the integrated model of AC and HEATER. The counterexample path illustrated by Figure 6.4 on page 109 is part of this equivalence class.

Level 3 allows the designer to examine conditions on the initial control states and potential errors in the initial variable values that generate a feature interaction.

6.3.4 Level 4: Distinct Final States

All the paths that lead to the same set of final control states are considered equivalent.

Definition 4: $\forall s \in \mathit{FICS} \bullet [s] = \{p \in \mathit{FIPaths} \mid \mathit{lst\_cs}(p) = s\}$

There is one equivalence class at Level 4 for the integrated model of AC and HEATER. The counterexample path illustrated by Figure 6.4 on page 109 is part of this equivalence class.

Level 4 allows the designer to find features that immediately conflict with each other when a feature interaction is detected in the initial state.

6.4 On-the-fly Counterexample Grouping for Concurrent Components

This section shows how the definitions to create a representation of the set of all feature interactions in a reduced set of equivalence classes, shown in Section 6.3, are implemented in my method and tool *Alfie*.

My on-the-fly method to detect and group feature interactions for a pair of active safety features running concurrently, is illustrated in Figure 6.8. As the process described in Chapter 5, my method iteratively: (1) asks SMV to generate a counterexample, (2) creates the counterexample to its equivalence class for the desired level, (3) represents this equivalence class as an LTL expression, (4) creates a new property that is the disjunction of this LTL expression with the invariant and the LTL expressions representing previously generated counterexamples, and (5) repeats the process by re-running the model checker on the same model with the new property. The iterative process runs automatically via scripts and it is repeated until no more counterexamples are found, producing as output one representative counterexample per equivalence class, *i.e.,* one distinct feature interaction per class. I call this set of representative counterexamples *CErep*. By disjuncting an LTL expression of the equivalence class with the property, we disallow the generation of any more counterexamples in that equivalence class. The iterative process that *Alfie* follows is summarized by **Algorithm 2**, presented in Chapter 5. The model must contain the following:

¬FI: Macro that specifies the FIDP that specifies the lack of a feature interaction in the integrated model, as described in Chapter 3, either for same or conflicting actuators. However, only interactions of the *Immediate* type are considered in this dissertation.

progress: Macro that specifies (progressing_trans $\vee$ final_states), a condition indicating that at least one progressing transitions is taken at a small-step unless the model reaches a final control state in the integrated model. The macro progressing_trans is formed by disallowing the conjunction of all transitions variables in the integrated model taking non-progressing transitions. There is one condition for each transition variable name Tr_i in the integrated model. Therefore, the macro progressing_trans has the form $\neg(\bigwedge\ (\mathsf{Tr}_i{=}tn \vee \mathsf{Tr}_i{=}t0))$. For example, the progressing_trans macro for HEATER is $\neg((\mathsf{TrH}_i{=}tn \vee \mathsf{TrH}_i{=}t0) \wedge (\mathsf{TrDO}_i{=}tn \vee \mathsf{TrDO}_i{=}t0) \wedge (\mathsf{TrSET}_i{=}tn \vee \mathsf{TrSET}_i{=}t0))$. Final states are control states that are not source states of any contributing transition.

sys_stable: Macro that specifies when all features (a) have generated all the outputs for the current inputs and (b) are ready to begin with new inputs. sys_stable is the conjunction of the stable macro definitions for each feature.

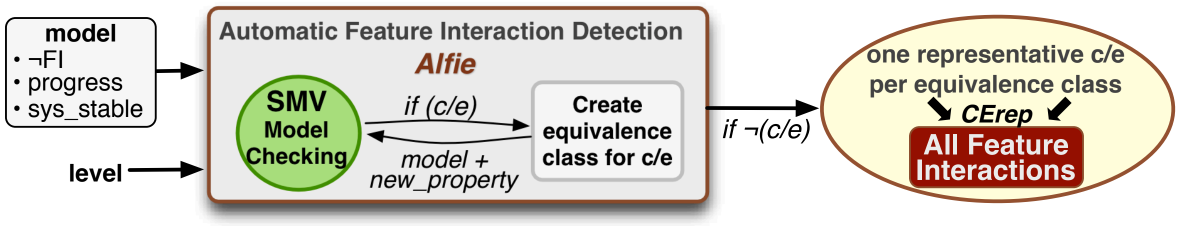


Figure 6.8: On-the-fly grouping level process for concurrent components

Alfie's process starts by generating q, the element of *FIPaths* from a counterexample c. A path q in *FIPaths*, resulting from the execution of multiple concurrent features, is a sequence of big-steps. At each big-step, a sequence of transitions is taken and a set of control state variables has changed their values, reflecting the new state of the system at sys_stable. To limit the model checking exploration, the macro progress is incorporated in the property to be checked. Thus, *Alfie* begins by checking the property

$$\textbf{prop}:\ \mathrm{G}(\mathsf{progress}) \rightarrow \mathrm{G}(\mathsf{sys\_stable} \rightarrow \neg\mathsf{FI})$$

to get the first counterexample. In the rest of this chapter, I will use inv to denote the condition (sys_stable $\rightarrow$ ¬FI). The macro progress included in the property implements the definition of the function *progress* as described in Section 6.1.4, ensuring that the counterexample returned by the model checker contains at least one progressing transition in each big-step. Then, *Alfie* applies the rest of the definition of *FIPaths* to c, generating

Level	LTL property
4	(G (progress)) → ((G (inv)) ∨ (inv U (¬inv ∧ (A.sA=ON ∧ H.sH=ON ∧ H.sON=DO ∧ H.sDO=HEAT ∧ H.sSET=CHANGE))))
3	(G (progress)) → ((G (inv)) ∨ ((A.sA=OFF ∧ H.sH=OFF ∧ H.sON=DO ∧ H.sDO=IDLE ∧ H.sSET=CHANGE) ∧ (inv U (¬inv ∧ (A.sA=ON ∧ H.sH=ON ∧ H.sON=DO ∧ H.sDO=HEAT ∧ H.sSET=CHANGE)))))
2	(G (progress)) →((G (inv)) ∨ (inv U (¬inv ∧ (A.TrA=t_7 ∧ H.TrDO=t_5))))
1	(G (progress)) → ((G (inv)) ∨ (((A.sA=OFF ∧ H.sH=OFF ∧ H.sON=DO ∧ H.sDO=IDLE ∧ H.sSET=CHANGE) ∧ (inv U ((A.TrA=t_1) ∧ (inv U ((A.TrA=t_4) ∧ (inv U (¬inv ∧ (A.TrA=t_7)))))))) ∧ ((A.sA=OFF ∧ H.sH=OFF ∧ H.sON=DO ∧ H.sDO=IDLE ∧ H.sSET=CHANGE) ∧ (inv U ((H.TrH=t_1) ∧ (inv U ((H.TrDO=t_3) ∧ (inv U (¬inv ∧ (H.TrDO=t_5))))))))))

Table 6.2: LTL properties per level of equivalence classes for counterexample shown in Figure 6.4, which illustrates a feature interaction for AC and HEATER

the element q: the function *trunc* creates the subpath of the counterexample ending in the first configuration that fails the invariant, while the functions *reduce_config_loops*, *reduce_init_config* and *reduce_vals* are applied to the resulting subpath. For each element q of *FIPaths*, *Alfie* creates an LTL expression according to the desired level, L, that is added to the invariant as a disjunction, thus generating the property to check next

$$\textbf{prop\_L}\text{: } G(\mathsf{progress}) \rightarrow ((G(\mathsf{inv})) \vee L).$$

As in the process described in Section 5.6, *Alfie* ensures that the antecedent of the property **prop** is not vacuously satisfied by verifying the property EG progress before generating results for any level of counterexample equivalence classes.

Next, I explain in order from least complex to most complex the LTL expression that represents the equivalence class of an element of *FIPaths* according to the desired level. These LTL expressions are illustrated in Table 6.2 for the counterexample shown in Figure 6.4, where the LTL expression L added per level is highlighted.

6.4.1 Level 4: Distinct Final States

For a path $q \in$ *FIPaths*, in Level 4 *Alfie* adds to the invariant a disjunction with an LTL expression L that has the value of the set of control states in the last configuration of q (which is an element of *FICS*), generating the following property with L highlighted:

$$\textbf{prop\_L4}\text{: } (\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \ (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \mathit{lst\_cs}(q)))\ ).$$

prop_L4 forces the model checker to find another set of control states in the first configuration that fails the invariant. The process concludes when all the sets of control states in *FICS* have been discovered.

The justification of the loop-invariant for *Alfie* with respect of Level 4 for concurrent components is the same as in Chapter 5 for Level 4, but in this case, the definition of *lst_cs* returns a set of control states in *FICS*.

6.4.2 Level 3: Distinct Initial and Final States

For a path $q \in \mathit{FIPaths}$, in Level 3 *Alfie* adds to the invariant a disjunction with an LTL expression L describing the set of control states in the initial configuration of q, and the set of control states in the last configuration of q (which is an element of *FICS*), generating the following property with L highlighted:

$$\textbf{prop\_L3}\text{: } (\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \ (\mathit{fst\_cs}(q) \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \mathit{lst\_cs}(q))))\ ).$$

Over multiple paths, *Alfie* groups all final control states with the same initial control states together in a disjunction with the invariant property for brevity of the LTL expression. In the next iteration, the model checker searches for paths that either start with the same set of initial control states, but end at a different set of control states that fail the invariant, or start with a different set of initial control states and end at a set of control states where the invariant fails. The process concludes when all combinations of sets of control states in *InitCS* that reach a set of control states in *FICS* have been discovered.

The justification of the loop-invariant for *Alfie* with respect of Level 3 for concurrent components is the same as in Chapter 5 for Level 3, but in this case, the definition of *fst_cs* returns a set of control states in *InitCS* and *lst_cs* returns a set of control states in *FICS*.

6.4.3 Level 2: Distinct Last Transitions

For a path $q \in \mathit{FIPaths}$, in Level 2 *Alfie* adds to the invariant a disjunction with an LTL expression L that has the value of the set of transitions taken in the last big-step in q (which leads to a set of control states in *FICS*) generating the following property with L highlighted:

$$\textbf{prop\_L2}\text{: } (\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \ (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \mathit{lst\_trans\_big\_step}(q)))\ ),$$

prop_L2 forces the model checker to find other set of transitions that leads to a set of control states in *FICS*. The process concludes when all the sets of transitions in *FIT* are discovered.

The justification of the loop-invariant for *Alfie* with respect of Level 2 for concurrent components is the same as in Chapter 5 for Level 2, but in this case, the definition of *lst_trans_big_step* returns a set of transitions taken in the last big-step of q.

6.4.4 Level 1: Distinct Paths

For a path $q \in \mathit{FIPaths}$, in Level 1 *Alfie* adds to the invariant a disjunction with an LTL expression L that is the conjunction of two LTL expressions, one per model. Each of the LTL expressions makes the model checker accept any path with the same sequence of transitions as the path followed in that model, and all EFSM looping variations of that path, except that the set of transitions in the last big-step must be the same in all paths.

The LTL expression that recognizes all EFSM looping variants of a path is constructed using the LTL operator Until, similarly to the process followed in Section 5.6.4. Through the use of the Until operator, the paths in each model with EFSM loops whose states all satisfy the feature interaction property at sys_stable, are included in this equivalence class. The LTL expression for one of the models must be conjuncted to the LTL expression of the other component, since the sequence of transitions in both components happen simultaneously. Then, the conjunction, *i.e.,* the LTL expression L, should be added to the property as a disjunction with the invariant, thus generating the following property with L highlighted:

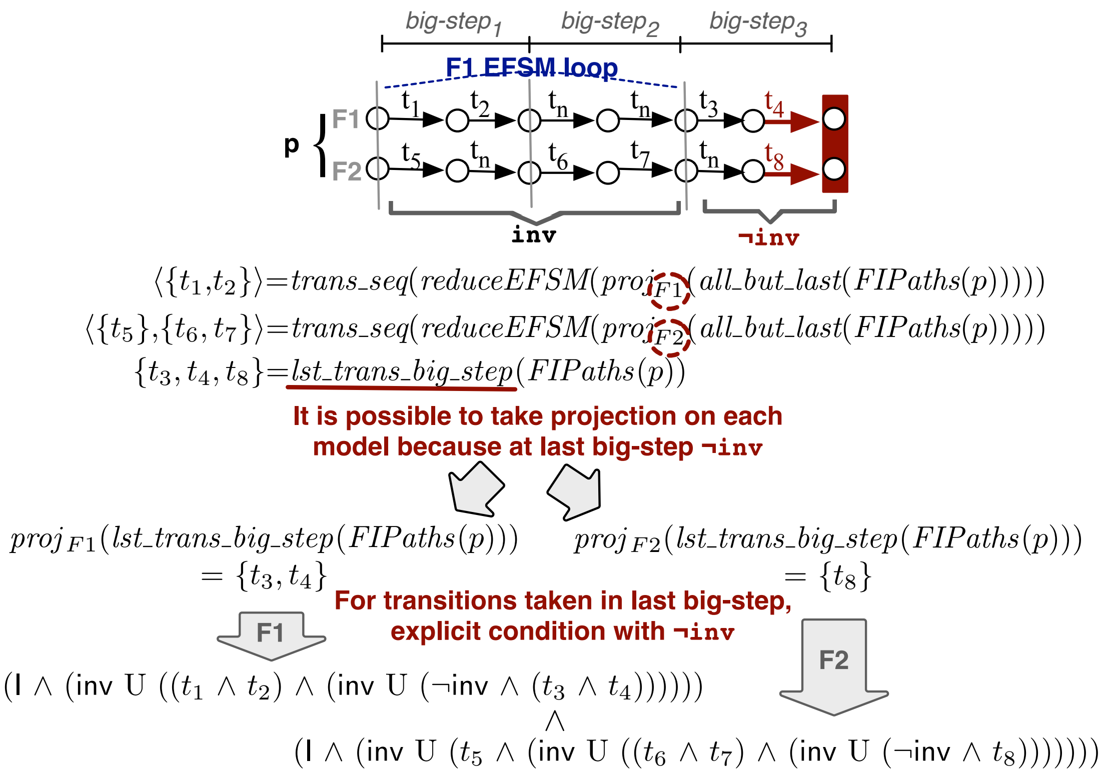
$$\begin{aligned}\textbf{prop\_L1}\text{: } & (\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \\ & ((\mathit{fst\_cs}(q) \wedge (\mathsf{inv}\ \mathrm{U}\ (tA_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (tA_{\mathsf{last}-1} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge tA_{last})))))))\\ & \quad \wedge \\ & (\mathit{fst\_cs}(q) \wedge (\mathsf{inv}\ \mathrm{U}\ (tB_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (tB_{\mathsf{last}-1} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge tB_{\mathsf{last}}))))))))),\end{aligned}$$

where

$$\begin{aligned}\{\ (\langle tA_1,\cdots,tA_{\mathsf{last}}\rangle,\langle tB_1,\cdots,tB_{\mathsf{last}}\rangle)\ \bullet \\ \langle tA_1,\cdots,tA_{\mathsf{last}-1}\rangle &= \mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{proj}_A(\mathit{all\_but\_last\_big\_step}(q)))),\\ \langle tB_1,\cdots,tB_{\mathsf{last}-1}\rangle &= \mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{proj}_B(\mathit{all\_but\_last\_big\_step}(q)))),\\ tA_{\mathsf{last}} &= \mathit{proj}_A(\mathit{lst\_trans\_big\_step}(q)),\\ tB_{\mathsf{last}} &= \mathit{proj}_B(\mathit{lst\_trans\_big\_step}(q))\ \}.\end{aligned}$$

prop_L1 forces the model checker to accept the paths that are described in the properties for each component.

Note that the LTL expression per model only considers the transitions for one feature, even though the equivalence classes for Level 1 are defined in terms of *lst_trans_big_step*, which is the set of transitions of the combined model, because each LTL expression (a) explicitly indicates that the transitions in the last big-step are the ones in which the invariant fails, and (b) specifies that any other transitions in the sequence satisfy the invariant. Therefore, even if the projection per model is taken for the transitions returned by *lst_trans_big_step*, the projected transitions are still representing the common big-step in which the invariant fails. These ideas are illustrated in Figure 6.9, where the LTL expression describing the sequence of transitions for F1 conjuncted with the LTL expression describing the sequence of transitions for F2 define the equivalence class for path p. Both features have to take paths of the same length to the configuration that fails the invariant.


big-step1
big-step2
big-step3
F1 EFSM loop
p
F1
F2
t1
t2
tn
tn
t3
t4
t5
tn
t6
t7
tn
t8
inv
¬inv
It is possible to take projection on each model because at last big-step ¬inv
F1
For transitions taken in last big-step, explicit condition with ¬inv
F2

$$\langle\{t_1,t_2\}\rangle = trans\_seq(reduceEFSM(proj_{F1}(all\_but\_last(FIPaths(p)))))$$

$$\langle\{t_5\},\{t_6,t_7\}\rangle = trans\_seq(reduceEFSM(proj_{F2}(all\_but\_last(FIPaths(p)))))$$

$$\{t_3,t_4,t_8\} = \underline{lst\_trans\_big\_step}(FIPaths(p))$$

$$proj_{F1}(lst\_trans\_big\_step(FIPaths(p))) = \{t_3,t_4\} \qquad proj_{F2}(lst\_trans\_big\_step(FIPaths(p))) = \{t_8\}$$

$$(\mathsf{I} \wedge (\mathsf{inv}\ \mathrm{U}\ ((t_1 \wedge t_2) \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge (t_3 \wedge t_4))))))$$
$$\wedge$$
$$(\mathsf{I} \wedge (\mathsf{inv}\ \mathrm{U}\ (t_5 \wedge (\mathsf{inv}\ \mathrm{U}\ ((t_6 \wedge t_7) \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge t_8)))))))$$

Figure 6.9: LTL expressions per model for concurrent path p

The process concludes when all combinations of distinct paths per model, which are differentiated by the set of transitions in the last big-step, *i.e.,* transitions in *FIT*, have been discovered, including looping variants with EFSM loops that all satisfy the invariant in the sequence per model. The motivation for differentiating paths with respect to their last set of transitions is similar to the motivation given in Section 5.6.4. The justification of the loop-invariant for *Alfie* with respect of Level 1 is described next.

***Justification of Alfie loop-invariant*:**

(⇒) $\forall p \in CE \bullet (p \models \textbf{prop\_L1} \Rightarrow \exists c \in CErep \bullet p \in [c])$

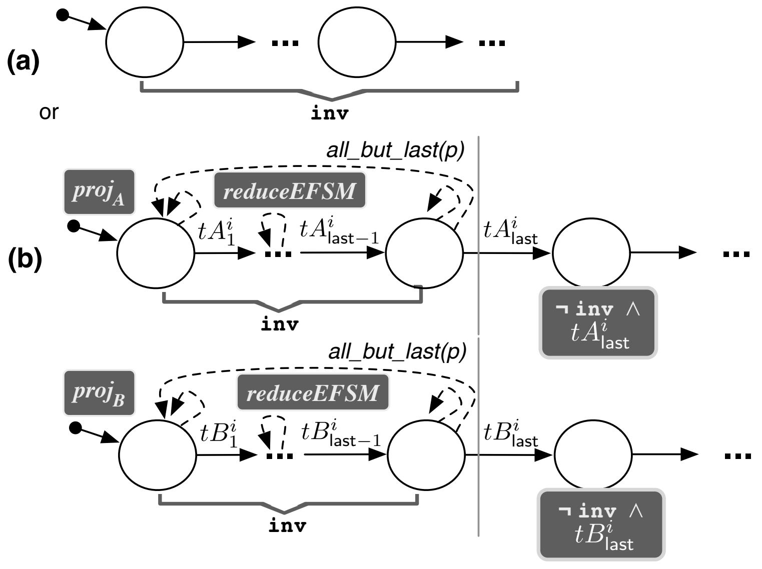
For Level 1, property **prop_L1** has the form:

$$\begin{aligned}
&(\mathrm{G}\ (\mathsf{progress})) \rightarrow ((\mathrm{G}\ (\mathsf{inv})) \vee \\
&\quad ((\mathsf{I}_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (tA^1_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (tA^1_{\mathsf{last}-1} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge tA^1_{last})))))))\\
&\quad \wedge (\mathsf{I}_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (tB^1_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (tB^1_{\mathsf{last}-1} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge tB^1_{\mathsf{last}}))))))))\\
&\qquad \vee \ldots \vee\\
&\quad ((\mathsf{I}_k \wedge (\mathsf{inv}\ \mathrm{U}\ (tA^k_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (tA^k_{\mathsf{last}-1} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge tA^k_{\mathsf{last}})))))))\\
&\quad \wedge (\mathsf{I}_k \wedge (\mathsf{inv}\ \mathrm{U}\ (tB^k_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (tB^k_{\mathsf{last}-1} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge tB^k_{\mathsf{last}})))))))))
\end{aligned}$$

where

$$\begin{aligned}
\{\ ((\langle tA^i_1,\cdots,tA^i_{\mathsf{last}}\rangle,\langle tB^i_1,\cdots,tB^i_{\mathsf{last}}\rangle) \mid \exists\ c_i \in CErep \bullet&\\
\langle tA^i_1,\cdots,tA^i_{\mathsf{last}-1}\rangle &= trans\_seq(reduceEFSM(proj_A(all\_but\_last(FIPaths(c_i))))),\\
\langle tB^i_1,\cdots,tB^i_{\mathsf{last}-1}\rangle &= trans\_seq(reduceEFSM(proj_B(all\_but\_last(FIPaths(c_i))))),\\
tA^i_{\mathsf{last}} &= proj_A(lst\_trans\_big\_step(FIPaths(c_i))),\\
tB^i_{\mathsf{last}} &= proj_B(lst\_trans\_big\_step(FIPaths(c_i)))\ \}.
\end{aligned}$$

If $p \in CE$ satisfies **prop_L1**, then p must be of one of two forms:



For case **(a)**, p is not in CE. For case **(b)**, a path p has (b1) both $\langle tA^i_1,..,tA^i_{k-1}\rangle$ and $\langle tB^i_1,..,tB^i_{k-1}\rangle$ as the sequence of transitions resulting after removing any EFSM loops per model from the element of *FIPaths* of p (excluding the last big-step), returned respectively by $trans\_seq(reduceEFSM(proj_A(all\_but\_last(FIPaths(c_i)))))$ and $trans\_seq(reduceEFSM(proj_B(all\_but\_last(FIPaths(c_i)))))$, for some c_i, and (b2) t^i_k as the set of transitions that leads to the first control state that fails the invariant, returned by $\mathit{lst\_trans\_big\_step}(\mathit{FIPaths}(c_i))$, but projected into each of the models as tA^i_k and tB^i_k. Therefore, $p \in [c_i]$ for $c_i \in \mathit{CErep}$. $\square$

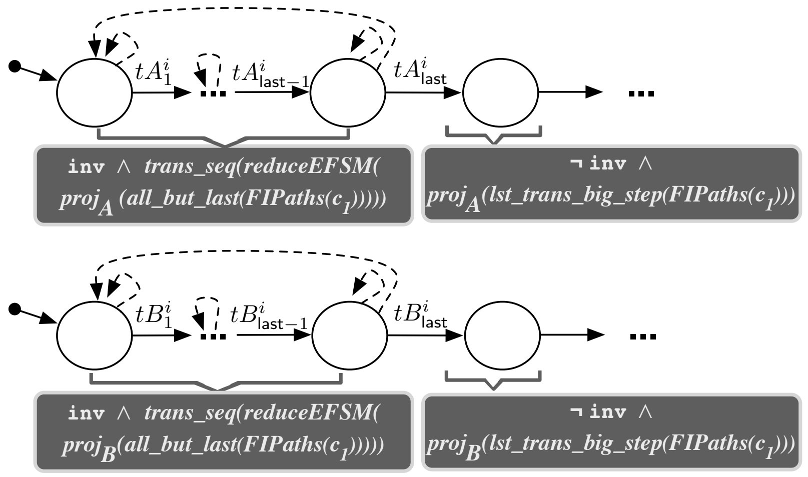
($\Leftarrow$) $\forall p \in \mathit{CE} \bullet (\exists c \in \mathit{CErep} \bullet p \in [c] \Rightarrow p \models$ **prop_L1**)

Let path p be a member of the equivalence class of $c_1 \in \mathit{CErep}$. By **Algorithm 2** for Level 1, when c_1 was generated by the model checker, property **prop_L1** included the LTL expression

$$\begin{gathered}((\mathsf{I}_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (tA^1_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (tA^1_{\mathsf{last}-1} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \\ \mathit{proj}_A(\mathit{lst\_trans\_big\_step}(\mathit{FIPaths}(c_1)))))))))) \\ \wedge\ (\mathsf{I}_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (tB^1_1 \wedge \cdots\ (\mathsf{inv}\ \mathrm{U}\ (tB^1_{\mathsf{last}-1} \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge \\ \mathit{proj}_B(\mathit{lst\_trans\_big\_step}(\mathit{FIPaths}(c_1)))))))))))\end{gathered}$$

disjuncted with the invariant, with both
$\langle tA^1_1, .., tA^1_{\mathsf{last}-1}\rangle = \mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{proj}_A(\mathit{all\_but\_last}(\mathit{FIPaths}(c_1)))))$, and
$\langle tB^1_1, .., tB^1_{\mathsf{last}-1}\rangle = \mathit{trans\_seq}(\mathit{reduceEFSM}(\mathit{proj}_B(\mathit{all\_but\_last}(\mathit{FIPaths}(c_1)))))$.

If $p \in [c_1]$, p must have the form:


tA^i_1
$tA^i_{\mathsf{last}-1}$
tA^i_{last}
inv ∧ trans_seq(reduceEFSM(proj_A (all_but_last(FIPaths(c_1)))))
¬ inv ∧ proj_A(lst_trans_big_step(FIPaths(c_1)))
tB^i_1
$tB^i_{\mathsf{last}-1}$
tB^i_{last}
inv ∧ trans_seq(reduceEFSM(proj_B(all_but_last(FIPaths(c_1)))))
¬ inv ∧ proj_B(lst_trans_big_step(FIPaths(c_1)))

Therefore, path p satisfies property **prop_L1**. $\square$

Table 6.3 shows the number of cycle iterations, the number of equivalence classes per level the maximum BDD size for all cycles, and the total time for all iterations of the analysis of HEATER and AC, shown in Figure 6.2 and in Figure 6.3. The number of equivalence classes per level correspond to the ones shown in Figure 6.1. The detection of all equivalence classes of feature interactions per level for automotive active safety features is presented in Chapter 7.

AC-HEATER	Iterations	Equivalence Classes	BDD Nodes	Time
Level 4	1	1	11137	2.5s
Level 3	1	1	11149	2.5s
Level 2	1	1	13522	2.5s
Level 1	3	3	131657	6.9s

Table 6.3: Statistics for HEATER and AC

6.5 Related Work

This section concentrates on related approaches that have been proposed to deal with the feature interaction problem in the automotive domain.

Lochau and Goltz describe a test case generation method for feature interaction analysis between Statechart-like behavioural models [124]. Even though they describe the input of their analysis as Stateflow models, the semantics defined in the paper do not match Stateflow's, as an AND-state is not truly concurrent as they describe. Moreover, they claim that multiple features can be integrated and run in parallel within Stateflow, by using an AND-state that has all the features to be integrated as its components, which is incorrect. Considering their method as identifying feature interactions for Statechart-like models, the authors generate test cases when two features access (read or write) a shared variable, where each test case contains the input events that enables the feature interaction. In contrast, my method creates all equivalence classes of feature interactions for pairs of Stateflow models, and avoids slight data variations of paths that might be generated by test cases.

D'Souza, Gopinathan *et al.* describe a method to detect and resolve feature interactions using concepts of supervisory control theory and based on a notion of "conflict-tolerance" [67, 68]. Thinking of each feature as a *supervisor* or *controller*, this framework follows the process of Thistle *et al.* [164, 184] and detects an interaction as a blocking controller conjunction. Their resolution strategy involves two parts: (1) Use of a predefined priority of execution among features, and (2) Features are aware of potential conflicts, and therefore, each feature is extended with functionality that enables it to continue operating in the presence of a conflict and when the feature has lower priority. Thus, the feature might resume its controlling behaviour at a later time. One disadvantage of this approach is that it only detects one feature interaction. Also, the priorities used or the conflict-tolerance functionality added to features might not consider all possible conflicts in the system. I believe that the framework described would benefit from using my method to recognize all classes of conflicts in the system before applying the resolution.

There have been some approaches that intend to verify multiple features, although these approaches are not directly applied to the feature interaction problem or the automotive domain. Classen *et al.* [58] describe a novel method, *fSMV*, to apply symbolic model checking in order to verify features that are part of a software product line (SPL). In this framework, all subsets of features in the SPL are checked against the same property, which is advantageous for some applications, such as checking a requirement that should hold in any product derived from the SPL. However, it does not provide any advantage to my analysis because not all pairs of features have feature interactions. Instead, I use pruning, thus reducing the number of cases to be analyzed and verified. Blundell, Fisler *et al.* [27] propose a method to verify features in an SPL that communicate sequentially in a pipe-and-filter manner. They use flow analysis to derive a property describing the data provided from one feature to the next one in the pipe, and based on this information, constraints on successor feature's states are derived. These constraints are then used to perform lightweight checks to determine whether compositions of features violate system-wide properties. In contrast, automotive active safety features are analyzed as they execute concurrently in parallel.

6.6 Summary

In this chapter, I have addressed the problem of detecting feature interactions at design-time in a pair of models running concurrently, where the models control the vehicle's dynamics independently from the driver. My method and tool *Alfie* was generalized in this chapter to generate a set of counterexamples that is representative of the set of all feature interactions for models running concurrently. The features considered are active safety features that help control the dynamics of the vehicle, and they are commonly modelled in MATLAB's STATEFLOW. A feature interaction in the automotive domain occurs when the output requests to the actuators from two features can create contradictory physical forces that could lead to an unsafe outcome. This chapter illustrated my method with the models of an air conditioning (AC) system and a heater (HEATER) system, where a feature interaction is detected when both models send contradictory requests to the common actuator set_therm. My main goal is the application of these results to the automotive domain. Therefore, a case study that validates my method and tool *Alfie*, using automotive active safety features, is presented in Chapter 7.

This chapter showed two main contributions. First, how to detect a feature interactions for a pair of active safety features in the automotive domain using the model checker SMV. Second, a method and tool to automatically produce a representative set of the set of all feature interactions for a pair of features.

Chapter 7

Case Studies

This chapter shows how my method and tool *Alfie*, described in Chapter 6, is used on the "University of Waterloo Feature Model Set" (**UWFMS**) [102] to represent all feature interactions of the UWFMS in a set of equivalence classes. Scalability issues are also addressed in this chapter, showing specific examples.

The present chapter is organized as follows. Section 7.1 provides an overview of the UWFMS, which is a set of non-proprietary automotive active safety features that I created to carry on my case studies and validate my method. Full description of the UWFMS is given in Appendix A. Section 7.2 deals with scalability by describing strategies to partition a large LTL property into two (or more) separate model checking runs that together cover the original property, giving specific details of the partitioning process used in my case studies. Section 7.3 describes the results of my analysis for immediate actuator feature interactions, while Section 7.4 describes the results for conflicting actuators feature interactions. In this chapter, for brevity, I often call a pair of features a "combo". Section 7.5 considers related work on partitioning.

7.1 Overview of the Design Models in the UWFMS

UWFMS features are "Active Safety Systems" that use sensors, cameras, and radar to control the motion control systems of the vehicle (independently from the driver) with the intention of improving safety of the vehicle's occupants, although the intended behaviour of individual features might still lead to unsafe interactions between features. For active safety features, the motion control systems are the actuators Throttle, Brake and Steering. Each design model in the UWFMS was created using a subset of the MATLAB's STATEFLOW language and is described in detail in Appendix A. In this section, a brief overview of the

UWFMS features is provided, as well as, some information on the translated SMV models used in the case study. The seven UWFMS features are the following:

- *Cruise Control* (**CC**) helps cruise while driving forward by controlling the Throttle.
- *Collision Avoidance* (**CA**) helps to prevent or mitigate collisions when driving forward by controlling the Brake.
- *Park Assist* (**PA**) assists during parallel parking by controlling the Throttle, Brake and Steering.
- *Lane Guide* (**LG**) helps keep the vehicle within its lane by controlling the Steering.
- *Emergency Vehicle Avoidance* (**EVA**) pulls the vehicle over when an emergency vehicle needs the road to be cleared by controlling the Throttle, Brake and Steering.
- *Parking Space Centering* (**PSC**) assists during perpendicular parking by controlling the Throttle, Brake and Steering.
- *Reversing Assistance* (**RA**) helps prevent or mitigate collisions when reversing by controlling the Brake.

My UWFMS automotive active safety features are representative in type and complexity of models developed in industrial practice [99], even though they do not include failure modes (*e.g.,* fail-safe states for degraded modes of operation). Because the translated SMV models contain the same level of description as the UWFMS design models, the findings of my analysis can be directly understood in terms of the feature models in STATEFLOW, as validated by the traceability of the results listed in each section. Also, each section describing results from my analysis will discuss manageability, as well as scalability in the cases where partitioning is used.

Table 7.1 contains information on the size of these models in SMV. Total state space refers to the product of possible values for all variables in a model. Later, the number of reachable states is provided when verifying each feature interaction detection property (FIDP) per combo (pair of features). The model checking verification runs were performed on a 2.8 GHz AMD Opteron CPU with 32 GB of RAM.

7.2 Scalability via Partitioning

My method and tool *Alfie* generates a representation of the set of all feature interactions in a reduced set of equivalence classes, producing one counterexample per equivalence class. This process is summarized in **Algorithm 2** on page 88, which iteratively adds to the invariant an LTL expression for each equivalence class seen so far in the process. However, for large models with many distinct counterexamples, such as the active safety features running concurrently, the size of the LTL property including the representation of

	Total State Space	**# Trans.**	**# Vars.**	**Max. Vars. range**	**# Basic Control States**
CC	2.972e+14	18	17	100	13
CA	4.162e+11	26	15	100	9
PA	7.632e+17	20	21	100	12
LG	2.249e+11	19	16	100	9
EVA	1.130e+17	19	20	100	8
PSC	1.881e+19	18	21	100	12
RA	2.472e+09	18	15	100	8

Table 7.1: Information on the size of the UWFMS translated SMV models

equivalence classes already seen becomes too large to model check. LTL model checking depends on the size of the property, in addition to the size of the model [143].

In this section, I describe a strategy to partition the LTL property into two (or more) separate model checking runs that together cover the original property, thus making *Alfie*'s process scalable. Each partition is a condition, *e.g.,* part, added to the LTL property to restrict the search and divide the problem into smaller subproblems, which together, cover the original property.

There are different kinds of partitions that can be used. However, given the experience I gained during the analysis of my case studies, the set of criteria I chose for partitioning is the following, allowing the division of the problem into smaller categories while ensuring that no information is missed in the process. For a Boolean condition part:

(c1) A partition divides the problem disjointly in two: one of the form F(part) and the other of the form G(¬part).

(c2) After a partition is selected, the LTL properties to verify are smaller than the original property.

These characteristics make partitions guide the search to consider only part of the state space and reduce the LTL property to only those equivalence classes in which part is true somewhere along the path (or not). Thus, this process reduces the resources needed to generate more counterexamples when the search without partitions would not complete. More detail on the importance of these criteria will be explained next while describing the partitioning process.

To partition, the process starts by selecting a Boolean condition part, which allows the problem to be split in two, one with F(part) and the opposite with G(¬part), as identified by criteria **(c1)**. The partitioning process is illustrated in Figure 7.1. The partitioning condition is added as a conjunction to the antecedent of the property being verified. These conditions reduce the size of the LTL property, as required by criteria **(c2)**, because when verifying the case F(part), only the LTL expressions for which $c_i \models$ F(part) for $c_i \in$ *CErep*

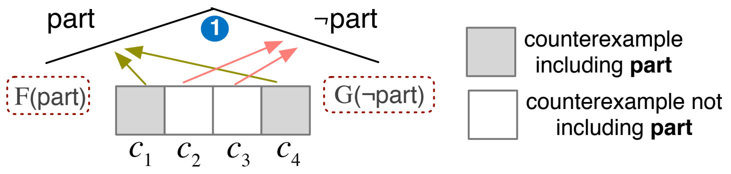

part
¬part
1
F(part)
G(¬part)
c_1 c_2 c_3 c_4
counterexample including **part**
counterexample not including **part**

Figure 7.1: Illustration of a simple partitioning process

are included in the subproblem to be verified. While verifying the case F(part), the model checker only generates counterexamples in which part is true somewhere along the path, thus reducing the model checking problem. The counterexamples that fall in this case, *i.e.,* counterexamples that contain part, become part of the *positive branch.* An example of the initial property to verify in the *positive branch*, after selecting the condition part, for the partitioning process illustrated in Figure 7.1 is:

$$((\mathrm{G}(\mathsf{progress})) \land \mathrm{F}(\mathsf{part})) \rightarrow ((\mathrm{G}(\mathsf{inv})) \lor c_1 \lor c_4).$$

Similar reasoning applies to case G(¬part), where the LTL expressions included in the subproblem are the ones for which $c_k \models \mathrm{G}(\neg\mathsf{part})$ for $c_k \in CErep$. The counterexamples that fall in this case, *i.e.,* counterexamples that do not contain part, become part of the *negative branch.* An example of the initial property to verify in the *negative branch*, after selecting the condition part, for the partitioning process illustrated in Figure 7.1 is:

$$((\mathrm{G}(\mathsf{progress})) \land \mathrm{G}(\neg\mathsf{part})) \rightarrow ((\mathrm{G}(\mathsf{inv})) \lor c_2 \lor c_3).$$

Once the verification within a branch has been exhausted (*i.e.,* no more counterexamples are generated), the verification of the other branch can be performed.

Each partition divides the problem into two parts, however, each subproblem can be further divided if necessary. Figure 7.2 illustrates the process, where three partitions are needed to complete the verification effort. The partitioning process follows a depth first search, starting with the positive branch. The partitioning algorithm is given in **Algorithm 3**, which is recursive to allow for as many partitions as necessary.

In **Algorithm 3**, there are two cases in which the verification cannot continue, and therefore, the results generated by *Alfie* are not complete:

1. When Cadence SMV runs out of memory during verification, returning a segmentation fault (SIGSEGV), as checked in line 2.
2. When a new partition cannot be found, *i.e.,* when there are no more control states and transitions that can be selected from the counterexamples that are part of the current branch, as checked in line 17.

In both cases, *Alfie* returns all the counterexamples that it has been able to generate so far in the process, and continues the verification in the next branch. Thus, the process that *Alfie* follows produces as much information as possible, given the resources available.

Algorithm 3 – *alfie_partition*(m, level, thold, pos_cond, neg_cond, prop_L, *CErep*)

Input: m (model with macros inv, progress, sys_stable), level, thold (time threshold), pos_cond (partitioning elements in the positive condition), neg_cond (partitioning elements in the negative condition), prop_L (condition in the current branch)

Output: *CErep* (set of representative counterexamples, one per equivalence class)

```
1: RUN_SMV(m, prop_L)
2: if (verification runs out of memory) then  ▷ SMV returns SIGSEGV
3:   return CErep  ▷ Verification cannot continue, return current CErep
4: else
5:   if (verification completes within thold) then
6:     if (counterexample c generated) then
7:        ▷ Create LTL expression for c according to desired level
8:       L ← mk_ltl_expr(c, level)
9:       prop_L ← prop_L + L  ▷ Add L as disjunction with the invariant
10:      CErep ← CErep ∪ c  ▷ Add counterexample c to set CErep
11:      alfie_partition(m, level, thold, pos_cond, neg_cond, prop_L, CErep)
12:    else
13:      return CErep  ▷ Verification completed
14:    end if
15:  else ▷ Threshold value reached, partitioning needed
16:    part ← select_partition(CErep, pos_cond)  ▷ Select a partitioning element
17:    if (no new partition can be found) then
18:      return CErep  ▷ Verification cannot continue, return current CErep
19:    else
20:       ▷ Select counterexamples in the new positive and negative branch
21:      (CErep_p,CErep_n) ← select_ce(CErep, pos_cond, part)
22:       ▷ Create LTL expressions for counterexamples in each branch
23:      (prop_Lp,prop_Ln) ← mk_ltl_expr(CErep_p,CErep_n, level)
24:       ▷ Verification within positive branch
25:      pos_cond ← pos_cond ∪ part
26:      p' ← alfie_partition(m, level, thold*2, pos_cond, neg_cond, prop_Lp, CErep_p)
27:       ▷ Verification within negative branch
28:      neg_cond ← neg_cond ∪ ¬part
29:      n' ← alfie_partition(m, level, thold*2, pos_cond, neg_cond, prop_Ln, CErep_n)
30:      return p' ∪ n'  ▷ Verification completed!! Results from both branches
31:    end if
32:  end if
33: end if
```

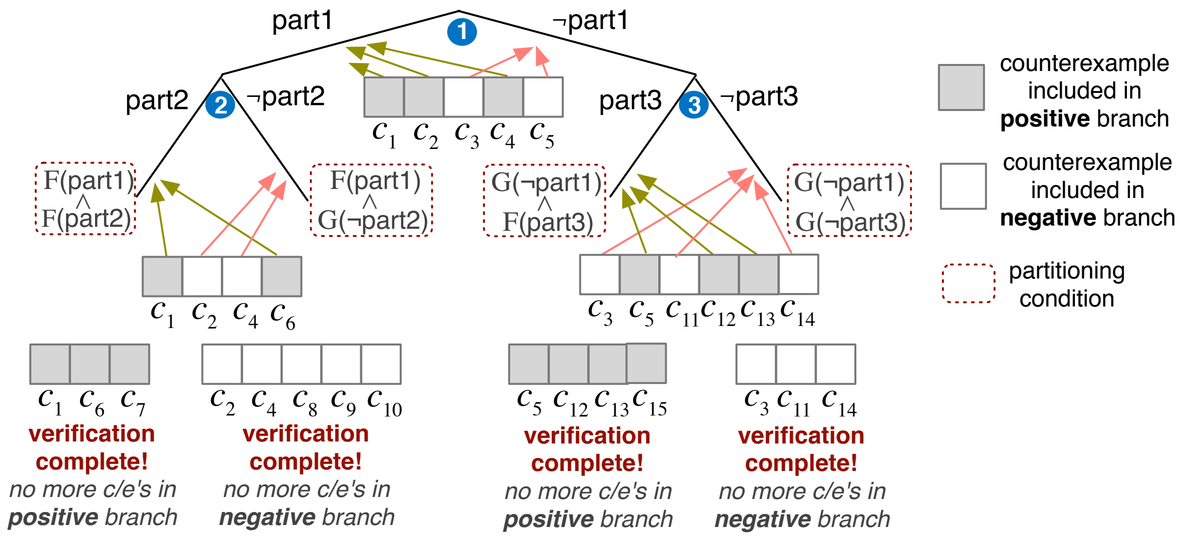


Figure 7.2: Illustration of a process with three partitions

During my experiments, I recognized several options for partitioning elements that can be used in the condition part, separately or using a combination of these elements:

Initial state: In this category, one can consider initial control states of the model or initial configurations of the system. There is at least one initial control state in the model. However, there likely exist multiple initial configurations, considering all distinct initial values of variables. For example, the allowed initial values of the input variable t for model AC in Figure 6.2 are 0, 1, and 2, while the input variable e can take on values enter and exit. Therefore, although there is only one initial control state, there are six potential different initial configurations with respect to the input variables t and e.

FI state: Control states in *FICS*, *i.e.,* states in which the feature interaction property fails. There are none if the model does not have a feature interaction.

Data variables: A restriction on the values that selected variables can take. If no appropriate selection is made, there might not be a significant reduction in the number of cases produced. For instance, for a variable x declared as a range of values 0..10, a good choice for partition could be $x <= 5$ and $x > 5$.

Transitions on path: Selection of a transition or a set of transitions in the model. The partition can be a condition on either (a) the presence of the transition(s) in a path, or (b) the ordered sequence of a set of transitions.

Control states on path: Selection of a control state or a set of control states of the model. The partition can be a condition on either (a) the presence of the control state(s) in a path, or (b) the ordered sequence of a set of control states.

But when is a partition selected? A partition is chosen when the threshold is reached. There are a few heuristic measures that I have discovered that help to determine when to perform partitioning:

- A value threshold on the length of the LTL property being analyzed, so the search is divided when the number of characters of the LTL property is greater than the value threshold set.
- A time threshold on the amount of time elapsed since the current iteration of the model checker started, so that the search is divided when the defined time threshold is reached.

The partitioning measure used in my case studies is a time threshold. The starting time threshold for partitioning is 1 hour, and it doubles every time a new partition is selected because, even if the problem is broken down into two subproblems, I want to ensure each subproblem had as much time as possible to complete. In the example shown in Figure 7.2, both partition 2 and partition 3 would be given 2 hours each to complete their verification. If either partition needs to be partitioned again, that partition would be given 4 hours, and so on. The memory usage at each iteration is set to unlimited, and no other jobs ran while my analysis completed, so SMV was able to use all of the 32 GB of memory available in the server when needed.

In my case studies, control states or transitions are used as partitioning elements. Finding a partitioning element that divides the set *CErep* more evenly considerably reduces the size of the LTL property because only about half of the LTL expressions would be represented in the initial property for the positive branch, and similarly for the initial property in the negative branch. Therefore, when the time threshold has elapsed, *Alfie* constructs a frequency list, *i.e.,* a list of control states and transitions with an associated count of their presence in each of the counterexamples that are part of the current partition. When the first partition is selected, all the counterexamples seen so far in the process are considered. From this frequency list, *Alfie* selects the element whose frequency number is closer to $(n/2)$ and that has not been selected as a partitioning element yet, where n is the number of counterexamples in the current branch. **Algorithm 4** describes the selection of a partitioning element that divides the search more evenly.

The process of partitioning has the disadvantage that some verification effort might be repeated. This occurs when, after the division of the equivalence classes of counterexamples seen so far in the process into a positive and negative branch, a counterexample generated within a branch is reduced to an equivalence class seen previously but not included in the current branch. If a counterexample does not contain a partitioning variable, a, the LTL representation of its equivalence class goes in the negative branch, however, this equivalence class may include paths that have the variable a in a loop. In this case, when results are combined, my method arbitrarily chooses one counterexample to represent the same equivalence class generated from both partitions. For illustration purposes, for the combo CA-EVA to be discussed in Section 7.3.4, consider the equivalence classes:

Algorithm 4 – *select_partition*(*CErep*, parts_in_branch)

```
Input: CErep (set of representative counterexamples in the current branch),
    parts_in_branch (partitioning elements part of the current branch)
Output: new_part (a new partitioning element that divides the search the most evenly)

1: freq_list ← mk_frequency_list(CErep)
2: n ← | CErep |
3:  ▷ Select control state or transition that divides CErep most evenly
4: new_part ← select_part_even(freq_list, n)
5: while (new_part ∈ parts_in_branch) do
6:     freq_list ← freq_list − new_part  ▷ Reduce freq_list so a different element is chosen
7:     new_part ← select_part_even(freq_list, n)  ▷ Select a partition not seen before
8: end while
9: return new_part
```

L1_1 $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35})\rangle, \langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{22})\rangle]$
and
L1_25 $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{29})\rangle, \langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{22})\rangle]$
have been generated when the first partition on $\mathsf{CA\_}t_{29}$ is required. In this case, **L1_25** is included in the positive branch while **L1_1** is included in the negative branch. Then, the following counterexample can be generated by SMV within the positive branch (with the EFSM loop to eliminate highlighted):

$$\langle\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{29}), (\mathsf{CA\_}t_{34}), (\mathsf{CA\_}t_{35})\rangle,$$
$$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{22})\rangle\rangle$$

This counterexample then gets reduced to the previously seen equivalence class **L1_1**, whose LTL expression was not included in the positive branch. Therefore, more iterations than equivalence classes can be reported when partitioning is used. Concrete examples of the advantages of using partitioning during my case studies will be given in the sections discussing scalability for particular pairs of active safety features.

7.3 Same Actuator Feature Interactions

This section shows the results of the detection of immediate feature interactions for same actuators, following the schema shown in Section 3.2.1, and repeated here for convenience:

$$\mathrm{G}\ \neg(|\ \mathsf{assign\_}X_1 - \mathsf{assign\_}X_2\ | > \texttt{value\_threshold})$$

where the schema uses $\mathsf{assign\_}X$ to represent an assignment value request made to actuator X. Because the request to an actuator is modelled by a parameterized event, the applied schema will have the variable that has the value associated with the command, *i.e.,* set_X, and a Boolean that indicates the presence of the command, *i.e.,* X_req. Recall that a counterexample is returned by the model checker when a feature interaction is detected. The actual property checked is shown in the corresponding section describing the pairs of features that interact with respect to an actuator and its threshold. Table 7.2 shows each actuator and its respective threshold considered in the FIDPs for same actuator.

	Actuator	**Value threshold**
Same Actuators	Brake	30
	Throttle	20
	Steering	1

Table 7.2: Elements of multiple feature influence and thresholds for same actuators

The value thresholds should be given by domain experts. For the list shown in Table 7.2, I selected the value thresholds as sufficiently different values to identify feature interactions given the range of possible values of each actuator.

Although the number of combos with potential feature interactions from my non-proprietary UWFMS is $\binom{7}{2}$, because the UWFMS has 7 automotive active safety features, only pairs that influence the same actuator need to be checked. As shown in Figure 7.3, for immediate same actuator feature interactions, only 22 pairs are analyzed. I ran my analysis for same actuator conflicts on all 22 of these combinations, and feature interactions were detected in 4 combos (marked in bold in Figure 7.3). In the rest of this section, the marked pairs are discussed in detail.

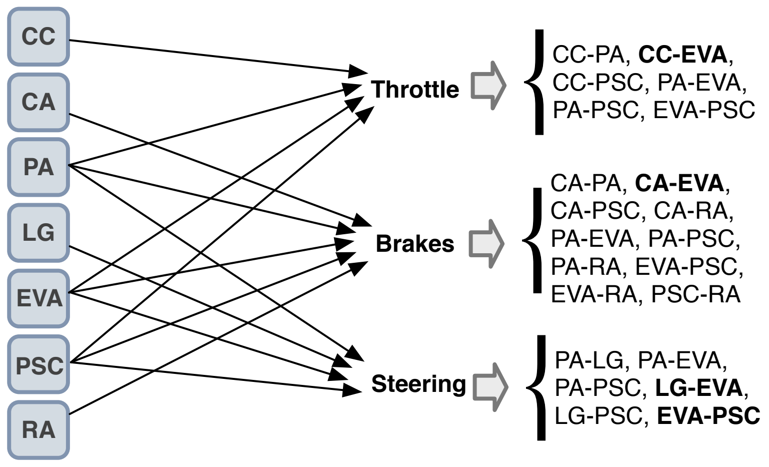

CC
CA
PA
LG
EVA
PSC
RA
Throttle
Brakes
Steering
CC-PA, **CC-EVA**, CC-PSC, PA-EVA, PA-PSC, EVA-PSC
CA-PA, **CA-EVA**, CA-PSC, CA-RA, PA-EVA, PA-PSC, PA-RA, EVA-PSC, EVA-RA, PSC-RA
PA-LG, PA-EVA, PA-PSC, **LG-EVA**, LG-PSC, **EVA-PSC**

Figure 7.3: Diagram of potential and actual same actuator feature interactions in UWFMS combos

7.3.1 Feature Interactions between LG and EVA

The feature interactions detected between LG and EVA for actuator Steering were identified using SMV with the following property as invariant:

$$(\mathsf{sys\_stable} \rightarrow \neg(\texttt{Steering\_req}_{\text{LG}} \wedge \texttt{Steering\_req}_{\text{EVA}} \wedge (\mid \texttt{set\_Steering}_{\text{LG}} - \texttt{set\_Steering}_{\text{EVA}} \mid > 1))).$$

The results of the analysis are summarized in Table 7.3, showing the number of iterations of the model checker, the equivalence classes discovered per level, the maximum BDD nodes for all iterations and the time taken to complete the process. The number of reachable states for the analysis of all levels with combo LG-EVA is 4.43376e+12.

LG-EVA	**Iterations**	**Equivalence Classes**	**BDD Nodes**	**Total Time**
Level 4	1	1	30058	3.56s
Level 3	1	1	30088	3.55s
Level 2	4	4	45682	6.83s
Level 1	9	9	1208347	5m15s

Table 7.3: Same actuator feature interaction analysis results for LG-EVA

The following list shows the equivalence classes reported per level for the combo pair LG-EVA, along with an explanation of the results for this combo. Discussions on traceability and manageability are left to Section 7.3.5. No partitions were needed to complete the analysis for Level 1, thus the number of iterations is the same as the number of equivalence classes and no discussion on scalability is included in this section.

Level 4 – Distinct Final States
This equivalence class lets the modeller observe that a feature interaction occurs when LG and EVA are requesting steering to opposite directions, as a request of steering with -1 indicates that the vehicle shall turn the wheels to the right, while a request of steering with 1 indicates that wheels shall turn to the left.

L4_1: [(LG_sLG=sENABLED,LG_sENABLED=sENGAGED,LG_sENGAGED=sASSIST_RIGHT,
EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sPULLOVER)]

Level 3 – Distinct Initial and Final States
The equivalence class uncovers the same information as Level 4 because there is only one initial state.

Level 2 – Distinct Last Transitions
Each equivalence class in this level lets the modeller observe the combination of transitions that allow LG and EVA to request steering to opposite directions.

L2_1: [LG_t_{28}, EVA_t_{23}]

L2_2: [LG_t_{28}, EVA_t_{26}]

L2_3: [LG_t_{25}, EVA_t_{23}]

L2_4: [LG_t_{25}, EVA_t_{26}]

Level 1 – Distinct Paths

Each equivalence class in this level lets the modeller observe the distinct paths in the models that allow LG and EVA to request steering to opposite directions. For instance, although L1_1–L1_3 take the same path in LG, there are three distinct paths taken in EVA, reaching the state in *FICS* with both, EVA_t_{23} and EVA_t_{26}. In L1_1, the path through EVA reaches a feature interaction via EVA_t_{23}, while in L1_3 the feature interaction is reached via EVA_t_{26}. In contrast, in L1_2 the feature interaction is reached via EVA_t_{26} while going through EVA_t_{23}, but in this case, EVA_t_{23} is in the prefix of the path and it is not taken simultaneously with LG_t_{28} to produce a feature interaction.

L1_1: [⟨(LG_t_{14}), (LG_t_{34}), (LG_t_{28})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23})⟩]

L1_2: [⟨(LG_t_{14}), (LG_t_{34}), (LG_t_{28})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{27}), (EVA_t_{26})⟩]

L1_3: [⟨(LG_t_{14}), (LG_t_{34}), (LG_t_{28})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{26})⟩]

L1_4: [⟨(LG_t_{14}), (LG_t_{34}), (LG_t_{23}), (LG_t_{25})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23})⟩]

L1_5: [⟨(LG_t_{14}), (LG_t_{34}), (LG_t_{23}), (LG_t_{25})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{27}), (EVA_t_{26})⟩]

L1_6: [⟨(LG_t_{14}), (LG_t_{34}), (LG_t_{23}), (LG_t_{25})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{26})⟩]

L1_7: [⟨(LG_t_{14}), (LG_t_{34}), (LG_t_{28}), (LG_t_{26}), (LG_t_{25})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23})⟩]

L1_8: [⟨(LG_t_{14}), (LG_t_{34}), (LG_t_{28}), (LG_t_{26}), (LG_t_{25})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{27}), (EVA_t_{26})⟩]

L1_9: [⟨(LG_t_{14}), (LG_t_{34}), (LG_t_{28}), (LG_t_{26}), (LG_t_{25})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{26})⟩]

Another interesting case to discuss is the relationship between L1_1 and L1_8, where L1_1 appears to be a prefix of L1_8. Thus, this case might at first seem like the ability to reach two feature interactions in path L1_8, but the LTL properties for Level 1 explicitly disallow such situation, and only the last two transitions in L1_8 lead to a feature interaction. Otherwise, the counterexample that generated L1_8 would have been satisfied by the LTL expression describing L1_1. Therefore, although transitions LG_t_{28} and EVA_t_{23} seem to occur simultaneously in L1_8 because of their relative position in the equivalence class, they actually occurred at different steps in the counterexample path from which L1_8 was generated. LG_t_{28} and EVA_t_{23} appear to be at the same position in L1_8 through reductions made by *FIPaths* and *reduceEFSM*. Combo PSC-EVA in Section 7.3.2 would have a similar discussion to the one described in this section.

7.3.2 Feature Interactions between PSC and EVA

The feature interactions detected between CC and EVA for actuator Steering were identified using SMV with the following property as invariant:

$$(\mathsf{sys\_stable} \rightarrow \neg(\texttt{Steering\_req}_{\text{PSC}} \wedge \texttt{Steering\_req}_{\text{EVA}} \wedge \\ \mid \texttt{set\_Steering}_{\text{PSC}} - \texttt{set\_Steering}_{\text{EVA}} \mid > 1)).$$

The results of the analysis are summarized in Table 7.4, showing the number of iterations of the model checker, the equivalence classes discovered per level, the maximum BDD nodes for all iterations and the time taken to complete the process. The number of reachable states for the analysis of all levels with combo PSC-EVA is 8.77652e+11.

PSC-EVA	Iterations	Equivalence Classes	BDD Nodes	Total Time
Level 4	1	1	33192	5.76s
Level 3	1	1	33202	4.75s
Level 2	2	2	68736	6.51s
Level 1	3	3	283048	21.78s

Table 7.4: Same actuator feature interaction analysis results for PSC-EVA

The following list shows the equivalence classes reported per level for the combo pair PSC-EVA. Discussions on traceability and manageability are left to Section 7.3.5. No partitions were needed to complete the analysis for Level 1, thus no discussion on scalability is included in this section.

Level 4 – Distinct Final States

L4_1: [(EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sPULLOVER, PSC_sPSC=sENABLED,PSC_sENABLED=sENGAGED,PSC_sENGAGED=sMOVE_RIGHT)]

Level 3 – Distinct Initial and Final States
The equivalence class uncovers the same information as Level 4 because there is only one initial state.

Level 2 – Distinct Last Transitions

L2_1: [PSC_t_{16}, EVA_t_{23}]

L2_2: [PSC_t_{16}, EVA_t_{26}]

Level 1 – Distinct Paths

L1_1: [⟨(PSC_t_{13}), (PSC_t_{29}), (PSC_t_{16})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23})⟩]

L1_2: [⟨(PSC_t_{13}), (PSC_t_{29}), (PSC_t_{16})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{26})⟩]

L1_3: [⟨(PSC_t_{13}), (PSC_t_{29}), (PSC_t_{16})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{27}), (EVA_t_{26})⟩]

7.3.3 Feature Interactions between CC and EVA

The feature interactions detected between CC and EVA for actuator Throttle were identified using SMV with the following property as invariant:

$$(\mathsf{sys\_stable} \rightarrow \neg(\mathtt{Throttle\_req}_{\mathrm{CC}} \wedge \mathtt{Throttle\_req}_{\mathrm{EVA}} \wedge (\mid \mathtt{set\_Throttle}_{\mathrm{CC}} - \mathtt{set\_Throttle}_{\mathrm{EVA}} \mid > 20))).$$

The results of the analysis are summarized in Table 7.5, showing the number of iterations of the model checker, the equivalence classes discovered per level, the maximum BDD nodes for all iterations and the time taken to complete the process. The number of reachable states for the analysis of all levels with combo CC-EVA is 2.33095e+11 .

CC-EVA	Iterations	Equivalence Classes	BDD Nodes	Total Time
Level 4	3	3	857197	2m28s
Level 3	3	3	857209	2m28s
Level 2	2	2	841251	1m42s
Level 1	4	3	14879504	80m87s

Table 7.5: Same actuator feature interaction analysis results for CC-EVA

The following list shows the equivalence classes reported per level for the combo pair CC-EVA. Discussions on traceability and manageability are left to Section 7.3.5, while the end of this section discusses scalability as one partition was used to generate the results reported for Level 1.

Level 4 – Distinct Final States

L4_1: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING,
CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sINC_SPEED,
EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sCOAST)]

L4_2: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING,
CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sHOLD_SPEED,
EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sCOAST)]

L4_3: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING,
CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sDEC_SPEED,
EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sCOAST)]

Level 3 – Distinct Initial and Final States

The three equivalence classes uncover the same information as Level 4 because there is only one initial state.

Level 2 – Distinct Last Transitions

L2_1: [CC_LOGIC_CONTROL_t_{20}, EVA_t_{24}]

L2_2: [CC_LOGIC_CONTROL_t_{20}, EVA_t_{27}]

Level 1 – Distinct Paths

L1_1: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}), (CC_LOGIC_CONTROL_t_{20})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24})⟩]

L1_2: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}), (CC_LOGIC_CONTROL_t_{20})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{27})⟩]

L1_3: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}), (CC_LOGIC_CONTROL_t_{20})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}),(EVA_t_{26}),(EVA_t_{27})⟩]

Discussion of Scalability for CC-EVA – Same Actuator

One partition was needed for the analysis of Level 1 with combo CC-EVA to complete, when detecting same actuator feature interactions, as illustrated in Figure 7.4. The starting time threshold for partitioning is 1 hour. The process for combo CC-EVA using one partition completed, and it did so in a short time frame (about 80 minutes). These results also illustrate that, when all branches complete the verification, complete results are generated and reported.

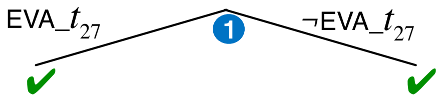


Figure 7.4: Partitions needed during analysis of Level 1 for CC-EVA

7.3.4 Feature Interactions between CA and EVA

The feature interactions detected between CA and EVA for actuator Brake were identified using SMV with the following property as invariant:

$$(\mathsf{sys\_stable} \rightarrow \neg(\texttt{Brake\_req}_{\mathrm{CA}} \wedge \texttt{Brake\_req}_{\mathrm{EVA}} \wedge (\mid \texttt{set\_Brake}_{\mathrm{CA}} - \texttt{set\_Brake}_{\mathrm{EVA}} \mid > 30))).$$

The results of the analysis are summarized in Table 7.6, showing the number of iterations of the model checker, the equivalence classes discovered per level (for Level 1 the results are incomplete, which is indicated by the '+' sign), the maximum BDD nodes for all iterations and the time taken to complete the process. The number of reachable states for the analysis of all levels with combo CA-EVA is 2.63757e+09.

CA-EVA	Iterations	Equivalence Classes	BDD Nodes	Total Time
Level 4	1	1	26394	3.06s
Level 3	1	1	26405	3.06s
Level 2	6	6	37696	6.9s
Level 1	142	44+	119890367	157h59m30s

Table 7.6: Same actuator feature interaction analysis results for CA-EVA

The following list shows the equivalence classes reported per level for the combo pair CA-EVA. Discussions on traceability and manageability are left to Section 7.3.5, while the end of this section discusses scalability because the combo CA-EVA required partitioning to generate the results reported for Level 1 (although the results are incomplete).

Level 4 – Distinct Final States
This equivalence class lets the modeller observe that a feature interaction occurs when CA and EVA are requesting sufficiently different braking forces: CA requests hard braking while EVA requests soft braking.

L4_1: [(CA_sCA=sENABLED,CA_sENABLED=sENGAGED,CA_sENGAGED=sMITIGATE, EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sSLOW)]

Level 3 – Distinct Initial and Final States
The equivalence class uncovers the same information as Level 4 because there is only one initial state in automotive features designed in Stateflow.

Level 2 – Distinct Last Transitions
Each equivalence class in this level lets the modeller observe the combination of transitions that allow CA and EVA to request hard braking and soft braking simultaneously.

L2_1: [CA_t_{35}, EVA_t_{22}]

L2_2: [CA_t_{35}, EVA_t_{25}]

L2_3: [CA_t_{30}, EVA_t_{22}]

L2_4: [CA_t_{30}, EVA_t_{25}]

L2_5: [CA_t_{29}, EVA_t_{22}]

L2_6: [CA_t_{29}, EVA_t_{25}]

Level 1 – Distinct Paths

Each equivalence class in this level lets the modeller observe the distinct paths in the models that allow CA and EVA to request simultaneously hard braking and soft braking. There are many equivalence classes identified in combo CA-EVA because of the many conditions that the features are prepared to react to. For instance, CA can deal with situations in which the vehicle is in an imminent collision course, or where the vehicle could be in a potential collision, or where no potential of collision is detected. When CA is executing, it is possible that several of these threat collision situations can occur, one after the other and in various combinations, thus, making CA react by requesting different degrees of braking force at different times. When CA executes concurrently with EVA, the number of possible situations to react to is very large, as illustrated by the cases listed in this section. However, the incorporation of environmental constraints might help reduce the number of cases generated. For instance, some constraints could be included to disallow the case in which an imminent collision threat is followed in the next step by a mild threat. But by not restricting the environment, my results let the modeller decide if this situation is possible, if it requires a particular resolution or if a constraint in the environment during analysis is required. As discussed in Section 6.1.7, my method generates the knowledge to make the best informed decision as to how to resolve the interactions presented.

L1_1: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{22})⟩]

L1_2: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{26}), (EVA_t_{22})⟩]

L1_3: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{25})⟩]

L1_4: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{27}), (EVA_t_{25})⟩]

L1_5: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{33}), (CA_t_{30})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{22})⟩]

L1_6: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{33}), (CA_t_{30})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{26}), (EVA_t_{22})⟩]

L1_7: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{19}), (CA_t_{21}), (CA_t_{30})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{22})⟩]

L1_8: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{19}), (CA_t_{21}), (CA_t_{30})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{26}), (EVA_t_{22})⟩]

L1_9: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35}), (CA_t_{32}), (CA_t_{30})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{22})⟩]

L1_10: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35}), (CA_t_{32}), (CA_t_{30})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{26}), (EVA_t_{22})⟩]

L1_11: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{19}), (\mathsf{CA}\_t_{29}), (\mathsf{CA}\_t_{32}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{22})\rangle]$

L1_12: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{19}), (\mathsf{CA}\_t_{29}), (\mathsf{CA}\_t_{32}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{26}), (\mathsf{EVA}\_t_{22})\rangle]$

L1_13: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{31}), (\mathsf{CA}\_t_{21}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{22})\rangle]$

L1_14: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{31}), (\mathsf{CA}\_t_{21}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{26}), (\mathsf{EVA}\_t_{22})\rangle]$

L1_15: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{33}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_16: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{33}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{27}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_17: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{19}), (\mathsf{CA}\_t_{21}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_18: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{19}), (\mathsf{CA}\_t_{21}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{27}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_19: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{32}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_20: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{32}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{27}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_21: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{19}), (\mathsf{CA}\_t_{29}), (\mathsf{CA}\_t_{32}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_22: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{19}), (\mathsf{CA}\_t_{29}), (\mathsf{CA}\_t_{32}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{27}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_23: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{31}), (\mathsf{CA}\_t_{21}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_24: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{31}), (\mathsf{CA}\_t_{21}), (\mathsf{CA}\_t_{30})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{27}), (\mathsf{EVA}\_t_{25})\rangle]$

L1_25: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{19}), (\mathsf{CA}\_t_{29})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{22})\rangle]$

L1_26: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{19}), (\mathsf{CA}\_t_{29})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{26}), (\mathsf{EVA}\_t_{22})\rangle]$

L1_27: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{33}), (\mathsf{CA}\_t_{22}), (\mathsf{CA}\_t_{29})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{22})\rangle]$

L1_28: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{22})\rangle]$

L1_29: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{22})\rangle]$

L1_30: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{22})\rangle]$

L1_31: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{30}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{22})\rangle]$

L1_32: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{30}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{22})\rangle]$

L1_33: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{32}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{22})\rangle]$

L1_34: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{32}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{22})\rangle]$

L1_35: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{25})\rangle]$

L1_36: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27}), (\mathsf{EVA\_}t_{25})\rangle]$

L1_37: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{25})\rangle]$

L1_38: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27}), (\mathsf{EVA\_}t_{25})\rangle]$

L1_39: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{25})\rangle]$

L1_40: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27}), (\mathsf{EVA\_}t_{25})\rangle]$

L1_41: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{30}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{25})\rangle]$

L1_42: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{30}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27}), (\mathsf{EVA\_}t_{25})\rangle]$

L1_43: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{32}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{25})\rangle]$

L1_44: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{32}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27}), (\mathsf{EVA\_}t_{25})\rangle]$

Discussion of Scalability for CA-EVA – Same Actuator

The analysis of Level 1 with combo CA-EVA used six partitions when detecting same actuator feature interactions, as illustrated in Figure 7.5. An analysis for CA-EVA without partitions, using Cadence SMV, runs out of memory after generating only 20 counterexamples. The size of the LTL property becomes too large for SMV to complete the verification, and therefore, partitioning is used.

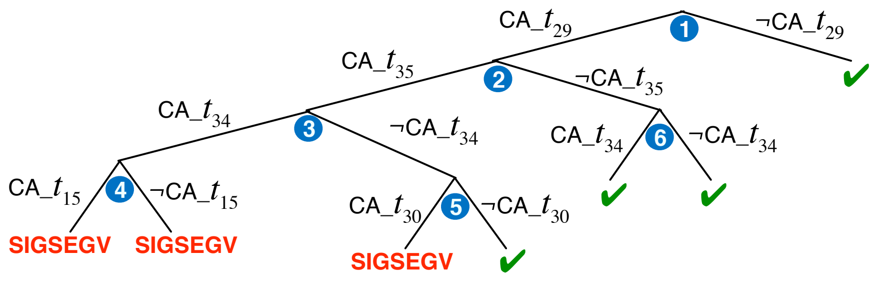


Figure 7.5: Partitions needed during analysis of Level 1 for CA-EVA

During the partitioning process, SMV can use all of the 32 GB of available memory, but there were some branches in which Cadence SMV ran out of memory during verification, marked with SIGSEGV in Figure 7.5. As described in **Algorithm 3**, *Alfie* recovers from these segmentation faults and continues to generate equivalence classes from the next conditional branch. Even though the output from *Alfie* does not generate all the counterexamples representing equivalence classes of feature interactions, because of the branches with SIGSEGV, the process with partitioning is still beneficial by producing 44 counterexamples, which is more than the 20 produced by the process without partitioning.

7.3.5 Discussion of Traceability and Manageability for Same Actuator Feature Interactions

Traceability – For the same actuator combo pairs CA-EVA, LG-EVA and PSC-EVA, in which none of the features include ordered-compositions, the results can be easily traced back to the Stateflow models by (1) following the hierarchy of control states, as the name of the states reported in the counterexample by SMV are unchanged from the Stateflow models (in the results from Level 3 and 4), (2) locating the transition name reported in the counterexample, as these models use one transition variable for the transition taken (in the results from Level 2), and (3) following the sequence of transition names reported in the counterexample from the only initial state (in the results from Level 1).

For the same actuator combo pair CC-EVA, where CC includes an ordered-composition, the results can be traced back to the Stateflow models by (1) following all the hierarchies of control states, as the name of the states reported in the counterexample by SMV are unchanged from the Stateflow models (in the results from Level 3 and 4), (2) following the hierarchies of transition names reported in the counterexample, as model CC includes an ordered-composition, and therefore, it has several transition variables to report the transitions taken in the big-step (in the results from Level 2), and (3) following the sequence of transition names, hierarchically in the case of CC, reported in the counterexample from the only initial state (in the results from Level 1). Following the transition names hierarchically is not too complicated, as the name of the transition variable indicates the superstate in which the transition is located. For instance, $\mathsf{CC\_LOGIC\_CONTROL\_}t_{22}$ indicates that transition t_{22} is within the control state LOGIC_CONTROL of feature CC.

Manageability – The results are manageable as they can be simply listed in each section for all levels. In contrast, an approach showing all counterexamples would have been hard to generate (probably impossible with current resources), and moreover, very hard to understand and analyze. As an illustration of the reduction achieved by my method, Section 5.5 showed in Table 5.3 some of the elements of *FIPaths* that would be classified as data variants by feature designers, compared to the more compact representation of equivalence classes shown in Table 5.2. However, for some combos such as CA-EVA, a very large number of cases is reported, which might be reduced by incorporating some environmental constraints. The challenge is that some of these constraints appear to be specific to a particular feature or combination of features, which would make my method dependent on the set of features that are part of the system, somehow defeating the idea of the generality of my definition. It might be interesting to investigate if it is possible to find a set of environmental constraints that is useful and independent of the features analyzed.

7.4 Conflicting Actuator Feature Interactions

This section shows the results of the detection of immediate feature interactions for conflicting actuators, following the schema shown in Section 3.2.1, and repeated here for convenience:

$$\mathrm{G}\ \neg((\mathsf{assign\_}X > \texttt{value\_threshold}_X) \wedge (\mathsf{assign\_}Y > \texttt{value\_threshold}_Y))$$

where the schema uses $\mathsf{assign\_}X$ and $\mathsf{assign\_}Y$ to represent an assignment value request made to actuator X and actuator Y respectively. Because the request to an actuator is modelled by a parameterized event, the applied schema will have the variable that has the value associated with the command, *i.e.,* $\mathsf{set\_}X$, and a Boolean that indicates the presence of the command, *i.e.,* $X\mathsf{\_req}$, and similarly for actuator Y. A counterexample is returned by the model checker when a feature interaction is detected. The actual property checked is shown in the section describing the pairs of features that interact with respect to a pair of actuators and their thresholds. Table 7.7 shows the pairs of actuators and their respective thresholds considered in the feature interaction detection for conflicting actuators.

	Actuator	**Value threshold**
Conflicting Actuators	Brake	40
	Throttle	30
	Throttle	40
	Steering	0

Table 7.7: Elements of multiple feature influence and thresholds for conflicting actuators

Figure 7.6 shows, for immediate conflicting actuators feature interactions, the 27 pairs that are analyzed by *Alfie*. However, feature interactions were detected in only 4 out of 27 combos (marked in bold in Figure 7.6). In the rest of this section, the marked pairs are discussed in detail.

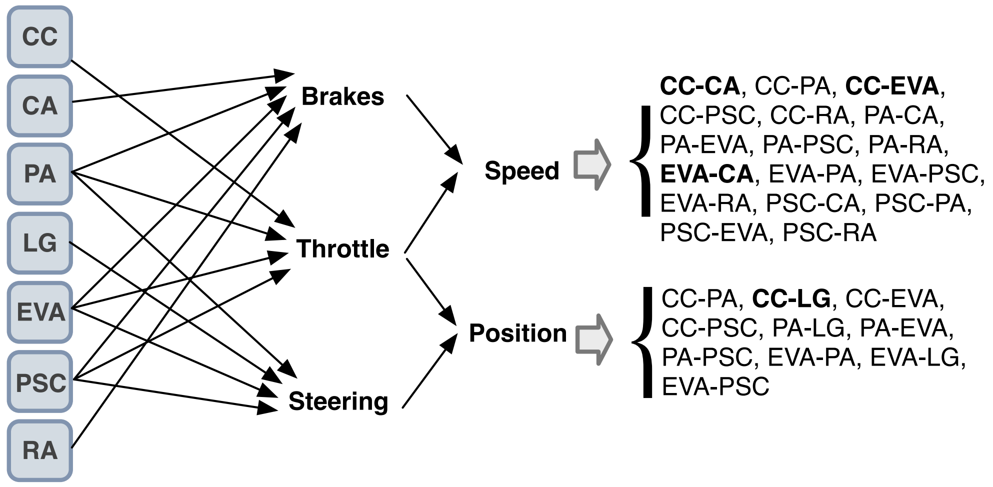

CC
CA
PA
LG
EVA
PSC
RA
Brakes
Throttle
Steering
Speed
Position
CC-CA, CC-PA, **CC-EVA**, CC-PSC, CC-RA, PA-CA, PA-EVA, PA-PSC, PA-RA, **EVA-CA**, EVA-PA, EVA-PSC, EVA-RA, PSC-CA, PSC-PA, PSC-EVA, PSC-RA
CC-PA, **CC-LG**, CC-EVA, CC-PSC, PA-LG, PA-EVA, PA-PSC, EVA-PA, EVA-LG, EVA-PSC

Figure 7.6: Diagram of potential and actual conflicting actuators feature interactions in UWFMS combos

7.4.1 Feature Interactions between CC and EVA

The feature interactions detected between CC and EVA for actuators Throttle and Brake were identified using SMV with the following property as invariant:

$$(\mathsf{sys\_stable} \rightarrow \neg(\mathtt{Throttle\_req}_{\mathrm{CC}} \wedge \mathtt{Brake\_req}_{\mathrm{EVA}} \wedge ((\mathtt{set\_Throttle}_{\mathrm{CC}} > 30) \wedge (\mathtt{set\_Brake}_{\mathrm{EVA}} > 40))).$$

The results of the analysis are summarized in Table 7.8, showing the number of iterations of the model checker, the equivalence classes discovered per level, the maximum BDD nodes for all iterations and the time taken to complete the process. The number of reachable states for the analysis of all levels with combo CC-EVA is 2.33095e+11.

CC-EVA	**Iterations**	**Equivalence Classes**	**BDD Nodes**	**Total Time**
Level 4	3	3	1228888	7m12s
Level 3	3	3	1228899	7m7s
Level 2	3	3	933399	2m25s
Level 1	4	4	8643233	34m7s

Table 7.8: Conflicting actuators feature interaction analysis results for CC-EVA

The following list shows the equivalence classes reported per level for the combo pair CC-EVA. Discussions on traceability and manageability are left to Section 7.4.5. No partitions were needed to complete the analysis for Level 1, thus no discussion on scalability is included in this section.

Level 4 – Distinct Final States

This equivalence class lets the modeller observe that a feature interaction occurs when CC requests an increase in throttle as EVA requests braking.

L4_1: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING, CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sINC_SPEED EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sPULLOVER)]

L4_2: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING, CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sHOLD_SPEED EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sPULLOVER)]

L4_3: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING, CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sDEC_SPEED EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sPULLOVER)]

Level 3 – Distinct Initial and Final States

The three equivalence classes uncover the same information as Level 4 because there is only one initial state.

Level 2 – Distinct Last Transitions

L2_1: [CC_LOGIC_CONTROL_t_{20}, CC_SPEED_SETTING_t_{33}, EVA_t_{23}]

L2_2: [CC_LOGIC_CONTROL_t_{20}, EVA_t_{23}]

L2_3: [CC_LOGIC_CONTROL_t_{20}, EVA_t_{26}]

Level 1 – Distinct Paths

This equivalence class show the different paths that allow CC and EVA request throttle and brake simultaneously. An interesting case to discuss is L1_1 compared to L1_2, as the path in EVA is the same, while the path in CC varies in the last step. The variation occurs when sibling SPEED_SETTING in the ordered-composition of CC takes a progressing transition in the last big-step of L1_2, while the same sibling takes no progressing transition in the other equivalence classes reported in this section. One can argue that L1_1 and L1_2 are not distinct feature interactions, but it is only by analyzing the results that the modeller can decide if the transition taken in sibling SPEED_SETTING is needed or not. With respect to ordered-compositions, an idea to refine the notion of equivalence would be to inspect each sibling in the big-step where the feature interaction is detected and try to identify the sibling that directly requested the action that generated the feature interaction. The potential problem that I anticipate with this refinement is that if another sibling is indirectly contributing to feature interaction, its contribution would be hidden in an equivalence class that only presents information about the one sibling that requested the action on an actuator. For instance, based on this idea for refinement, L1_1 and L1_2 would be put into the same equivalence class, represented by L1_2.

L1_1: $[\langle(\mathsf{CC\_LOGIC\_CONTROL\_}t_{22}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{18}, \mathsf{CC\_SPEED\_SETTING\_}t_{31}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{20}, \mathsf{CC\_SPEED\_SETTING\_}t_{33})\rangle, \langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23})\rangle]$

L1_2: $[\langle(\mathsf{CC\_LOGIC\_CONTROL\_}t_{22}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{18}, \mathsf{CC\_SPEED\_SETTING\_}t_{31}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{20})\rangle, \langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23})\rangle]$

L1_3: $[\langle(\mathsf{CC\_LOGIC\_CONTROL\_}t_{22}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{18}, \mathsf{CC\_SPEED\_SETTING\_}t_{31}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{20})\rangle, \langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27}), (\mathsf{EVA\_}t_{26})\rangle]$

L1_4: $[\langle(\mathsf{CC\_LOGIC\_CONTROL\_}t_{22}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{18}, \mathsf{CC\_SPEED\_SETTING\_}t_{31}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{20})\rangle, \langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26})\rangle]$

7.4.2 Feature Interactions between CC and LG

The feature interactions detected between CC and LG for actuators Throttle and Steering were identified using SMV with the following property as invariant:

$$(\mathsf{sys\_stable} \rightarrow \neg(\mathtt{Throttle\_req}_{\mathrm{CC}} \wedge \mathtt{Steering\_req}_{\mathrm{LG}} \wedge ((\mathtt{set\_Throttle}_{\mathrm{CC}} > 40) \wedge (\mathtt{set\_Steering}_{\mathrm{LG}} > 0))).$$

The results of the analysis are summarized in Table 7.9, showing the number of iterations of the model checker, the equivalence classes discovered per level, the maximum BDD nodes for all iterations and the time taken to complete the process. The number of reachable states for the analysis of all levels with combo CC-LG is 1.93977e+12.

CC-LG	**Iterations**	**Equivalence Classes**	**BDD Nodes**	**Total Time**
Level 4	3	3	1191238	3m7s
Level 3	3	3	1194220	3m9s
Level 2	2	2	1189155	1m59s
Level 1	3	3	3645577	10m37s

Table 7.9: Conflicting actuators feature interaction analysis results for CC-LG

The following list shows the equivalence classes reported per level for the combo pair CC-LG. Discussions on traceability and manageability are left to Section 7.4.5. No partitions were needed to complete the analysis for Level 1, thus no discussion on scalability is included in this section.

Level 4 – Distinct Final States

L4_1: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING, CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sINC_SPEED LG_sLG=sENABLED,LG_sENABLED=sENGAGED,LG_sENGAGED=sASSIST_RIGHT)]

L4_2: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING, CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sHOLD_SPEED LG_sLG=sENABLED,LG_sENABLED=sENGAGED,LG_sENGAGED=sASSIST_RIGHT)]

L4_3: [(CC_CC=LOGIC_CONTROL,CC_ENABLED=ENGAGED,CC_ENGAGED=ACCELERATING, CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sDEC_SPEED LG_sLG=sENABLED,LG_sENABLED=sENGAGED,LG_sENGAGED=sASSIST_RIGHT)]

Level 3 – Distinct Initial and Final States
The three equivalence classes uncover the same information as Level 4 because there is only one initial state.

Level 2 – Distinct Last Transitions

L2_1: [CC_LOGIC_CONTROL_t_{20}, LG_t_{28}]

L2_2: [CC_LOGIC_CONTROL_t_{20}, LG_t_{25}]

Level 1 – Distinct Paths

L1_1: $[\langle(\mathsf{CC\_LOGIC\_CONTROL\_}t_{22}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{18}, \mathsf{CC\_SPEED\_SETTING\_}t_{31}),$
$(\mathsf{CC\_LOGIC\_CONTROL\_}t_{20})\rangle,$
$\langle(\mathsf{LG\_}t_{14}), (\mathsf{LG\_}t_{34}), (\mathsf{LG\_}t_{28})\rangle]$

L1_2: $[\langle(\mathsf{CC\_LOGIC\_CONTROL\_}t_{22}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{18}, \mathsf{CC\_SPEED\_SETTING\_}t_{31}),$
$(\mathsf{CC\_LOGIC\_CONTROL\_}t_{20})\rangle,$
$\langle(\mathsf{LG\_}t_{14}), (\mathsf{LG\_}t_{34}), (\mathsf{LG\_}t_{23}), (\mathsf{LG\_}t_{25})\rangle]$

L1_3: $[\langle(\mathsf{CC\_LOGIC\_CONTROL\_}t_{22}), (\mathsf{CC\_LOGIC\_CONTROL\_}t_{18}, \mathsf{CC\_SPEED\_SETTING\_}t_{31}),$
$(\mathsf{CC\_LOGIC\_CONTROL\_}t_{20})\rangle,$
$\langle(\mathsf{LG\_}t_{14}), (\mathsf{LG\_}t_{34}), (\mathsf{LG\_}t_{28}), (\mathsf{LG\_}t_{26}), (\mathsf{LG\_}t_{25})\rangle]$

7.4.3 Feature Interactions between CC and CA

The feature interactions detected between CC and EVA for actuators Throttle and Brake were identified using SMV with the following property as invariant:

$$(\mathsf{sys\_stable} \rightarrow \neg(\texttt{Throttle\_req}_{\text{CC}} \wedge \texttt{Brake\_req}_{\text{CA}} \wedge ((\texttt{set\_Throttle}_{\text{CC}} > 30) \wedge (\texttt{set\_Brake}_{\text{CA}} > 40))).$$

The results of the analysis are summarized in Table 7.10, showing the number of iterations of the model checker, the equivalence classes discovered per level (for Level 1 the results are incomplete, as indicated by the '+' sign), the maximum BDD nodes for all iterations and the time taken to complete the process. The number of reachable states for the analysis of all levels with combo CC-CA is 1.16548e+11.

CC-CA	Iterations	Equivalence Classes	BDD Nodes	Total Time
Level 4	3	3	2141806	5m18s
Level 3	3	3	2141336	4m59s
Level 2	4	4	4493705	9m0s
Level 1	77	15+	109091660	231h25m3s

Table 7.10: Conflicting actuators feature interaction analysis results for CC-CA

The following list shows the equivalence classes reported per level for the combo pair CC-CA. Discussions on traceability and manageability are left to Section 7.4.5, while the end of this section discusses about scalability because the combo CC-CA required partitioning to generate the results reported for Level 1 (although the results are incomplete).

Level 4 – Distinct Final States

L4_1: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING, CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sINC_SPEED, CA_sCA=sENABLED,CA_sENABLED=sENGAGED,CA_sENGAGED=sMITIGATE)]

L4_2: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING, CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sHOLD_SPEED, CA_sCA=sENABLED,CA_sENABLED=sENGAGED,CA_sENGAGED=sMITIGATE)]

L4_3: [(CC_sCC=sLOGIC_CONTROL,CC_sENABLED=sENGAGED,CC_sENGAGED=sACCELERATING, CC_sLOGIC_CONTROL=sENABLED,CC_sSPEED_SETTING=sDEC_SPEED, CA_sCA=sENABLED,CA_sENABLED=sENGAGED,CA_sENGAGED=sMITIGATE)]

Level 3 – Distinct Initial and Final States

The three equivalence classes uncover the same information as Level 4 because there is only one initial state.

Level 2 – Distinct Last Transitions

L2_1: [CC_LOGIC_CONTROL_t_{20}, CC_SPEED_SETTING_t_{33}, CA_t_{35}]

L2_2: [CC_LOGIC_CONTROL_t_{20}, CA_t_{35}]

L2_3: [CC_LOGIC_CONTROL_t_{20}, CA_t_{30}]

L2_4: [CC_LOGIC_CONTROL_t_{20}, CA_t_{29}]

Level 1 – Distinct Paths

L1_1: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}), (CC_LOGIC_CONTROL_t_{20}, CC_SPEED_SETTING_t_{33})⟩, ⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35})⟩]

L1_2: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}), (CC_LOGIC_CONTROL_t_{20})⟩, ⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35})⟩]

L1_3: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}), (CC_LOGIC_CONTROL_t_{20})⟩, ⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{33}), (CA_t_{30})⟩]

L1_4: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}), (CC_LOGIC_CONTROL_t_{20}, CC_SPEED_SETTING_t_{33})⟩, ⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{19}), (CA_t_{21}), (CA_t_{30})⟩]

L1_5: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}), (CC_LOGIC_CONTROL_t_{20})⟩, ⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{19}), (CA_t_{21}), (CA_t_{30})⟩]

L1_6: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35}), (CA_t_{32}), (CA_t_{30})⟩]

L1_7: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20}, CC_SPEED_SETTING_t_{33})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{19}), (CA_t_{29}), (CA_t_{32}), (CA_t_{30})⟩]

L1_8: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{19}), (CA_t_{29}), (CA_t_{32}), (CA_t_{30})⟩]

L1_9: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20}, CC_SPEED_SETTING_t_{33})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35}), (CA_t_{31}), (CA_t_{21}), (CA_t_{30})⟩]

L1_10: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35}), (CA_t_{31}), (CA_t_{21}), (CA_t_{30})⟩]

L1_11: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{19}), (CA_t_{29})⟩]

L1_12: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{33}), (CA_t_{22}), (CA_t_{29})⟩]

L1_13: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35}), (CA_t_{31}), (CA_t_{29})⟩]

L1_14: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{33}), (CA_t_{30}), (CA_t_{31}), (CA_t_{29})⟩]

L1_15: [⟨(CC_LOGIC_CONTROL_t_{22}), (CC_LOGIC_CONTROL_t_{18}, CC_SPEED_SETTING_t_{31}),
(CC_LOGIC_CONTROL_t_{20})⟩,
⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35}), (CA_t_{32}), (CA_t_{22}), (CA_t_{29})⟩]

Discussion of Scalability for CC-CA – Conflicting Actuators

Nine partitions were required for the analysis of Level 1 with combo CC-CA to finish, when detecting conflicting actuators feature interactions, as illustrated in Figure 7.7. As in the case for the detection of same actuator feature interactions, the starting time threshold for partitioning is 1 hour, doubling every time a new partition is selected, while the memory usage is set to unlimited, so SMV could use the 32GB of memory if needed. Still, in some cases, Cadence SMV ran out of memory in some branches while trying to complete the verification, indicated by SIGSEGV in Figure 7.7. Even if the results generated are not complete, *Alfie* tries to generate as much information as possible by recovering from these segmentation faults and continuing with the generation of equivalence classes in the next conditional branch, as illustrated in Figure 7.7. The partitioning process generated 15 distinct equivalence classes using 77 iterations and nine partitions.

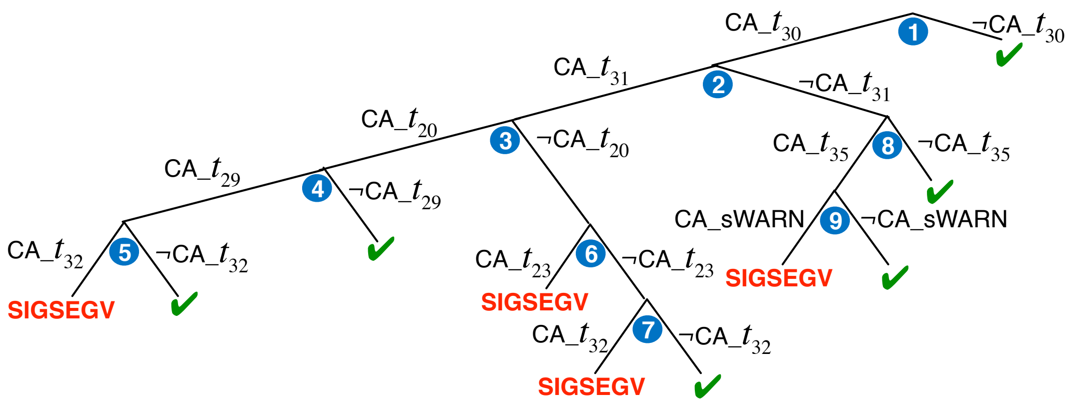


Figure 7.7: Partitions needed during analysis of Level 1 for CC-CA

7.4.4 Feature Interactions between CA and EVA

The feature interactions detected between CA and EVA for actuators Throttle and Brake were identified using SMV with the following property as invariant:

$$(\mathsf{sys\_stable} \rightarrow \neg(\texttt{Throttle\_req}_{\mathrm{EVA}} \wedge \texttt{Brake\_req}_{\mathrm{CA}} \wedge ((\texttt{set\_Throttle}_{\mathrm{EVA}} > 30) \wedge (\texttt{set\_Brake}_{\mathrm{CA}} > 40))).$$

The results of the analysis are summarized in Table 7.11, showing the number of iterations of the model checker, the equivalence classes discovered per level, the maximum BDD nodes for all iterations and the time taken to complete the process. The number of reachable states for the analysis of all levels with combo CA-EVA is 2.63757e+09.

The following list shows the equivalence classes reported per level for the combo pair CA-EVA. Discussions on traceability and manageability are left to Section 7.4.5, while the end of this section discusses about scalability because the combo CA-EVA required partitioning to generate the results reported for Level 1 (although the results are incomplete).

CA-EVA	**Iterations**	**Equivalence Classes**	**BDD Nodes**	**Total Time**
Level 4	1	1	21043	2.98s
Level 3	1	1	21102	2.98s
Level 2	6	6	45016	7.41s
Level 1	176	33+	127561652	280h59m50s

Table 7.11: Conflicting actuators feature interaction analysis results for CA-EVA

Level 4 – Distinct Final States

L4_1: [(CA_sCA=sENABLED,CA_sENABLED=sENGAGED,CA_sENGAGED=sMITIGATE,
EVA_sEVA=sENABLED,EVA_sENABLED=sENGAGED,EVA_sENGAGED=sCOAST)]

Level 3 – Distinct Initial and Final States
The equivalence class uncovers the same information as Level 4 because there is only one initial state.

Level 2 – Distinct Last Transitions

L2_1: [CA_t_{35}, EVA_t_{24}]

L2_2: [CA_t_{35}, EVA_t_{27}]

L2_3: [CA_t_{30}, EVA_t_{24}]

L2_4: [CA_t_{30}, EVA_t_{27}]

L2_5: [CA_t_{29}, EVA_t_{24}]

L2_6: [CA_t_{29}, EVA_t_{27}]

Level 1 – Distinct Paths

L1_1: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24})⟩]

L1_2: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{23}), (EVA_t_{27})⟩]

L1_3: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24}), (EVA_t_{26}), (EVA_t_{27})⟩]

L1_4: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{33}), (CA_t_{30})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24})⟩]

L1_5: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{19}), (CA_t_{21}), (CA_t_{30})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24})⟩]

L1_6: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35}), (CA_t_{32}), (CA_t_{30})⟩, ⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24})⟩]

L1_7: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{19}), (CA_t_{29}), (CA_t_{32}), (CA_t_{30})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24})⟩]

L1_8: [⟨(CA_t_{14}), (CA_t_{16}), (CA_t_{35}), (CA_t_{31}), (CA_t_{21}), (CA_t_{30})⟩,
⟨(EVA_t_{13}), (EVA_t_{20}), (EVA_t_{24})⟩]

L1_9: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_10: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_11: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{21}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_12: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{21}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_13: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{32}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_14: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{32}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_15: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{29}), (\mathsf{CA\_}t_{32}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_16: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{29}), (\mathsf{CA\_}t_{32}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_17: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{21}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_18: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{21}), (\mathsf{CA\_}t_{30})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_19: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{29})\rangle, \langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24})\rangle]$

L1_20: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle, \langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24})\rangle]$

L1_21: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle, \langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24})\rangle]$

L1_22: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{30}), (\mathsf{CA\_}t_{31}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24})\rangle]$

L1_23: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{35}), (\mathsf{CA\_}t_{32}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24})\rangle]$

L1_24: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_25: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{19}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_26: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{23}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_27: $[\langle(\mathsf{CA\_}t_{14}), (\mathsf{CA\_}t_{16}), (\mathsf{CA\_}t_{33}), (\mathsf{CA\_}t_{22}), (\mathsf{CA\_}t_{29})\rangle,$
$\langle(\mathsf{EVA\_}t_{13}), (\mathsf{EVA\_}t_{20}), (\mathsf{EVA\_}t_{24}), (\mathsf{EVA\_}t_{26}), (\mathsf{EVA\_}t_{27})\rangle]$

L1_28: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{31}), (\mathsf{CA}\_t_{29})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{27})\rangle]$

L1_29: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{31}), (\mathsf{CA}\_t_{29})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{26}), (\mathsf{EVA}\_t_{27})\rangle]$

L1_30: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{33}), (\mathsf{CA}\_t_{30}), (\mathsf{CA}\_t_{31}), (\mathsf{CA}\_t_{29})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{27})\rangle]$

L1_31: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{33}), (\mathsf{CA}\_t_{30}), (\mathsf{CA}\_t_{31}), (\mathsf{CA}\_t_{29})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{26}), (\mathsf{EVA}\_t_{27})\rangle]$

L1_32: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{32}), (\mathsf{CA}\_t_{22}), (\mathsf{CA}\_t_{29})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{23}), (\mathsf{EVA}\_t_{27})\rangle]$

L1_33: $[\langle(\mathsf{CA}\_t_{14}), (\mathsf{CA}\_t_{16}), (\mathsf{CA}\_t_{35}), (\mathsf{CA}\_t_{32}), (\mathsf{CA}\_t_{22}), (\mathsf{CA}\_t_{29})\rangle,$
$\langle(\mathsf{EVA}\_t_{13}), (\mathsf{EVA}\_t_{20}), (\mathsf{EVA}\_t_{24}), (\mathsf{EVA}\_t_{26}), (\mathsf{EVA}\_t_{27})\rangle]$

Discussion of Scalability for CA-EVA – Conflicting Actuators

Ten partitions were required for the analysis of Level 1 for CA-EVA to finish, when detecting conflicting actuators feature interactions, as illustrated in Figure 7.8. The starting time threshold for partitioning is 1 hour, doubling when a new partition is selected, and always using as much as 32 GB of memory during verification. For CA-EVA, there were also some branches in which Cadence SMV ran out of memory during verification, marked in Figure 7.8 with SIGSEGV. Although the results generated are not complete, *Alfie* generated as many equivalence classes as possible (*i.e.,* 33 distinct equivalence classes). As seen in Table 7.11, more iterations than equivalence classes are reported because of repeated information generated during partitioning.

7.4.5 Discussion of Traceability and Manageability for Conflicting Actuators Feature Interactions

Traceability – For the conflicting actuators combo pair CA-EVA, which does not include ordered-compositions, the results can be easily traced back to the Stateflow models by (1) following the hierarchy of control states, as the name of the states reported in the counterexample by SMV are unchanged from the Stateflow models (in the results from Level 3 and 4), (2) locating the transition name reported in the counterexample by SMV, as these models use only one transition variable for the last transition taken (in the results from Level 2), and (3) following the sequence of transition names reported in the counterexample by SMV from the only initial state (in the results from Level 1).

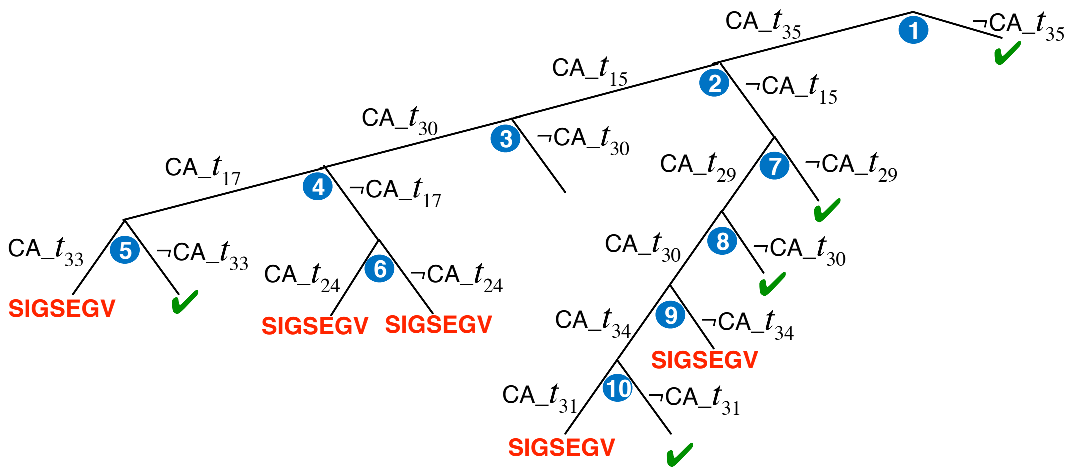


Figure 7.8: Partitions needed during analysis of Level 1 for CA-EVA

For the conflicting actuator combo pairs CC-CA, CC-EVA, and CC-LG, in which feature CC includes an ordered-composition, the results can be traced back to the STATEFLOW models by (1) following all the hierarchies of control states, as the name of the states reported in the counterexample by SMV are unchanged from the STATEFLOW models (in the results from Level 3 and 4), (2) following the hierarchies of transition names reported in the counterexample, as model CC includes an ordered-composition, and therefore, it requires several transition variables to report the transitions taken in the big-step (in the results from Level 2), and (3) following the sequence of transition names, hierarchically in the case of CC, reported in the counterexample from the only initial state (in the results from Level 1). Following the transition names hierarchically is not too complicated, as the name of the transition variable indicates the superstate in which the transition is located.

Manageability – The results are manageable as they can be listed in each section for all levels and for all combos, whereas an approach that shows all data variants of the equivalence classes in each section would be hard to generate and analyze.

7.5 Related Work

This section discusses approaches that partition a verification run by modifying the property. Pu and Zhang define search state partitioning for LTL model checking by introducing new data variables in the model, and use these variables as conditions to partition the search [150, 151]. Their partitions are also disjoint conditions as in my method. However, my method does not introduce variables that are not part of the original model. Sebastiani *et al.* [158] propose the use of generalized symbolic trajectory evaluation (GSTE) [188] as property-driven partitioning. It uses the state variables in the property automaton to drive the partitioning, creating an abstraction of the system. In contrast, my partitioning works on the concrete model of the system.

7.6 Summary

This chapter has shown the use of my method and tool *Alfie*, described in Chapter 6, and also demonstrated the reduction produced by my equivalence classes of counterexamples in the detection of feature interactions for combinations of seven automotive design models. My case studies also showed the advantages of using partitioning to generate more equivalence classes of feature interactions.

Chapter 8

Conclusions

This chapter summarizes the main contributions of my work, as well as its limitations. Finally, I discuss my plans for future work.

8.1 Contributions

My contributions presented in this dissertation are the following:

- The identification of the characteristics of automotive active safety systems that make model checking a promising technique to detect feature interactions.
- A systematic, complete and general definition of feature interactions that identifies contradictory requests by software features to actuators. This definition is based on the set of actuators controlled by the features and domain expert knowledge.
- The creation of the UWFSM set of non-proprietary automotive active safety feature models (without vehicle dynamics), designed in MATLAB's STATEFLOW, to use in my case study because there is not a publicly available set of models.
- The creation of the translator *mdl2smv* that generates SMV models from automotive features designed using a subset of the MATLAB's STATEFLOW language. The translated SMV models contain the same level of description as the design models.
- A novel method and tool, called *Alfie*, to detect a set of counterexamples that is representative of the set of all counterexamples to an invariant for an extended finite state machine (EFSM) model by modifying on-the-fly the property being verified.

Alfie divides the set of all counterexamples into equivalence classes based on similarity in states and transitions in the EFSM path, producing one representative counterexample per equivalence class.

- The process to detect a feature interaction when two Stateflow models are running concurrently, using the model checker SMV, generating a counterexample when a feature interaction is identified. The behaviour of the vehicle dynamics is left completely unrestricted to identify any conflicting requests to actuators.
- The generalization of my method and tool *Alfie* to pairs of concurrent Stateflow models running in parallel to detect to detect a set of counterexamples that is representative of the set of all counterexamples (feature interactions), generating a representative counterexample per equivalence classes.
- The use of a partitioning strategy by *Alfie* to provide a scalable solution to the problem of finding all feature interactions. This strategy breaks down the LTL property that represents the equivalence classes of counterexamples seen so far in the process into subproblems, which cover the original LTL property when the verification becomes too large to model check.

The validation of my contributions is summarized as follows:

- My definition of feature interactions has the following attributes, which are validated in Section 3.3:
  - *systematic*, by creating LTL properties automatically from a list of actuators and their value thresholds.
  - *complete* with respect to the set of actuators and thresholds provided by domain experts to detect any conflicting actuator requests between pairs of features.
  - *general*, by being independent of the behaviours of the features that are part of the system.
- My translator *mdl2smv* generates SMV models from Stateflow active safety design models that retain the same level of detail, validated by checking the traceability of my case study results in Chapter 7.
- My novel method and tool *Alfie* generates a set of counterexamples that is representative of the set of feature interactions between pairs of active safety features, with a counterexample produced per equivalence class, making my results *manageable*. This process is validated by illustrating the reduction achieved by my method in Section 5.5 and by observation of the results of my case study in Chapter 7.

- *Alfie* is made *scalable* by using a strategy that partition the problem when the size of the LTL property is too large to model check, validated experimentally in Chapter 7 by analyzing the partitioning used in my case study.

8.2 Limitations

There are some factors that might limit the adoption of my method. This section describes the limitations of my work.

My definition of feature interactions for active safety features is advantageous with respect to the traditional approach to detect interactions by automotive domain experts, where domain experts try to list the behaviours in which they expect the features to interact in an unsafe manner. However, my definition is only complete with respect to the expert knowledge provided, *i.e.,* the thresholds provided by domain experts. An outstanding problem is how to determine value thresholds that do not miss feature interactions that can lead to safety risks. Also, my definition does not consider vehicle dynamics, thus making analysis using formal verification practical. More research is needed to incorporate vehicle dynamics.

My translator from MATLAB's STATEFLOW to SMV, *mdl2smv*, does not recognize nor handle the following STATEFLOW syntax:

- Condition actions in transitions
- Actions within states
- Connective junctions
- Graphical and MATLAB functions
- In(state_name) condition functions
- Temporal conditions (use of operators such as `after`, `at`, `every` within STATEFLOW)
- Event broadcasting

However, I have observed that these syntactic elements were not needed in industrial practice when modelling automotive design features, and that an equivalent design can likely be created without these syntactic elements.

My method and tool *Alfie* detect feature interactions for automotive active safety features, which are intra-vehicle features. For inter-vehicle features, the number and kind of features are not fixed, and therefore, model checking may not be a good approach for detection. Also, for Level 1, *Alfie* could miss some distinct bugs because of the reduction of EFSM loops, for instance, in the case of multiple self-looping transitions that all generate a feature interaction. This is a design decision I made, which makes the verification more efficient. Even with the reduction of EFSM loops, some combination of features still generated a large number of equivalence classes for Level 1 during my case study, such as combo CA-EVA. However, including even one instance of each different EFSM loop would make the problem of generating all equivalence classes of counterexamples for Level 1 even harder, based on my observation while experimenting with this idea.

In my case studies, some combination of features generated a very large number of equivalence classes, although they were not able to complete even while using partitioning. This issue might be aided by the incorporation of environmental constraints. However, some of these constraints appear to be specific to a particular feature or combination of features, which could make my method dependent on the set of features that are part of the system.

8.3 Future Work

For my definition of feature interactions, instead of simply considering a fixed value as a threshold, I plan to consider different value thresholds depending on the environmental conditions, *e.g.,* using a greater value threshold to detect interactions when cruising at low speeds. This consideration may help the analysis be more efficient, but I do not expect any other changes in the feature interaction definition or in the definition of equivalence classes of counterexamples.

I am also interested in generalizing my methods and tools to feature models that are designed in languages other than Stateflow. For this generalization, the first step would be the creation of a translator from the newly considered design language to the language in which analysis is going to be performed. Cadence SMV has been a good fit for our current analysis, and although we briefly explored options such as bounded model checking [57] without improvement to our iterative method, it might be worth considering other model checkers such as NuSMV [54] or SPIN [93]. Depending on the source and target language, in the future it might possible to integrate the ability to simulate counterexamples generated.

The features considered in this dissertation are intra-vehicle features, which can be thought of as a fixed set in the car because they are selected at release or resale time. With the new trend of inter-vehicle features, which can be integrated or removed dynamically, *e.g.,* infotainment services or road-assistance services, it would be interesting to explore the detection of feature interactions for inter-vehicle features and what the impact in methods and tools would be. Some of the approaches proposed for detection of feature interactions for Internet applications seem relevant when considering inter-vehicle features.

Another option to explore is the generalization of my definition of feature interactions, as well as to my method *Alfie*, to recognize multiple configurations that fail the invariant in a path, although applying the method to a real system did not seem computational feasible when I performed some trials with such generalization.

My method is validated by a case study with my UWFMS set of non-proprietary automotive feature design models to detect all feature interactions of type *Immediate*. I plan to expand on my results and techniques by tackling the detection of feature interactions of type *Temporal*. The challenges that I expect to overcome are: (a) the generalization of the LTL representation of the equivalence classes when dealing with requests separated in time, using history variables or a more expressive property representation, and (b) the description of strategies that partition the equivalence class into smaller categories in the case that the size of the property is too big.

Also, because I believe that my results will have relevance to the solution of the feature interaction problem in other cyber physical systems, I intend to study the applicability of my techniques to other domains, *e.g.,* in networked medical systems.

APPENDICES

Appendix A

Non-Proprietary Automotive Feature Set: UWFMS

This chapter describes the functionality of each of the automotive feature design models in my non-proprietary set and show how such functionality is modelled using the subset of the STATEFLOW language described in Chapter 4. The set of non-proprietary feature design models is called the "University of Waterloo Feature Model Set" (**UWFMS**). The UWFMS is novel in the sense that there is not a publicly available set of models that I could use to validate my methods to detect feature interactions in the automotive domain. Some of these feature design models are based on TRW Automotive features' textual descriptions provided by their website [5]. These features are regarded by TRW as 'Active Safety Systems', under the heading of 'Driver Assist Systems'. The other features were devised by Richard Fanson[1] and I to have a larger set of feature models to work with.

Each section provides the final STATEFLOW design model per feature used in the case study described in Chapter 7, as well as a brief description of the design decisions made when modelling the UWFMS in STATEFLOW. Each of the non-proprietary feature models were designed to fulfill a goal, with no explicit intention that these features might interact with each other in an unsafe manner. The inputs, outputs and local variables used in each design model are also summarized in a table for each type of variables. These tables are placed close to the design model to help the reader understand the figures.

A.1 Cruise Control (CC)

UWFMS's CC is based on TRW's ACC description:

[1]Richard Fanson is a Mechatronics engineer who helped in the design of the UWFMS. His knowledge and insights, as well as those from the engineers at GM, helped to make the UWFMS representative [99].

"TRW's Adaptive Cruise Control (ACC) technology improves upon standard cruise control by automatically adjusting the vehicle speed and distance to that of a target vehicle. ACC uses a long range radar sensor to detect a target vehicle up to 200 meters in front and automatically adjusts the ACC vehicle speed and gap accordingly. ACC decelerates or accelerates the vehicle according to the desired speed and distance settings established by the driver. As per standard cruise control, the driver can override the system at any time".

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

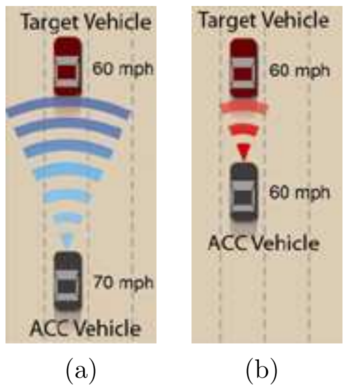


Figure A.1: Adaptive Cruise Control Functionality: (a) The ACC vehicle approaches the Target vehicle at 70 mph (ACC's vehicle set speed); (b) Due to the proximity, ACC starts coasting by adjusting the ACC vehicle's speed to 60 mph, matching the Target vehicle's speed.

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

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

Figure A.2 presents CC's functionality modelled in Stateflow, which will be explained in the following paragraphs. Table A.1 shows the local variables, Table A.2 the input variables, and Table A.3 the output variables used in UWFMS's CC design model.

The details of UWFMS's CC design in Stateflow are as follows: CC should automatically accelerate and decelerate the CC vehicle based on a target speed specified by the driver and a constant distance separation with a Target vehicle, set at 50 meters. Unlike TRW's CC description, our design uses KPH (kilometers per hour). CC consists of two siblings in an ordered-composition: the LOGIC_CONTROL state, which controls the vehicle logic for the throttle, and the SPEED_SETTING state, which keeps track of the target speed. The default of the LOGIC_CONTROL sibling is the DISABLED state, and CC starts in the DISABLED state when the car is turned on. When the driver turns CC on, by pressing the CC_Enabled button, CC enters the ENABLED state and sets the target speed to 0 (which remains 0 until the feature is engaged).

The default of the ENABLED superstate is the DISENGAGED state. Pressing the Set/Accel button, while driving forward at a speed greater than 40 KPH, makes CC to enter the ENGAGED state. The default of the ENGAGED superstate is the COASTING state. CC must be in the COASTING state when either the current speed exceeds the target speed, or if the vehicle ahead is less than 50 meters away. Coast means that the CC vehicle either maintains or decreases its speed by requesting no throttle, but not by braking. CC moves to the ACCELERATING state when the current speed is less then the target speed and the Target vehicle is farther than 50 meters from the CC vehicle. The throttle output is proportional to the difference between the target speed and the current speed. Depressing the brake or pressing the Cancel button shall cause CC to enter the OVERRIDE state. This state remains active until either the Set/Accel or Resume/Coast buttons is pressed, which makes CC to enter the ENGAGED state. An Error event at any time when CC is on will cause a transition to the FAIL state, and it will not be able to recover from this condition until the CC vehicle is restarted.

The default of the SPEED_SETTING sibling is the HOLD_SPEED state. Pressing the Set/Accel button makes CC to enter the INC_SPEED state and locks in the current speed value as the target speed. If the Set/Accel button is held while the target speed is less than 100 KPH, the target speed is incremented for every cycle unit that occurs. Release of the Set/Accel button sends CC back to the HOLD_SPEED state. Pressing the Resume/Coast button makes CC to enter the DEC_SPEED state and locks in the current speed value as the target speed. If the Resume/Coast button is held while the target speed is greater than 0 KPH, the target speed is decremented for every cycle unit that occurs. Release of the Resume/Coast button sends CC back to the HOLD_SPEED state.

The definition of stable for UWFMS's CC is (sCC=sLOGIC_CONTROL).

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

Table A.1: Local variables used in Cruise Control (CC)

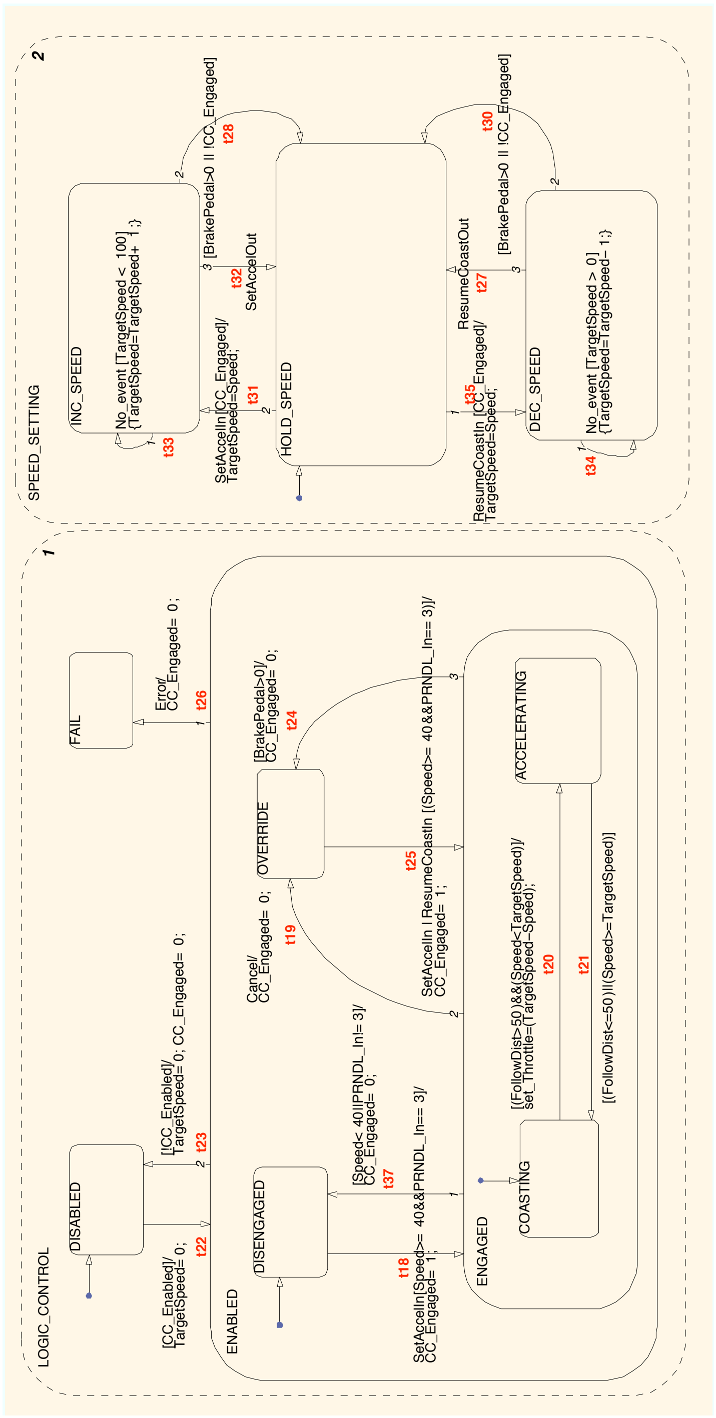
Type	**Name**	**Meaning**
event	No_event	Signal at a cycle unit for the STATEFLOW model when no other event is generated
event	SetAccelIn	Signal indicating depression of Set/Accel button
event	SetAccelOut	Signal indicating release of Set/Accel button
event	ResumeCoastIn	Signal indicating depression of Resume/Coast button
event	ResumeCoastOut	Signal indicating release of Resume/Coast button
event	Cancel	Signal indicating depression of Cancel button
event	Error	Signal indicating when an error has occurred
data	CC_Enabled	Driver controlled main power to enable/disable the feature [Boolean]
data	FollowDist	Number to define distance to the vehicle ahead (Value between 0 and 100) [Meters in integers]
data	BrakePedal	Value of brake pedal input represented as a percentage of maximum depression [Percentage in integers]
data	AccelPedal	Value of accelerator pedal input represented as a percentage of maximum depression [Percentage in integers]
data	Speed	Current speed of the vehicle (value within the range of 0 to 100) [KPH in integers]

data	PRNDL_In	The input representing the current gear selection with the following values assumed:
		PRNDL = 0: Park
		PRNDL = 1: Reverse
		PRNDL = 2: Neutral
		PRNDL = 3: Drive
		PRNDL = 4: Low
		[Gear selection in integers]

Table A.2: Input variables used in Cruise Control (CC)

Type	**Name**	**Meaning**
data	set_Throttle	Value proportional to difference between the target speed and current speed [Percentage in integers]
data	TargetSpeed	Target cruising speed of the feature (This variable is an output to display its value) [KPH in integers]

Table A.3: Output variables used in Cruise Control (CC)



SPEED_SETTING
2
INC_SPEED
No_event [TargetSpeed < 100]
{TargetSpeed=TargetSpeed+ 1;}
t33
[BrakePedal>0 || !CC_Engaged]
t28
SetAccelOut
t32
SetAccelIn [CC_Engaged]/
TargetSpeed=Speed;
t31
HOLD_SPEED
ResumeCoastOut
t27
[BrakePedal>0 || !CC_Engaged]
t30
ResumeCoastIn [CC_Engaged]/
TargetSpeed=Speed;
t35
DEC_SPEED
No_event [TargetSpeed > 0]
{TargetSpeed=TargetSpeed- 1;}
t34
LOGIC_CONTROL
1
DISABLED
FAIL
Error/
CC_Engaged= 0;
t26
[!CC_Enabled]/
TargetSpeed= 0; CC_Engaged= 0;
t23
[CC_Enabled]/
TargetSpeed= 0;
t22
ENABLED
DISENGAGED
OVERRIDE
[BrakePedal>0]/
CC_Engaged= 0;
t24
Cancel/
CC_Engaged= 0;
t19
SetAccelIn | ResumeCoastIn [(Speed>= 40&&PRNDL_In== 3)]/
CC_Engaged= 1;
t25
[Speed< 40||PRNDL_In!= 3]/
CC_Engaged= 0;
t37
SetAccelIn[Speed>= 40&&PRNDL_In== 3]/
CC_Engaged= 1;
t18
ENGAGED
ACCELERATING
COASTING
[(FollowDist>50)&&(Speed<TargetSpeed)]/
set_Throttle=(TargetSpeed-Speed);
t20
[(FollowDist<=50)||(Speed>=TargetSpeed)]
t21

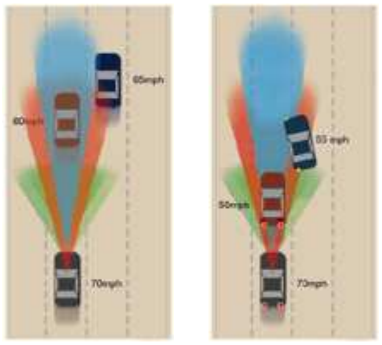
Figure A.2: Cruise Control (CC) Stateflow design model

A.2 Collision Avoidance (CA)

UWFMS's CA is based on TRW's CW description:

> "TRW's Collision Warning (CW) System can assist drivers by helping to prevent or mitigate accidents. Combining long and short range radars with a video camera, TRW's Collision Warning monitors the road ahead (including part of the side-fronts). In the event that a vehicle or obstacle approaches, TRW's collision warning system can notify the driver of a possible collision through audible or visual alerts and can also provide braking force as soon as an imminent collision is detected. "

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



(a) (b)

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

Figure A.4 presents CA's functionality modelled in Stateflow, which we will explain in the following paragraphs. UWFMS's CA adds a level of collision threat to the two proposed by TRW's CW. Table A.4 describes the input variables and Table A.5 describes the output variables used in UWFMS's CA design model.

The details of UWFMS's CA design in Stateflow are as follows: The CA feature begins in the DISABLED state. When the driver turns CA on by pressing the CA_Enabled button, CA enters the ENABLED state. The default of the ENABLED superstate is the DISENGAGED state. Once the CA vehicle's speed exceeds 25 KPH while driving forward, the feature will enter the ENGAGED state.

When engaged, CA is able to warn the driver and/or take action if a possible threat is encountered. I assumed that some external pre-processing unit computes data from the sensors and converts it into a threat input for CA, which is a reasonable assumption as it occurs in practice in the automotive industry [16]. The threat input could be either: None=0, Mild=1, Near=2, Imminent=3. If a threat input is received while in either the IDLE, WARN, AVOID, or MITIGATE states, one of the following transitions occurs depending on the threat:

- In the case that no threat is detected, or the obstacle ceases to be present, no warning and no braking intervention will be made. CA changes to the IDLE state.
- In the case of a mild threat, a mild threat warning will be output (*i.e.,* Warning = 1) with no braking intervention. CA enters the WARN state.
- In the case of a near threat, a near threat warning will be output (*i.e.,* Warning = 2) and soft braking will occur to slow the vehicle and avoid the potential collision (*i.e.,* Brake = 30%). CA enters the AVOID state.
- In the case of an imminent threat, an imminent threat warning will be output (*i.e.,* Warning = 3) and full force braking will occur to mitigate the collision as much as possible (*i.e.,* Brake = 80%). CA enters the MITIGATE state.

If the vehicle is brought to a halt when in either the AVOID or MITIGATE states, CA will go to the HALT state and the vehicle will be held at halt until the driver presses on the brake pedal beyond a threshold (for our model, the brake pedal threshold is set to 10 percent of depression) to resume control of the vehicle. Then, CA will enter the ENABLED but DISENGAGED state. A driver acceleration pedal input exceeding 35 percent of depression will cause CA to enter the OVERRIDE state from any state in the ENABLED superstate. Releasing the acceleration pedal to less than 35 percent of depression while CA_Enabled is still true sends CA back to the ENABLED state. An Error event at any time when CA is on will cause a transition to the FAIL state, and it will not be able to recover from this condition until the vehicle is restarted.

UWFMS's CA relies on the fact that the pre-processing threat assessment accounts for vehicle speeds so that what once was a threat will not remain a threat at speeds lower than 25 KPH or at a stop. Otherwise, the feature could be stuck at halt or other undesirable outcomes.

The definition of stable for UWFMS's CA is defined as 1, *i.e., true* because the only one transition is taken in each big-step as ordered-compositions are not present in CA.

Type	Name	Meaning
event	Error	A signal indicating when an error has occurred
data	CA_Enabled	Driver controlled main power to enable/disable the feature [Boolean]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	AccelPedal	Value of physical pedal input represented as a percentage of maximum depression [Percentage in integers]
data	Speed	Current speed of the vehicle (value within the range of 0 to 100) [KPH in integers]

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

data	PRNDL_In	The input representing the current gear selection with the following values assumed:
		PRNDL = 0: Park
		PRNDL = 1: Reverse
		PRNDL = 2: Neutral
		PRNDL = 3: Drive
		PRNDL = 4: Low
		[Gear selection in integers]

Table A.4: Input variables used in Collision Avoidance (CA)

Type	Name	Meaning

data	set_Brake	Output indicating if the feature is physically intervening by applying brake force:
		set_Brake = 30: Soft-Braking
		set_Brake = 80: Hard-Braking
		[Brake degree as percentage in integers]

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

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

Table A.5: Output variables used in Collision Avoidance (CA)

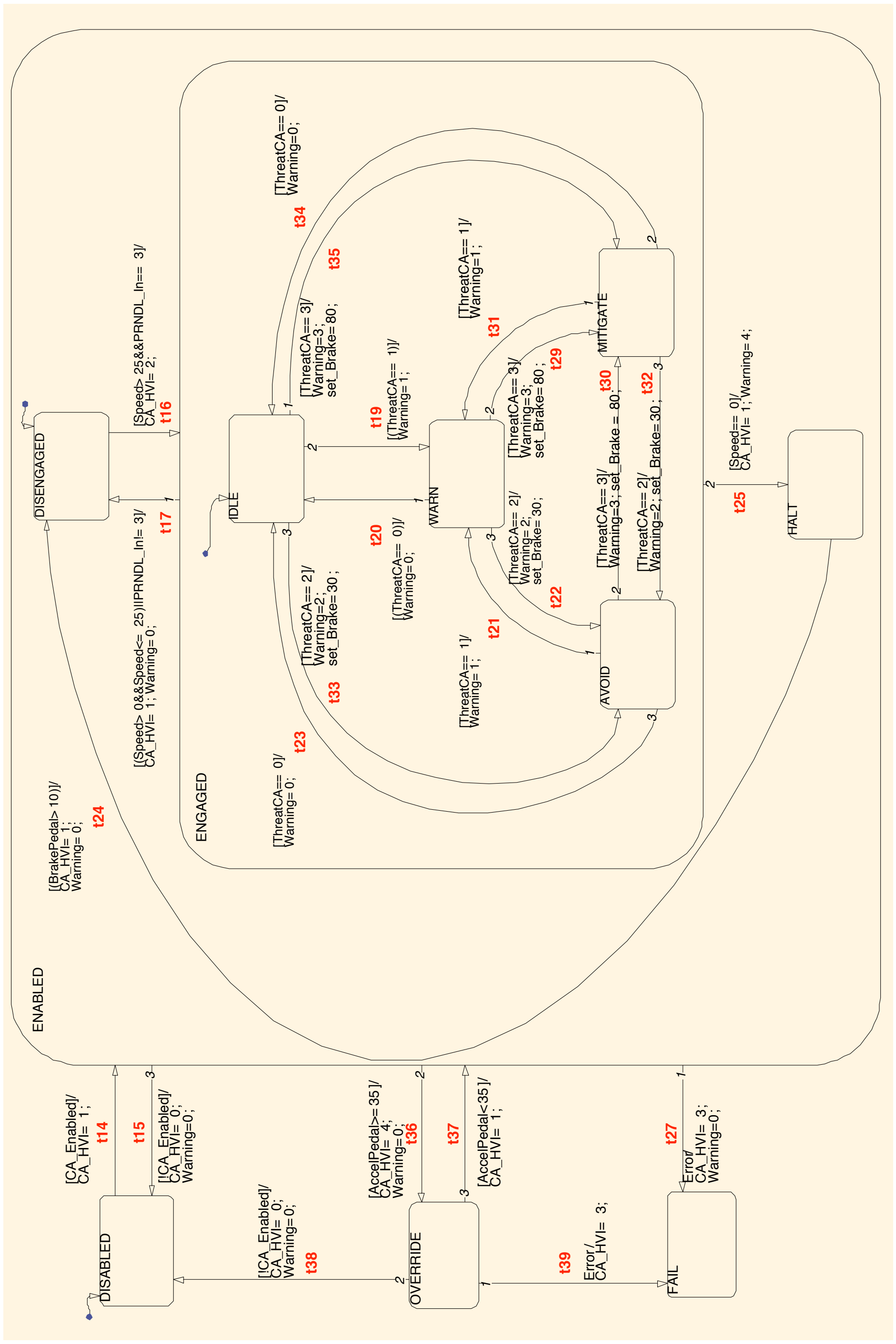

ENABLED
ENGAGED
DISENGAGED
IDLE
WARN
MITIGATE
AVOID
HALT
DISABLED
OVERRIDE
FAIL
[Speed> 25&&PRNDL_In== 3]/ CA_HVI= 2;
t16
[(Speed> 0&&Speed<= 25)||PRNDL_In!= 3]/ CA_HVI= 1; Warning= 0;
t17
[(BrakePedal> 10)]/ CA_HVI= 1; Warning= 0;
t24
[ThreatCA== 0]/ Warning=0;
t34
[ThreatCA== 3]/ Warning=3; set_Brake=80;
t35
[(ThreatCA== 1)]/ Warning=1;
t19
[(ThreatCA== 0)]/ Warning=0;
t20
[ThreatCA== 2]/ Warning=2; set_Brake=30;
t33
[ThreatCA== 0]/ Warning=0;
t23
[ThreatCA== 1]/ Warning=1;
t31
[ThreatCA== 3]/ Warning=3; set_Brake=80;
t29
[ThreatCA== 2]/ Warning=2; set_Brake=30;
t22
[ThreatCA== 1]/ Warning=1;
t21
[ThreatCA== 3]/ Warning=3; set_Brake = 80;
t30
[ThreatCA== 2]/ Warning=2; set_Brake=30;
t32
[Speed== 0]/ CA_HVI= 1; Warning= 4;
t25
[CA_Enabled]/ CA_HVI= 1;
t14
[!CA_Enabled]/ CA_HVI= 0; Warning=0;
t15
[AccelPedal>=35]/ CA_HVI= 4; Warning=0;
t36
[AccelPedal<35]/ CA_HVI= 1;
t37
Error/ CA_HVI= 3; Warning=0;
t27
[!CA_Enabled]/ CA_HVI= 0; Warning=0;
t38
Error/ CA_HVI= 3;
t39

Figure A.4: Collision Avoidance (CA) Stateflow design model

A.3 Park Assist (PA)

UWFMS's PA is based on TRW's PA description:

> "The TRW Park Assist (PA) system combines electrically powered steering with environmental sensing to aid drivers during parallel parking maneuvers. The system uses short range radar sensors to evaluate the length of the parking slot. From this information, the steering trajectory is calculated and the proper steering angle is automatically chosen. The driver monitors the steering. "

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

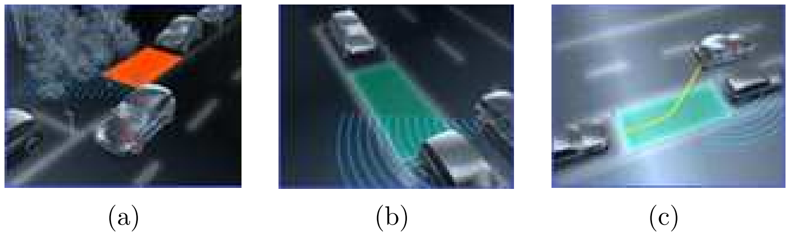


Figure A.5: Park Assist Functionality: (a) PA monitors for an empty space where the PA vehicle can fit; (b) If a large enough space is found and the driver accepts the space, PA starts the parking maneuver; (c) PA automatically adjusts the steering, throttle and braking during the parking maneuver.

Figure A.6 presents PA's functionality modelled in Stateflow, which we will explain in the following paragraphs. Unlike TRW's PA, in our design, we included the ability for PA to operate the throttle and braking system while the parking maneuver is completed. PA communicates with the driver when some decisions need to be made, such as accepting parking spot, changing gears or enabling the next parking action (*e.g.,* indicate NextPA). Table A.6 describes the input variables and Table A.7 describes the output variables used in the PA's Stateflow design model.

The details of UWFMS's PA design in Stateflow are as follows: PA starts in the DISABLED state when the car is turned on. When the driver turns PA on by pressing the PA_Enabled button, PA enters the ENABLED state. The default of the ENABLED superstate is the IDLE state. The enabled feature PA will remain idle until the speed of the vehicle is less than 10 KPH while driving forward to move to the SEARCHING state and start monitoring the sizes of the adjacent parking spaces. As observed in practice, We assumed that the sensor data is pre-processed by an external module and that PA simply obtains a SpaceFound boolean input indicating when a large enough space has been found. When PA receives the SpaceFound boolean with value true, it will enter the PROMPTING state to prompt the driver (through the HVI) to stop the car and accept the space, or decline the space. If the space is declined or if the driver fails to stop, the external module will make the SpaceFound boolean false, and PA will return to the SEARCHING state to monitor for another parking space. If the driver stopped the car and the space is accepted, PA enters the ENGAGED superstate, entering directly the SWIVEL_OUT state.

While engaged, PA will take over and perform the parallel park maneuver. PA relies on an external component that monitories the progress of the maneuver, sending back to PA a Next event through a HVI when a transition to the next step of the parking maneuver is initiated. PA also relies on the driver to change gears when required, as feature engineers indicated that this process could not be performed automatically by a feature[2]. The maneuver is broken up into 5 steps, each of which correspond to a state:

1. At state SWIVEL_OUT, when receiving a Next event along with variables PRNDL set to 1 (*i.e.,* Reverse) and speed set to a value in the range 1..5, PA will first reverse and turn into the parking space. PA will enter the SWIVEL_IN state.
2. At state SWIVEL_IN, when receiving a Next event along with PRNDL set to 1 (*i.e.,* Reverse) and speed set to a value in the range 1..5, PA will continue to reverse but turn the wheels the other way to swivel the front end into the parking space. PA will move to the STOP1 state.
3. At state STOP1, when receiving a Next event and speed is 0, PA will stop and straighten the wheels. PA will go to the CENTER state.
4. At state CENTER, when receiving a Next event along with PRNDL set to 3 (*i.e.,* Drive) and speed set to a value in the range 1..5, PA will pull forward into the middle of the parking space. PA will enter the STOP2 state.
5. At state STOP2, when speed is 0, PA will finally stop since the maneuver is complete. PA will move to the DISABLED state.

Braking by the driver or the detection of a threat during the maneuver will send PA into the OVERRIDE state, which can be resumed, where the parking process left off, when the threat or braking ceases. Some pre-processing of sensor inputs will compute the sensor data and convert it into a threat input for PA. Steering or acceleration by the driver during the maneuver will send PA to the ABORT state. PA is designed so the feature cannot resume from ABORT since the vehicle is likely off its trajectory path and cannot complete the parking maneuver. An Error event at any time when PA is on will cause a transition to the FAIL mode that cannot be left until the vehicle is shut off and restarted.

The definition of stable for UWFMS's PA is defined as 1, *i.e., true* because the only one transition is taken in each big-step as ordered-compositions are not present in PA.

[2]Clarification made by feature design engineers during Alma Juarez's visits to GM Research and Development (2007-2008).

Type	**Name**	**Meaning**
event	Next	An signal from a diagnostic external component to indicate when to transition between the different phases of the parking maneuver
event	Error	A signal indicating when an error has occurred
data	PA_Enabled	Driver selected main power to enable/disable the feature [Boolean]
data	SpaceFound	Value to indicate whether or not the sensors have found an appropriately sized space [Boolean]
data	Accepted	Driver's input through a HVI to indicate (when prompted) that the driver accepts the space found, so PA can begin the maneuver [Boolean]
data	Declined	Driver's input through a HVI to indicate (when prompted) that the driver does not accept the space found, so PA looks for another space [Boolean]
data	SteerIn	Driver controlled physical steering wheel input as a steering wheel angle (0 means centred and any other value means steering input from the driver) [Angle in integers]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	AccelPedal	Value of physical pedal input represented as a percentage of maximum depression [Percentage in integers]
data	Speed	Current speed of the vehicle (value within the range of 0 to 100) [KPH in integers]
data	Threat	Input from a pre-processing threat assessment block that converts sensor inputs into a threat; For PA, it only indicates presence of obstruction [Boolean]

data	PRNDL_In	The input representing the current gear selection with the following values assumed: PRNDL = 0: Park
		PRNDL = 1: Reverse
		PRNDL = 2: Neutral
		PRNDL = 3: Drive
		PRNDL = 4: Low
		[Gear selection in integers]

Table A.6: Input variables used in Park Assist (PA)

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

Table A.7: Output variables used in Park Assist (PA)

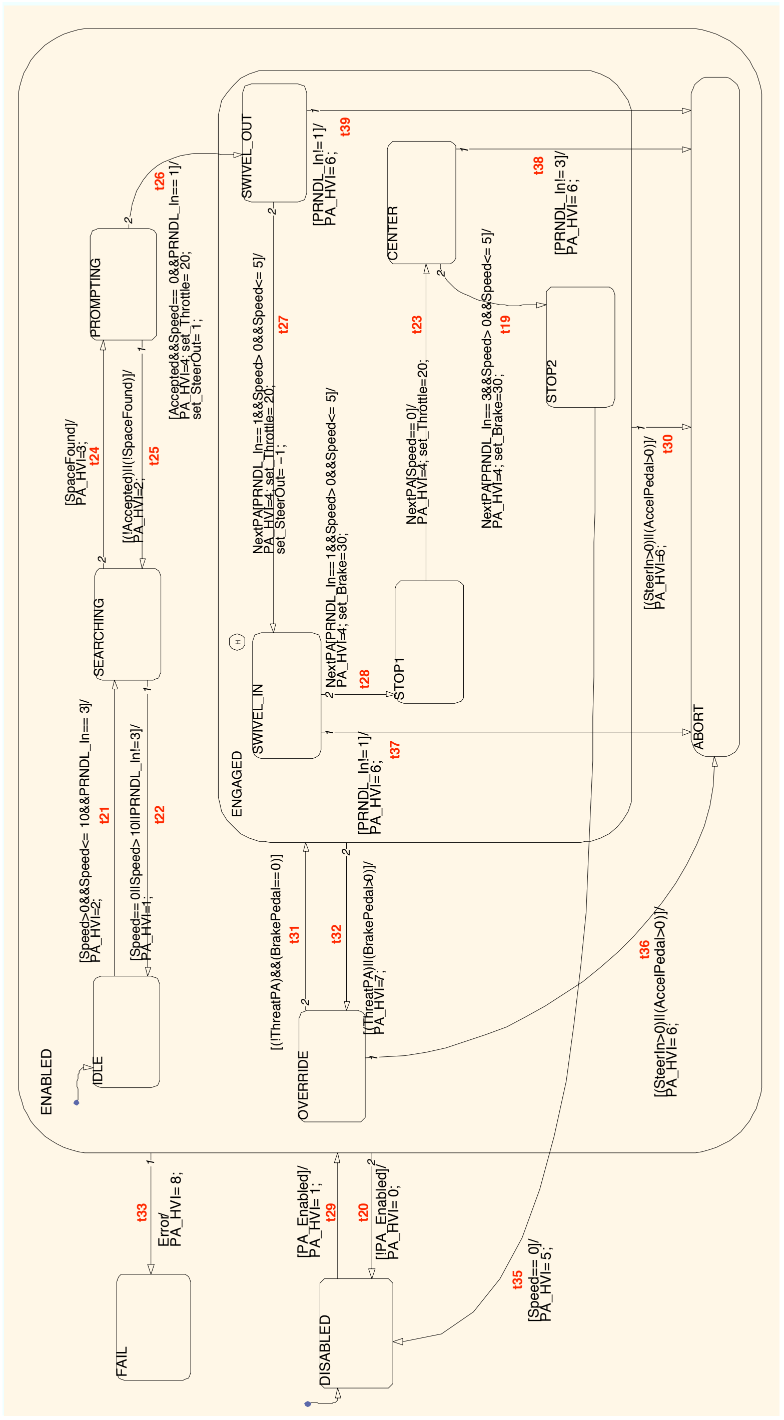

ENABLED
IDLE
SEARCHING
PROMPTING
ENGAGED
SWIVEL_OUT
SWIVEL_IN
CENTER
STOP1
STOP2
ABORT
OVERRIDE
FAIL
DISABLED
t21 [Speed>0&&Speed<= 10&&PRNDL_In== 3]/ PA_HVI=2;
t22 [Speed== 0||Speed> 10||PRNDL_In!=3]/ PA_HVI=1;
t24 [SpaceFound]/ PA_HVI=3;
t25 [(!Accepted)||(!SpaceFound)]/ PA_HVI=2;
t26 [Accepted&&Speed== 0&&PRNDL_In== 1]/ PA_HVI=4; set_Throttle= 20; set_SteerOut= 1;
t27 NextPA[PRNDL_In== 1&&Speed> 0&&Speed<= 5]/ PA_HVI=4; set_Throttle= 20; set_SteerOut= -1;
t28 NextPA[PRNDL_In== 1&&Speed> 0&&Speed<= 5]/ PA_HVI=4; set_Brake=30;
t23 NextPA[Speed== 0]/ PA_HVI=4; set_Throttle=20;
t19 NextPA[PRNDL_In== 3&&Speed> 0&&Speed<= 5]/ PA_HVI=4; set_Brake=30;
t39 [PRNDL_In!= 1]/ PA_HVI=6;
t38 [PRNDL_In!= 3]/ PA_HVI= 6;
t37 [PRNDL_In!= 1]/ PA_HVI=6;
t30 [(SteerIn>0)||(AccelPedal>0)]/ PA_HVI=6;
t31 [(!ThreatPA)&&(BrakePedal==0)]
t32 [(ThreatPA)||(BrakePedal>0)]/ PA_HVI=7;
t36 [(SteerIn>0)||(AccelPedal>0)]/ PA_HVI=6;
t33 Error/ PA_HVI= 8;
t29 [PA_Enabled]/ PA_HVI= 1;
t20 [!PA_Enabled]/ PA_HVI= 0;
t35 [Speed== 0]/ PA_HVI= 5;
H

Figure A.6: Park Assist (PA) Stateflow design model

A.4 Lane Guide (LG)

UWFMS's LG is based on TRW's LG description:

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

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

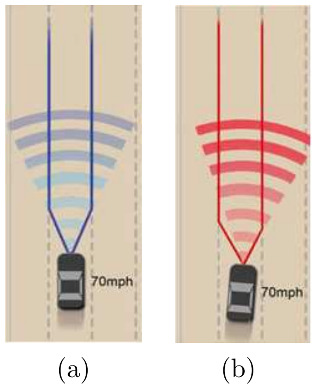


Figure A.7: Lane Guide Functionality: (a) As the LG vehicle cruises in a lane, LG monitors the lane markings and the vehicle's position; (b) If vehicle drifts from its lane, LG first can warn the driver, and even provide steering corrective input.

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

The details of USFMS's LG design in Stateflow are as follows: LG starts in the DISABLED state when the car is turned on. When the driver turns LG on by pressing the LG_Enabled button, LG enters the ENABLED state. The default of the ENABLED superstate is the DISENGAGED state. LG relies on a external module that pre-process the sensor data, which sends back to LG a LaneDrift value indicating the amount of vehicle drifting. If the vehicle is adequately centered in the lane, LaneDrift is equal to 0. If the LaneDrift input reaches the threshold of -10, the vehicle is drifting to the left. If the LaneDrift input reaches the threshold of +10, the vehicle is drifting to the right.

LG can be used in two modes: “Warn” and “Assist”. Both modes are contained within the state ENGAGED.

- If LG_Mode = 0, this means that LG is set to “Warn”, and the feature will output LG_Warning = 1 to indicate that the vehicle drifts too far to the left or to the right, but no other action other than the warning is taken. While LG_Mode = 0, if the vehicle is drifting left (*i.e.,* LaneDrift < -10), LG will enter the WARN_LEFT state, whereas if the vehicle is drifting right (*i.e.,* LaneDrift > 10), LG will enter the WARN_RIGHT state.
- If LG_Mode = 1, this means that LG is set to “Assist”, and the feature will output LG_Warning = 0 (*i.e.,* no warning indication). While LG_Mode = 1, if the vehicle is drifting left (*i.e.,* LaneDrift < -10), LG will enter the ASSIST_LEFT state, and output SteerOut = -1; if the vehicle is drifting right (*i.e.,* LaneDrift > 10), LG will enter the ASSIST_RIGHT state and output SteerOut = 1. These SteerOut output values will indicate some external component to determine and request the steering required to center the car. If the mode is set to “Assist”, the warning will not activate when the vehicle drifts because PA will act to center the vehicle in its lane and a warning during the execution of PA’s centering would be annoying to the driver.

The turn signal indication, brake pedal depression, and steering wheel input (over a threshold of 10 and -10 degrees of rotation) will all send LG from any state in ENGAGED to the OVERRIDE state. When any of these conditions cease to be present, LG goes to the DISENGAGED state.

An Error event at any time when LG is on will cause a transition to the FAIL state, which will remain the active state until the vehicle is restarted.

The definition of stable for UWFMS’s LG is defined as 1, *i.e., true* because the only one transition is taken in each big-step as ordered-compositions are not present in LG.

Type	**Name**	**Meaning**
event	Error	A signal indicating when an error has occurred
data	LG_Enabled	Driver selected main power to enable/disable the feature [Boolean]
data	LG_Mode	Mode selected by the driver, indicating that LG works in either "Warn" mode (LGmode = 0) or "Assist" mode (LGmode = 1) [Mode value in integers]
data	LaneDrift	Value to indicate if the vehicle is centered or not in the lane (input from an external sensor processing component, where a 0 indicates centered, values less than -10 indicate drifting to the left, and values greater than 10 indicates drifting to the right) [Drifting value in integers]
data	SteerIn	Driver controlled physical steering wheel input as a steering wheel angle (For LG, values are within the range (-20,20), with 0 for centered) [Angle in integers]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	TurnSignal	Value indicating if the turn signal is on or not [Boolean]

data	PRNDL_In	The input representing the current gear selection with the following values assumed:
		PRNDL = 0: Park
		PRNDL = 1: Reverse
		PRNDL = 2: Neutral
		PRNDL = 3: Drive
		PRNDL = 4: Low
		[Gear selection in integers]

Table A.8: Input variables used in Lane Guide (LG)

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

Table A.9: Output variables used in Lane Guide (LG)

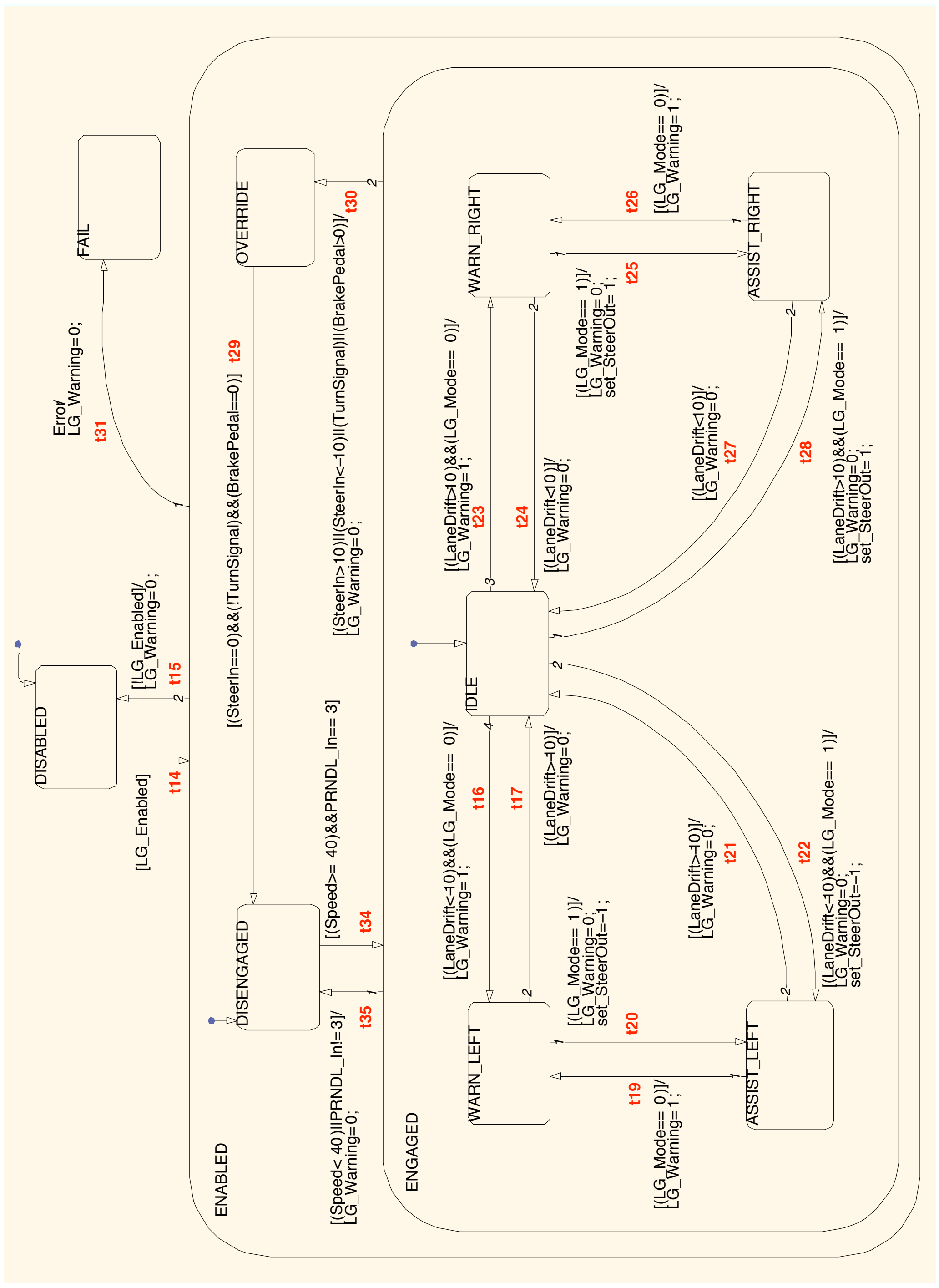


Figure A.8: Lane Guide (LG) Stateflow design model

A.5 Emergency Vehicle Avoidance (EVA)

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

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

The details of UWFMS's EVA design in Stateflow are as follows: EVA starts in the DISABLED state when the car is turned on. When the driver turns EVA on by pressing the EVA_Enabled button, EVA enters the ENABLED state.

At the ENABLED state, the feature will remain idle until a siren from an emergency vehicle is detected, as indicated by the Siren boolean input. When the Siren Boolean is true while driving forward, EVA will enters the ENGAGED state where the feature can acquire control of the vehicle. The default of the ENGAGED superstate is the SLOW state. EVA will monitor the right to see if pulling over is feasible. EVA relies on an external component that sends events back to EVA regarding the safely of a stopping location. Then, EVA receives a WayClear event when the location is safe, or a DontStop event when the location is unsafe. If at the SLOW state EVA gets as an input a WayClear value true, it is safe to pullover farther to the right, and EVA will enter the PULLOVER state. This transition to the PULLOVER state requests braking and also requests steering control by outputting SteerOut = -1, which indicates that the vehicle's wheels need to move to the right. The SteerOut output value will indicate some external component to determine and request the steering required to pull over the vehicle. In either of the SLOW or PULLOVER states, if EVA receives the DontStop boolean input value as true, EVA will enter the COAST state, and also request a slight throttle to keep the vehicle moving until a safe location is found. We assume that the value of throttle requested is adequate to ensure that the vehicle does not come to a stop in an unsafe location (*e.g.,* the middle of an intersection).

Any brake pedal depression or steering input of non-zero, and any acceleration pedal input greater than a 30% depression from the driver sends EVA to the OVERRIDE state from any state in the ENABLED superstate. When any of these conditions cease to be present, EVA goes to the ENGAGED state while the feature is still enabled. Otherwise, EVA transitions to the DISABLED state. An Error event at any time when EVA is on causes a transition to the FAIL state, which remains the active state until the vehicle is restarted.

The definition of stable for UWFMS's EVA is defined as 1, *i.e., true* because the only one transition is taken in each big-step as ordered-compositions are not present in EVA.

Type	**Name**	**Meaning**
event	Error	A signal indicating when an error has occurred
data	EVA_Enabled	Driver selected main power to enable/disable the feature [Boolean]
data	Siren	Input from an external component that uses GPS information and microphones to determine if an emergency vehicle is nearby and its whereabouts, so EVA can act by moving the EVA vehicle out of the emergency vehicle's path [Boolean]
data	WayClear	Input from an external component that uses GPS and radar information to indicate if pulling the EVA vehicle over into the far right lane is possible [Boolean]
data	DontStop	Input from an external component that uses GPS information to determine if the current braking action would cause the EVA vehicle to stop in an unsafe location [Boolean]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	AccelPedal	Value of physical pedal input represented as a percentage of maximum depression [Percentage in integers]
data	Speed	Current speed of the vehicle (value within the range of 0 to 100) [KPH in integers]

data	PRNDL_In	The input representing the current gear selection with the following values assumed: PRNDL = 0: Park
		PRNDL = 1: Reverse
		PRNDL = 2: Neutral
		PRNDL = 3: Drive
		PRNDL = 4: Low
		[Gear selection in integers]

Table A.10: Input variables used in Emergency Vehicle Avoidance (EVA)

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

Table A.11: Output variables used in Emergency Vehicle Avoidance (EVA)

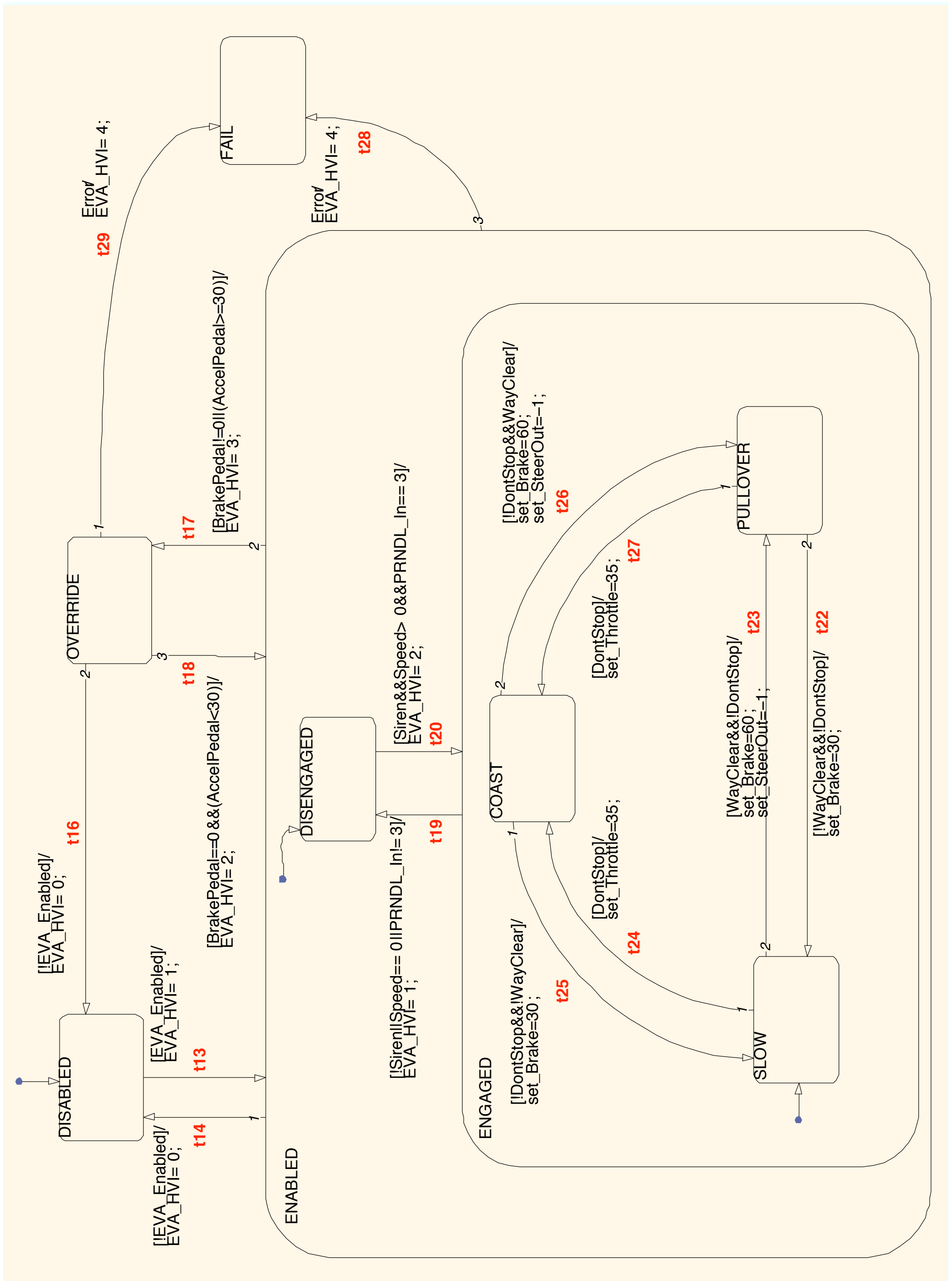


Figure A.9: Emergency Vehicle Avoidance Stateflow design model

A.6 Parking Space Centering (PSC)

The Parking Space Centering (PSC) feature assists drivers during perpendicular parking maneuvers. I assumed that the vehicle is already at a valid parking space; an error signal is sent if it is detected that this is not the case. The feature uses short range radar sensors to determine the location of the vehicle within the box.

Figure A.10 presents PSC's functionality modelled in STATEFLOW, which we will explain in the following paragraphs. Table A.12 describes the input variables and Table A.13 describes the output variables used in the PSC's STATEFLOW design model.

The details of UWFMS's PSC design in STATEFLOW are as follows: PSC defaults to the DISABLED state when the vehicle is turned on. When the drivers turns PSC on by pressing the PSC_Enabled button, PSC enters the ENABLED superstate, which defaults to the DISENGAGED state. When the vehicle is moving at less than 5 KPH while driving forward, PSC moves into the ENGAGED state, which defaults to the STRAIGHT state. At the STRAIGHT state, the vehicle begins moving straight forward, setting Throttle to 20. We assumed that the inputs for LeftLine, RightLine, and FrontLine are units of distance with 5 being a threshold for close distance to the front, and a threshold for not being centered. If the vehicle is farther from the left side than the right side (*i.e.,* LeftLine-RightLine > 5) then the feature enters the MOVE_LEFT state to correct the vehicle's position by setting SteerOut to -1 while Throttle $= 20$. If the vehicle is farther from the right side than the left side (*i.e.,* RightLine-LeftLine > 5) then the feature enters the MOVE_RIGHT state to correct the vehicle's position by setting SteerOut to 1 while Throttle $= 20$. These SteerOut output values will indicate some external component to determine and request the steering required to center the car, with the value of -1 used to represent the vehicle moving its wheels to the left and the value of +1 used to represent wheels pointed to the right. At any point while at the ENABLED state, when the front line of the parking box is 5 units with the difference between the two sides being less than 5, PA enters the HALT state. When the vehicle stops completely, PSC moves to the DISENGAGED state.

Any Error event at any time when PSC is on will cause the feature to transition to the FAILED state, which cannot be left until the vehicle is shut off and restarted. Steering, acceleration or braking by the driver sends PSC to the OVERRIDE state. When any of these conditions cease to be present, PSC goes to the ENABLED state while the feature is still enabled. Otherwise, PSC transitions to the DISABLED state.

The definition of stable for UWFMS's PSC is defined as 1, *i.e., true* because the only one transition is taken in each big-step as ordered-compositions are not present in PSC.

Type	**Name**	**Meaning**
event	Error	A signal indicating when an error has occurred
data	PSC_Enabled	Driver selected main power to enable/disable the feature [Boolean]
data	LeftLine	Input from radar, indicating the distance to the left side of the parking space, whether it is a line marking or an obstacle [Distance in integers]
data	RightLine	Input from radar, indicating the distance to the right side of the parking space, whether it is a line marking or an obstacle [Distance in integers]
data	FrontLine	Input from radar, indicating the distance to the front of the parking space, whether it is a line marking or an obstacle [Distance in integers]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	AccelPedal	Value of physical pedal input represented as a percentage of maximum depression [Percentage in integers]

data	PRNDL_In	The input representing the current gear selection with the following values assumed:
		PRNDL = 0: Park
		PRNDL = 1: Reverse
		PRNDL = 2: Neutral
		PRNDL = 3: Drive
		PRNDL = 4: Low
		[Gear selection in integers]

Table A.12: Input variables used in Parking Space Centering (PSC)

Type	Name	Meaning
data	set_Throttle	Request for throttle control of the vehicle as a percentage of maximum throttle (For PSC, only one constant value of throttle percentage is output, which corresponds to a reasonable acceleration for parking) [Percentage in integers]
data	set_Brake	An output request for braking force by the feature as a percentage of maximum braking ability (For PSC, only one value of braking is output, which corresponds to soft-braking) [Percentage in integers]
data	set_SteerOut	An output request for steering control of the vehicle (For PSC, the value -1 indicates that the vehicle shall turn the wheels to the right, 0 indicates that the wheels shall be centred, and 1 indicates that wheels shall turn to the left. Some external component will process these values and manipulate the wheels accordingly) [Steer request in integers]

Table A.13: Output variables used in Parking Space Centering (PSC)

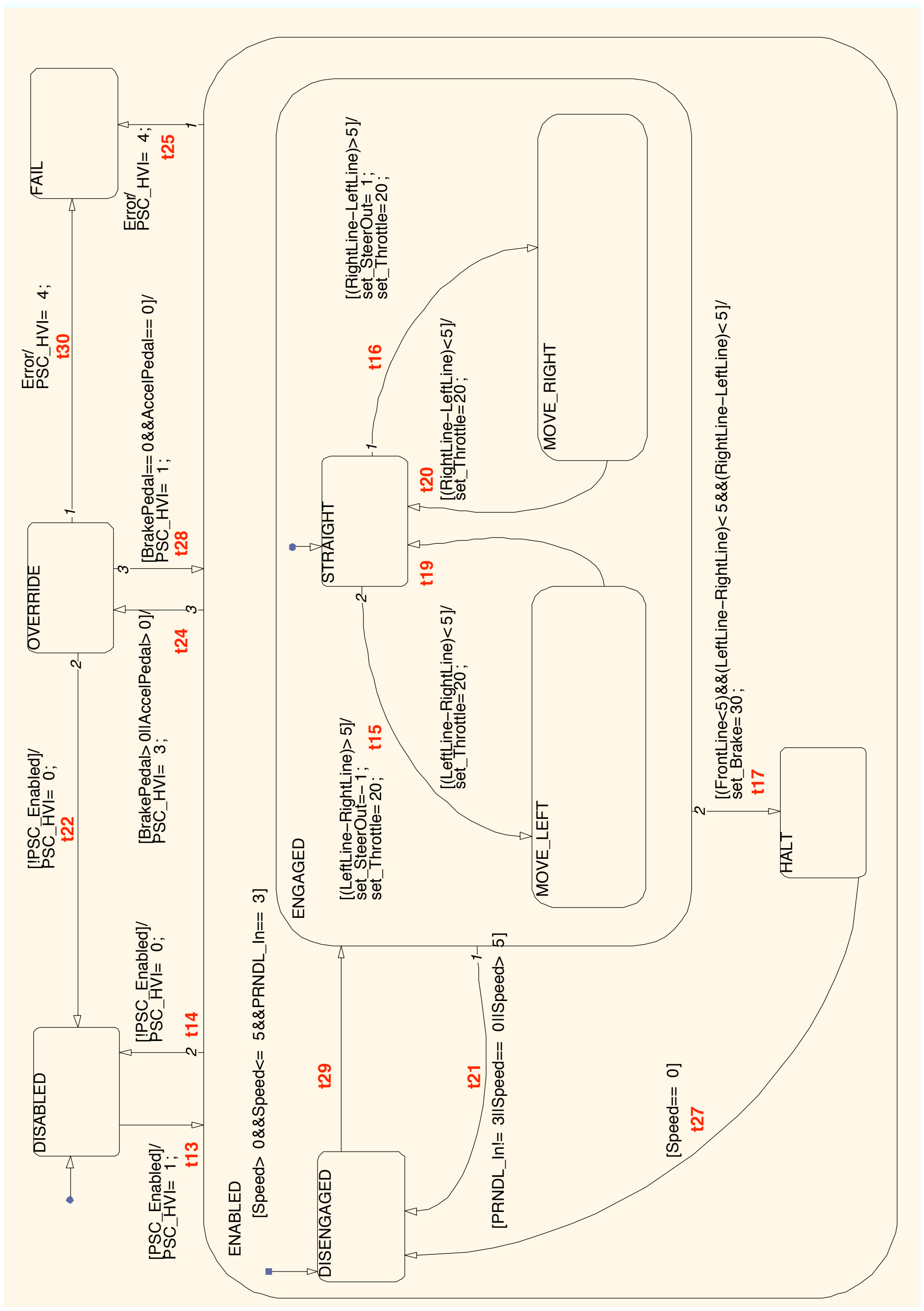

FAIL
OVERRIDE
DISABLED
ENABLED
ENGAGED
DISENGAGED
STRAIGHT
MOVE_RIGHT
MOVE_LEFT
HALT
Error/ PSC_HVI= 4; t30
Error/ PSC_HVI= 4; t25
[!PSC_Enabled]/ PSC_HVI= 0; t22
[BrakePedal== 0&&AccelPedal== 0]/ PSC_HVI= 1; t28
[BrakePedal> 0||AccelPedal> 0]/ PSC_HVI= 3; t24
[!PSC_Enabled]/ PSC_HVI= 0; t14
[PSC_Enabled]/ PSC_HVI= 1; t13
[Speed> 0&&Speed<= 5&&PRNDL_In== 3] t29
[PRNDL_In!= 3||Speed== 0||Speed> 5] t21
[Speed== 0] t27
[(RightLine−LeftLine)>5]/ set_SteerOut= 1; set_Throttle=20; t16
[(RightLine−LeftLine)<5]/ set_Throttle=20; t20
[(LeftLine−RightLine)<5]/ set_Throttle=20; t19
[(LeftLine−RightLine)>5]/ set_SteerOut=−1; set_Throttle= 20; t15
[(FrontLine<5)&&(LeftLine−RightLine)< 5&&(RightLine−LeftLine)<5]/ set_Brake=30; t17
1
2
3
3
1
2
1
2
1
2

Figure A.10: Parking Space Centering (PSC) Stateflow design model

A.7 Reversing Assistance (RA)

The Reversing Assistant (RA) feature can assist drivers by helping to prevent or mitigate collisions while reversing. Combining long and short range radars, RA monitors the path of the vehicle while reversing. In the event that a vehicle or obstacle approaches, RA notifies the driver of a possible collision and also brakes as soon as the threat of a collision becomes imminent.

Figure A.11 presents RA's functionality modelled in Stateflow, which we will explain in the following paragraphs. Table A.14 describes the input variables and Table A.15 describes the output variables used in the RA's Stateflow design model.

The details of UWFMS's RA design in Stateflow are as follows: RA will start at the DISABLED state when the vehicle is turned on. RA turns on by toggling the RA_Enabled button, making RA to enter the ENABLED superstate, which defaults to DISENGAGED. RA will remain in the DISENGAGED state until the vehicle is put into reverse, indicated by the input PRNDL_In = 1, while the vehicle moves between 10 and 25 KPH. This action will cause RA to enter the ENGAGED superstate, which defaults to the IDLE state. When engaged, RA will be able to warn the driver and/or take action if a possible threat is encountered. RA relies on an external component to pre-process sensor inputs and convert them into a threat input for RA, stored in ObstacleZone. The threat input data could be either: None=0, Mild=1, Imminent=2. If a threat input is received while in either the IDLE, WARN, or ASSIST states, one of the following transitions will occur depending on the threat:

- In the case that no threat is detected, or the obstacle ceases to be present, no warning and no braking intervention are made. RA changes to the IDLE state.
- In the case of a mild threat, a mild threat warning will be output (*i.e.,* Warning = 1) with no braking intervention. RA enters the WARN state.
- In the case of an imminent threat, an imminent threat warning will be output (*i.e.,* Warning = 2) and a braking request will be made to mitigate the collision as much as possible (*i.e.,* Brake = 60%). RA enters the ASSIST state.

If the vehicle stops while the feature is in the ASSIST state, RA will enter the HOLD state and remain there until an input of BrakePedal $>$ 20 is detected, so RA moves to the DISENGAGED state and the driver can regain control of the vehicle. Whenever the vehicle's gear selection changes from reverse (*i.e.,* if PRNDL != 1) at any time in the ENGAGED superstate, RA becomes DISENGAGED. A driver acceleration pedal input exceeding 35% of depression at any time will cause RA to enter the OVERRIDE state until the input becomes below the threshold, which causes RA to go back to the ENABLED state as long as the feature is still enabled. An Error event at any time when RA is on will send the feature to the FAILED state, where RA will remain until the car is turned off.

The definition of stable for UWFMS's RA is defined as 1, *i.e., true* because the only one transition is taken in each big-step as ordered-compositions are not present in RA.

Type	**Name**	**Meaning**
event	Error	A signal indicating when an error has occurred
data	RA_Enabled	Driver selected main power to enable/disable the feature [Boolean]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	AccelPedal	Value of physical pedal input represented as a percentage of maximum depression [Percentage in integers]
data	Speed	Current speed of the vehicle (value within the range of 0 to 100) [KPH in integers]

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

data	PRNDL_In	The input representing the current gear selection with the following values assumed:
		PRNDL = 0: Park
		PRNDL = 1: Reverse
		PRNDL = 2: Neutral
		PRNDL = 3: Drive
		PRNDL = 4: Low
		[Gear selection in integers]

Table A.14: Input variables used in Reversing Assistant (RA)

Type	Name	Meaning
data	set_Brake	An output request for braking force by the feature as a percentage of maximum braking ability (For RA, only one value of braking is output, which corresponds to mid-force braking) [Percentage in integers]

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

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

Table A.15: Output variables used in Reversing Assistant (RA)

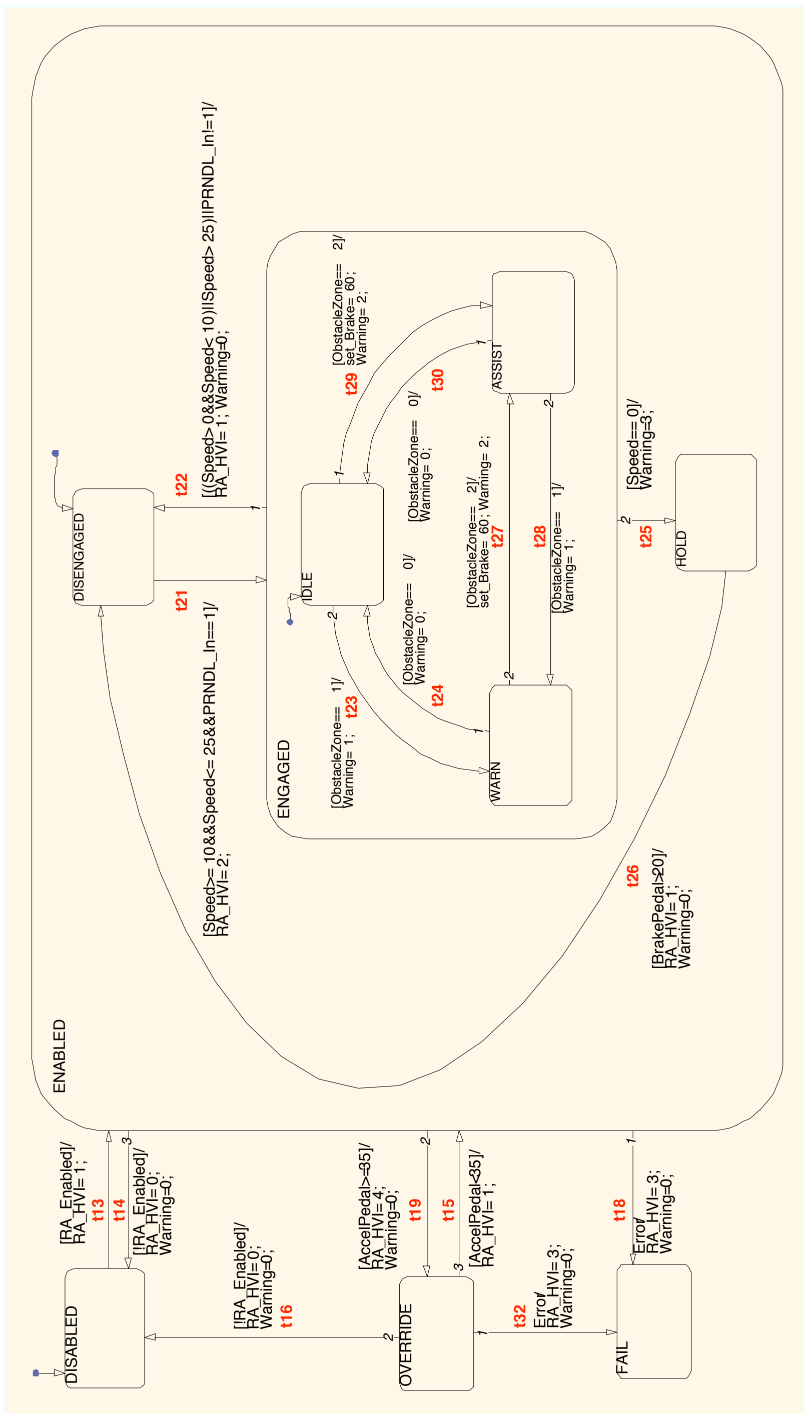

ENABLED
DISENGAGED
ENGAGED
IDLE
ASSIST
WARN
HOLD
DISABLED
OVERRIDE
FAIL
t22 [((Speed> 0&&Speed< 10)||Speed> 25)||PRNDL_In!=1]/ RA_HVI= 1; Warning=0;
t21 [Speed>= 10&&Speed<= 25&&PRNDL_In==1]/ RA_HVI=2;
t29 [ObstacleZone== 2]/ set_Brake= 60; Warning= 2;
t30 [ObstacleZone== 0]/ Warning= 0;
t27 [ObstacleZone== 2]/ set_Brake= 60; Warning= 2;
t28 [ObstacleZone== 1]/ Warning= 1;
t23 [ObstacleZone== 1]/ Warning= 1;
t24 [ObstacleZone== 0]/ Warning= 0;
t25 [Speed==0]/ Warning=3;
t26 [BrakePedal>20]/ RA_HVI=1; Warning=0;
t13 [RA_Enabled]/ RA_HVI=1;
t14 [!RA_Enabled]/ RA_HVI=0; Warning=0;
t16 [!RA_Enabled]/ RA_HVI=0; Warning=0;
t19 [AccelPedal>=35]/ RA_HVI=4; Warning=0;
t15 [AccelPedal<35]/ RA_HVI=1;
t18 Error/ RA_HVI=3; Warning=0;
t32 Error/ RA_HVI=3; Warning=0;

Figure A.11: Reversing Assistant (RA) Stateflow design model

A.8 Summary

My research effort is helped by producing the base for future feature interaction analysis, creating a set of non-proprietary automotive feature design models in Stateflow, called "University of Waterloo Feature Model Set" (UWFMS). This feature model set was created because there was no other set that I could have used to validate my methods and tools. The feature design models part of the UWFMS follow the syntactic modelling rules described in Section 4.2. Many of the design decision made were influenced by my interaction with the design engineers during my visits to GM Research and Development.

References

[1] Esterel Technologies. SCADE Suite Product Description, http://www.esterel-technologies.com. 65

[2] MathWorks Stateflow Documentation, http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/. 24

[3] Reactive Systems, Inc, http://www.reactive-systems.com. 65

[4] Rockwell Collins, http://www.rockwellcollins.com/. 65

[5] TRW automotive - the global leader in automotive safety systems, http://www.trw.com/. 4, 173

[6] *Z Formal Specification Notation — Syntax, Type System and Semantics, ISO/IEC 13568:2002*. International Organization for Standardization (ISO), 2002. 42

[7] *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*. International Telecommunication Union (ITU), 2007. 10, 42, 68

[8] Nasa langley formal methods site, http://shemesh.larc.nasa.gov/fm/index.html, 2008. 26

[9] R. Accorsi, C. Areces, W. Bouma, and M. de Rijke. Features as constraints. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 210–225. IOS Press, 2000. 19, 42

[10] S. Aggarwal, R. P. Kurshan, and K. K. Sabnani. A calculus for protocol specification and validation. In *Protocol Specification, Testing, and Verification*, pages 19–34, 1983. 28

[11] I. Aggoun and P. Combes. Observers in the SCE and SEE to detect and resolve feature interactions. In *Feature Interactions in Telecommunications Networks IV*, pages 198–212. IOS Press, 1997. 20

[12] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. In *International Workshop on Graph Transformation and Visual Modeling Techniques*, 2004. 66

[13] M. Amer, A. Karmouch, T. Gray, and S. Mankovski. Feature interaction resolution using fuzzy policies. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 94–112. IOS Press, 2000. 20

[14] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *International Conference on Engineering of Complex Computer Systems*, pages 212–221, 2001. 103

[15] P. K. Au and J. M. Atlee. Evaluation of a state-based model of feature interactions. In *Feature Interactions in Telecommunications Networks IV*, pages 153–167. IOS Press, 1997. 19, 42

[16] R. Baillargeon. Personal communication, 2008. 49, 179

[17] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. *SIGPLAN Notices*, 38(1):97–105, 2003. 10, 11, 68, 103

[18] C. Banphawatthanarak and B. H. Krogh. Verification of stateflow diagrams using smv: sf2smv 2.0. Technical Report CMU-ECE-2000-020, Carnegie Mellon University, 2000. 65

[19] L. Baresi, C. Ghezzi, A. Miele, M. Miraz, A. Naggi, and F. Pacifici. Hybrid service-oriented architectures: a case-study in the automotive domain. In *International Workshop on Software Engineering and Middleware*, pages 62–68. ACM, 2005. 3

[20] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Conference on Design Automation*, pages 970–975. ACM, 1999. 103

[21] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. MacKenzie. *Systems and Software Verification – Model Checking Techniques and Tools*. Springer, 2001. 26, 29

[22] J. Bereisa. Applications of Microcomputers in Automotive Electronics. *IEEE Transactions on Industrial Electronics*, IE-30(2):87–96, 1983. 1

[23] J. Bergstra and W. Bouma. Models for feature descriptions and interactions. In *Feature Interactions in Telecommunications Networks IV*, pages 31–45. IOS Press, 1997. 19, 42

[24] L. Blair, G. Blair, J. Pang, and C. Efstratiou. Feature' interactions outside a telecom domain. In *Workshop on Feature Interactions in Composed Systems*, pages 15–20, 2001. 21

[25] J. Blom. Formalisation of requirements with emphasis on feature interaction detection. In *Feature Interactions in Telecommunications Networks IV*, pages 61–77. IOS Press, 1997. 19, 42

[26] J. Blom, B. Jonsson, and L. Kempe. Using temporal logic for modular specification of telephone services. In *Feature Interactions in Telecommunications Systems*, pages 197–216. IOS Press, 1994. 19, 42

[27] C. Blundell, K. Fisler, S. Krishnamurthi, and P. V. Hentenryck. Parameterized interfaces for open system verification of product lines. In *International Conference on Automated Software Engineering*, pages 258–267. IEEE Computer Society, 2004. 129

[28] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. In *Computer Networks*, volume 14, pages 25–59, 1987. 42

[29] M. Boström and M. Engstedt. Feature interaction detection and resolution in the delphi framework. In *Feature Interactions in Telecommunications Systems III*, pages 157–172. IOS Press, 1995. 19, 42

[30] L. G. Bouma and H. Velthuijsen. Introduction to feature interactions in telecommunication systems. In L. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages vii–xiv, 1994. 18

[31] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y. Lin. The feature interaction problem in telecommunications systems. In *International Conference on Software Engineering for Telecommunications Switching Systems*, pages 59–62, July 1989. 1, 2

[32] K. H. Braithwaite and J. M. Atlee. Towards automated detection of feature interactions. In *Feature Interactions in Telecommunications Systems II*, pages 36–59. IOS Press, 1994. 20

[33] J. Bredereke. Families of formal requirements in telephony switching. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 257–273. IOS Press, 2000. 19, 41

[34] M. Broy. Challenges in automotive software engineering. In *International Conference on Software Engineering*, pages 33–42. ACM, 2006. 1, 2

[35] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann. Engineering Automotive Software. *Proc. of the IEEE*, 95(2):356–373, 2007. 1, 2

[36] G. Bruns, P. Mataga, and I. Sutherland. Features as services transformers. In *Feature Interactions in Telecommunications and Software Systems V*, pages 85–97. IOS Press, 1998. 19, 42

[37] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. 28

[38] R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski. Feature-interaction visualization and resolution in an agent environment. In *Feature Interactions in Telecommunications and Software Systems V*, pages 135–149. IOS Press, 1998. 20

[39] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142 – 170, 1992. 28

[40] M. Cain. Managing run-time interactions between call processing features. In *IEEE Communications Magazine*, volume 30, pages 44–50, 1992. 20

[41] M. Calder, M. Kolberg, E. Magill, D. Marples, and S. Reiff-Marganiec. Hybrid solutions to the feature interaction problem. In *Feature Interactions in Telecommunications and Software Systems VII*, pages 295–312. IOS Press, 2003. 20

[42] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003. 3, 7, 34, 41

[43] M. Calder, E. H. Magill, and D. Marples. Hybrid approach to software interworking problems: Managing interactions between legacy and evolving telecommunications software. In *IEE Proceedings - Software*, volume 146, pages 167–180, 1999. 20

[44] M. Calder and A. Miller. Using SPIN for feature interaction analysis: a case study. In *International SPIN workshop on Model Checking of Software*, pages 143–162. Springer-Verlag, 2001. 19, 42

[45] M. Calder and S. Reiff-Marganiec. Modelling legacy telecommunications switching systems for interaction analysis. In *Systems Engineering for Business Process Change*, pages 182–195. Springer, 2000. 20

[46] K. Camera. SF2VHD: A stateflow to VHDL translator. Master's thesis, University of California, Berkeley, 2001. 66

[47] E. J. Cameron and H. Veldhuijsen. Feature interactions in telecommunications systems (a tutorial). pages 18–23, Aug. 1993. 17, 18

[48] C. Capellmann, P. Combes, J. Petterson, B. Renard, and J. L. Ruiz. Consistent interaction detection - a comprehensive approach integrated with service creation. In *Feature Interactions in Telecommunications Networks IV*, pages 183–197. IOS Press, 1997. 19, 42

[49] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998. 53, 64, 66

[50] R. N. Charette. This car runs on code, `http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code`. 1

[51] M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. *International Journal on Software Tools for Technology Transfer*, 9(5):429–445, 2007. 10, 68, 85, 97, 102

[52] K. T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *International Design Automation Conference*, pages 86–91. ACM, 1993. 10, 68, 70

[53] C. Chi and R. Hao. Test generation for interaction detection in feature-rich communication systems. *Computer Networks*, 51(2):426 – 438, 2007. 21, 43

[54] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A New Symbolic Model Verifier. In *International Conference on Computer Aided Verification*, pages 495–499. Springer-Verlag, 1999. 65, 168

[55] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131, pages 52–71. Springer-Verlag, 1982. 28

[56] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994. 30

[57] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000. 7, 26, 97, 168

[58] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *International Conference on Software Engineering*, pages 321–330. ACM, 2011. 129

[59] P. Combes and S. Pickin. Formalisation of a user view of network and services for feature interaction detection. In *Feature Interactions in Telecommunications Systems*, pages 120–135. IOS Press, 1994. 19, 42

[60] F. Copty, A. Irron, O. Weissberg, N. P. Kropp, and G. Kamhi. Efficient debugging in a formal verification environment. In *Conference on Correct Hardware Design and Verification Methods*, pages 275–292, 2001. 10, 11, 68, 69, 86, 102

[61] R. G. Crespo. Predicting feature interactions by using inconsistency models. *Computer Networks*, 54(3):416–427, 2010. 22

[62] R. G. Crespo, M. Carvalho, and L. Logrippo. Distributed resolution of feature interactions for internet applications. *Computer Networks*, 51(2):382 – 397, 2007. Feature Interaction. 21, 22, 43

[63] J. Dabney and T. L. Harman. *Mastering Simulink*. Pearson/Prentice Hall, 2004. 9, 24

[64] L. de Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation: a synchronous point of view. In *Feature Interactions in Telecommunications and Software Systems V*, pages 262–275. IOS Press, 1998. 19, 42

[65] R. Debouk, B. Czerny, J. D'Ambrosio, and J. Joyce. Safety Analysis of Software-intensive Motion Control Systems. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, 2(1):281–286, 2009. 3

[66] R. Debouk, B. Czerny, J. D'Ambrosio, and J. J. Joyce. Safety strategy for autonomous systems. In *International Systems Safety Conference*. System Safety Society, 2011. 3

[67] D. D'Souza and M. Gopinathan. Conflict-tolerant features. In *International Conference on Computer Aided Verification*, volume 5123 of *LNCS*, pages 227–239. Springer Berlin / Heidelberg, 2008. 7, 23, 44, 128

[68] D. D'Souza, M. Gopinathan, S. Ramesh, and P. Sampath. Conflict-tolerant real-time features. *International Conference on Quantitative Evaluation of Systems*, pages 274–283, 2008. 23, 44, 128

[69] E. A. Emerson. The beginning of model checking: A personal perspective. In *25 Years of Model Checking*, volume 5000 of *LNCS*, pages 27–45. Springer Berlin / Heidelberg, 2008. 29

[70] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996. 30

[71] A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 384–398. Springer-Verlag, 1997. 103

[72] S. Esmaeilsabzali, N. Day, J. Atlee, and J. Niu. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering*, 15(2):235–265, 2010. 12, 46, 64

[73] A. Felty and K. Namjoshi. Feature specification and automatic conflict detection. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 179–192. IOS Press, 2000. 19, 42

[74] Ford. Structured analysis and design usign matlab/simulink/stateflow – modeling style guidelines. Technical report, Ford, 1999. 49

[75] M. Frappier, A. Mili, and J. Desharnais. Detecting feature interactions in relational specifications. In *Feature Interactions in Telecommunications Networks IV*, pages 123–137. IOS Press, 1997. 19, 42

[76] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 134–143. ACM, 2002. 103

[77] N. Fritsche. Run-time resolution of feature interactions in architectures with separated call and feature control. In *Feature Interactions in Telecommunications III*, pages 43–63. IOS Press, 1995. 20

[78] A. Gammelgaard and J. E. Kristensen. Interaction detection, a logical approach. In *Feature Interactions in Telecommunications Systems*, pages 178–196, 1994. 19, 42

[79] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Software Engineering Notes*, 24(6):146–162, 1999. 103

[80] J. P. Gibson. Feature requirements models: Understanding interactions. In *Feature Interactions in Telecommunications Networks IV*, pages 46–60. IOS Press, 1997. 19, 42

[81] J. P. Gibson. Towards a feature interaction algebra. In *Feature Interactions in Telecommunications and Software Systems V*, pages 217–231. IOS Press, 1998. 19, 42

[82] A. Gouya and N. Crespi. Detection and resolution of feature interactions in IP multimedia subsystem. *International Journal of Network Management*, 9(4):315–337, 2009. 21, 43

[83] N. Griffeth and H. Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In *Feature Interactions in Telecommunications Systems II*, pages 217–235. IOS Press, 1994. 20

[84] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135. Springer, 2003. 10, 68, 102

[85] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proc. of the IEEE*, 79(9):1305–1320, 1991. 65

[86] R. J. Hall. Feature combination and interaction detection via foreground/background models. In *Feature Interactions in Telecommunications and Software Systems V*, pages 232–246. IOS Press, 1998. 6, 19, 21, 42

[87] R. J. Hall. Feature Interactions in Electronic Mail. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 67–82. IOS Press, 2000. 22, 43

[88] G. Hamon and J. Rushby. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer*, 9(5–6):447–456, 2007. 49

[89] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8(3):231–274, June 1987. 9, 10, 24, 46, 68

[90] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *International Worshop on Formal Approaches to Testing of Software*, volume 2931 of *LNCS*, pages 42–59, 2003. 103

[91] M. Heisel and J. Souquières. A heuristic approach to detect feature interactions in requirements. In *Feature Interactions in Telecommunications and Software Systems V*, pages 165–171. IOS Press, 1998. 19, 41

[92] A. Hitchcock. Methods of analysis of IVHS safety. Technical report, Institute of Transportation Studies, UC Berkeley, 1992. 3

[93] G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003. 11, 42, 68, 97, 102, 168

[94] S. Homayoon and H. Singh. Methods of addressing the interactions of intelligent network services with embedded switch services. In *IEEE Communications Magazine*, volume 26, pages 42–46. IEEE Computer Society, 1988. 20

[95] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *International Conference on Software Engineering*, pages 232–243, 2003. 103

[96] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *Software Engineering*, 24(10):831–847, 1998. 20

[97] Y. Jia and J. M. Atlee. Run-time management of feature interactions. In *ICSE Workshop on Component-Based Software Engineering*, pages 115–134. IOS Press, 2003. 20

[98] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 445–459. Springer-Verlag, 2002. 11, 68, 102

[99] J. J. Joyce. Personal communication, 2008. 13, 98, 132, 173

[100] A. L. Juarez Dominguez. Verification of DFC call protocol correctness criteria. Master's thesis, School of Computer Science, University of Waterloo. May, 2005. 31

[101] A. L. Juarez Dominguez and N. A. Day. Compositional reasoning for port-based distributed systems. In *International Conference on Automated Software Engineering*, pages 376–379. ACM Press, 2005. 31

[102] A. L. Juarez Dominguez, N. A. Day, and R. T. Fanson. A preliminary report on tool support and methodology for feature interaction detection. Technical Report CS-2007-44, University of Waterloo, 2007. 14, 15, 98, 131

[103] A. L. Juarez Dominguez, N. A. Day, and R. T. Fanson. Translating Models of Automotive Features in MATLAB's Stateflow to SMV to Detect Feature Interactions. In *International Systems Safety Conference*. System Safety Society, 2008. 14, 15

[104] A. L. Juarez Dominguez, N. A. Day, and J. J. Joyce. Modelling Feature Interactions in the Automotive Domain. In *International Workshop on Modeling in Software Engineering*, pages 45–50. ACM Press, 2008. 2, 6, 12, 14, 15, 18

[105] A. L. Juarez Dominguez, J. J. Joyce, and R. Debouk. Feature interaction as a source of risk in complex software-intensive systems. In *International Systems Safety Conference*. System Safety Society, 2007. 3, 7, 14, 15, 26

[106] D. Kalita and P. P. Khargonekar. SF2STeP: A CAD tool for formal verification of timed stateflow diagrams. In *International Symposium on Computer Aided Control Systems Design*, pages 156–162, 2000. 66

[107] J. Kamoun and L. Logrippo. Goal-oriented feature interaction detection in the intelligent network model. In *Feature Interactions in Telecommunications and Software Systems V*, pages 172–186. IOS Press, 1998. 19, 42

[108] S. Kawauchi and T. Ohta. Mechanism for 3-way feature interactions occurrence and a detection system based on the mechanism. In *Feature Interactions in Telecommunications and Software Systems VII*, pages 313–328. IOS Press, 2003. 36

[109] D. O. Keck. A tool for the identification of interaction-prone call scenarios. In *Feature Interactions in Telecommunications and Software Systems V*, pages 276–290. IOS Press, 1998. 19, 41

[110] D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, 1998. 34, 41, 43

[111] A. Khoumsi. Detection and resolution of interactions between services of telephone networks. In *Feature Interactions in Telecommunications Networks IV*, pages 78–92, 1997. 19, 42

[112] A. Khoumsi and R. Bevelo. A detection method developed after a thorough study of the contest held in 1998. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 226–240. IOS Press, 2000. 19

[113] K. Kimbler. Towards a more efficient feature interaction analysis - a statistical approach. In *Feature Interactions in Telecommunications III*, pages 201–211. IOS Press, 1995. 19, 41

[114] K. Kimbler, E. Kuisch, and J. Muller. Feature interactions among pan-european services. In *Feature Interactions in Telecommunications Systems*, pages 73–85. IOS Press, 1994. 19, 41

[115] K. Kimbler and D. Sobirk. Use case drivem analysis of feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 167–177. IOS Press, 1994. 19, 41

[116] M. Kolberg, E. H. Magill, and M. Wilson. Compatibility issues between services and supporting networked appliances. In *IEEE Communications Magazine*, volume 41, pages 136–147, 2003. 22, 43

[117] D. Kroening and G. Weissenbacher. Counterexamples with loops for predicate abstraction. In *International Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 152–165. Springer Berlin / Heidelberg, 2006. 103

[118] I. H. Kruger, E. C. Nelson, and K. V. Prasad. Service-based software development for automotive applications. In *Convergence International Congress and Exposition On Transportation Electronic*. SAE International, 2004. 3

[119] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994. 29

[120] T. F. LaPorta, D. Lee, Y. Lin, and M. Yannakakis. Protocol feature interactions. In *International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*, pages 59–74. Kluwer, B.V., 1998. 19, 42

[121] E. A. Lee. Cyber-Physical Systems - Are Computing Foundations Adequate? *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, 2006. 2

[122] N. G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 2001. 5, 10, 68

[123] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20:684–707, 1994. 66

[124] M. Lochau and U. Goltz. Feature interaction aware test case generation for embedded control systems. *Electronic Notes in Theoretical Computer Science*, 264:37–52, 2010. 2, 7, 18, 23, 43, 128

[125] D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon University, 1993. 30

[126] J. Lu, J. Rupp, D. S. Rhode, M. Lopez, and L. Tellis. Active safety system. Patent number US 2008/0147277 A1, 2008. 4

[127] Y. Lu, J. M. Atlee, N. A. Day, and J. Niu. Mapping template semantics to SMV. In *International Conference on Automated Software Engineering*, pages 320–325. IEEE Computer Society, 2004. 66

[128] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, First edition, 1992. 27, 69

[129] D. Marples and E. H. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *Feature Interactions in Telecommunications and Software Systems V*, pages 115–134. IOS Press, 1998. 20

[130] K. L. McMillan. *Symbolic model checking.* Kluwer Academic, 1993. 9, 28, 30

[131] A. Metzger. Feature interactions in embedded control systems. *Computer Networks*, 45(5):625 – 644, 2004. Directions in Feature Interaction Research. 23, 35, 43

[132] A. Metzger and C. Webel. Feature Interaction Detection in Building Control Systems by Means of a Formal Product Model. In *Feature Interactions in Telecomunication and Software Systems*, pages 105–121. IOS Press, 2003. 23

[133] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Communications of the ACM*, 53(2):58–64, 2010. 65

[134] R. L. Mitchell. Toyota's lesson: Software can be unsafe at any speed, `http://blogs.computerworld.com/15547/toyotas_lesson_software_can_be_unsafe_at_any_speed`. 1

[135] M. Nakamura, Y. Kakuda, and T. Kikuno. Feature interaction detection using permutation symmetry. In *Feature Interactions in Telecommunications and Software Systems V*, pages 187–201. IOS Press, 1998. 19, 42

[136] M. Nakamura, P. Leelaprute, K. ichi Matsumoto, and T. Kikuno. On detecting feature interactions in the programmable service environment of internet telephony. *Computer Networks*, 45(5):605 – 624, 2004. Directions in Feature Interaction Research. 22, 43

[137] J. Niu, J. M. Atlee, and N. A. Day. Template semantics for model-based notations. *IEEE Transactions on Software Engineering*, 29:866–882, 2003. 66

[138] C. Norris Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996. 30

[139] J. O'Donnell. Advances in car technology bring high-class headaches, `http://www.usatoday.com/tech/news/2003-11-11-carrepairs_x.htm`. 1

[140] J. Pang and L. Blair. An adaptive run time manager for the dynamic integration and interaction resolution of features. In *International Conference on Distributed Computing Systems*, pages 445–450. IEEE Computer Society, 2002. 20

[141] J. Pang and L. Blair. Separating interaction concerns from distributed feature components. In *ETAPS Workshop on Software Composition*, volume 82 of *LNCS*, 2003. 22, 43

[142] D. Peled. Verification for robust specification. In *Theorem Proving in Higher Order Logics*, volume 1275 of *LNCS*, pages 231–241. Springer Berlin / Heidelberg, 1997. 29

[143] D. Peled. *Software Reliability Methods*. Springer, 2001. 12, 26, 28, 29, 133

[144] D. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of omega-regular languages. In *International Conference on Concurrency Theory*, pages 596–610. Springer-Verlag, 1996. 29

[145] P. J. Pingree and E. Mikk. The hivy tool set. volume 3114, pages 466–469. Springer, 2004. 66

[146] M. Plath and M. Ryan. Plug-and-play features. In *Feature Interactions in Telecommunications and Software Systems V*, pages 150–164. IOS Press, 1998. 19, 42

[147] M. Plath and M. Ryan. Defining features for CSP: Reflections on the feature interaction contest. In *Language Constructs for Describing Features*, pages 202–216. Springer Verlag, 2000. 19, 42

[148] K. P. Pomakis and J. M. Atlee. Reachability analysis of feature interactions: A progress report. In *International Symposium on Software Testing and Analysis*, pages 216–223. ACM SIGSOFT, 1996. 19, 42

[149] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *Future of Software Engineering*, pages 55–71. IEEE Computer Society, 2007. 1, 2

[150] F. Pu and W. Zhang. LTL model checking via search space partition. In *International Conference on Quality Software*, pages 418–428. IEEE Computer Society, 2006. 162

[151] F. Pu and W. Zhang. Combining search space partition and abstraction for ltl model checking. *Science in China Series F: Information Sciences*, 50(6):793–810, 2007. 162

[152] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer Berlin / Heidelberg, 1982. 28

[153] S. Reiff-Marganiec. Identifying resolution choices for an online feature manager. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 113–128. IOS Press, 2000. 20

[154] S. Reiff-Marganiec and M. Nakamura, editors. *Feature Interactions in Telecommunications and Software Systems X*. IOS Press, 2009. 5

[155] Z. E. Research. GM recalls on a software glitch, `http://www.zacks.com/stock/news/48854/GM+Recalls+on+a+Software+Glitch`. 1

[156] S. M. Rochefort and H. J. Hoover. An exercise in using constructive proof systems to address feature interactions in telephony. In *Feature Interactions in Telecommunications Networks IV*, pages 329–341. IOS Press, 1997. 19, 42

[157] N. Scaife, C. Sofronis, P. Caspi, et al. Defining and translating a "safe" subset of simulink/stateflow into lustre. In *International Conference on Embedded software*, pages 259–268. ACM Press, 2004. 66

[158] R. Sebastiani, E. Singerman, S. Tonetta, and M. Y. Vardi. GSTE is partitioned model checking. *Formal Methods in System Design*, 31(2):177–196, 2007. 163

[159] N. Sharygina and D. Peled. A combined testing and verification approach for software reliability. In *International Symposium of Formal Methods Europe*, pages 611–628. Springer-Verlag, 2001. 102

[160] M. L. Shooman. *Probabilistic Reliability: An Engineering Approach.* Brooklyn Polytechnic Institute series. McGraw-Hill, first edition, 1968. 5

[161] S. Siddiqi and J. M. Atlee. A hybrid model for specifying features and detecting interactions. In *Computer Networks*, volume 32, pages 471–485. Elsevier Science, 2000. 19

[162] K. K. Singh and G. Agnihotri. *System design through MATLAB, Control Toolbox and SIMULINK.* Springer Verlag, 2001. 17

[163] B. Stepien and L. Logrippo. Representing and verifying intentions in telephony features using abstract data types. In *Feature Interactions in Telecommunications Systems III*, pages 141–155. IOS Press, 1995. 19, 42

[164] J. G. Thistle, R. P. Malhamé, H. Hoang, and S. Lafortune. Feature interaction modelling, detection and resolution: A supervisory control approach. In *Feature Interactions in Telecommunications Networks IV*, pages 93–107. IOS Press, 1997. 19, 23, 42, 128

[165] M. Thomas. Modelling and analysing user views of telecommunications sevices. In *Feature Interactions in Telecommunications Networks IV*, pages 168–182. IOS Press, 1997. 19, 42

[166] S. Trage. Electronics: Driving Automotive Innovation, 2005. 1

[167] S. Tsang and E. H. Magill. Detecting feature interactions in the intelligent network. In *Feature Interactions in Telecommunications Systems*, pages 236–248. IOS Press, 1994. 20

[168] S. Tsang and E. H. Magill. Behaviour based run-time feature interaction problem in networked multimedia services. In *Feature Interactions in Telecommunications Networks IV*, pages 254–270. IOS Press, 1997. 20

[169] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *International Conference on Aspect-oriented Software Development*, pages 158–167. ACM Press, 2003. 18

[170] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. In *Journal of Systems and Software*, volume 49, pages 3–15, 1999. 2, 18

[171] K. J. Turner. Validating architectural feature descriptions using LOTOS. In *Feature Interactions in Telecommunications and Software Systems V*, pages 247–261. IOS Press, 1998. 20

[172] G. Utas. A pattern language of feature interactions. In *Feature Interactions in Telecommunications and Software Systems V*, pages 98–114. IOS Press, 1998. 20

[173] R. van der Linden. Using an architecture to help beat feature interactions. In *Feature Interactions in Telecommunications Systems II*, pages 24–35. IOS Press, 1994. 20

[174] R. Van Der Straeten and J. Brichau. Features and feature interactions in software engineering using logic. In *Feature Interactions in Composed Systems*, pages 79–88, 2001. 19

[175] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science*, pages 332–344, 1986. 28

[176] H. Velthuijsen. Distributed artificial intelligence for runtime feature-interaction resolution. In *Computer*, volume 26, pages 48–55. IEEE Computer Society, 1993. 20

[177] A. Walker. Computer, not car, is crash-prone, `http://www.autoworld.co.za/NewsArticle.aspx?Article=1908`. 1

[178] M. Weiss and B. Esfandiari. On feature interactions among web services. *International Journal of Web Services Research*, 2(4), 2005. 22, 43

[179] M. Weiss, B. Esfandiari, and Y. Luo. Towards a classification of web service feature interactions. *Computer Networks*, 51(2):359 – 381, 2007. 22

[180] M. W. Whalen. A parametric structural operational semantics for stateflow, uml statecharts, and rhapsody. Technical Report 2010-1, University of Minnesota Software Engineering Center, 2010. 49

[181] M. W. Whalen, D. D. Cofer, S. P. Miller, B. H. Krogh, and W. Storm. Integration of formal analysis into a model-based software development process. In *Prc. 12th Int'l Workshop on Industrial Critical Systems*, pages 68–84, 2007. 65, 113

[182] J. B. White. Car talk and talk and..., `http://online.wsj.com/article/SB10001424052748703778104576286631174569232.html`. 3

[183] M. Wilson, E. H. Magill, and M. Kolberg. An Online Approach for the Service Interaction Problem In Home Automation. In *IEEE Consumer Communication and Networking Conference*, pages 251– 256, 2005. 22, 43

[184] K. C. Wong, J. G. Thistle, R. P. Malhamé, and H.-H. Hoang. Supervisory control of distributed systems: Conflict resolution. *Discrete Event Dynamic Systems*, 10:131–186, 2000. 23, 128

[185] A. Wright. Automotive autonomy. *Communications of the ACM*, 54(7):16–18, 2011. 3

[186] A. Wright. Hacking cars. *Communications of the ACM*, 54(11):18–19, 2011. 3

[187] X. Wu, J. Buford, K. Dhara, V. Krishnaswamy, and M. Kolberg. Feature interactions between internet services and telecommunication services. In *International Conference on Principles, Systems and Applications of IP Telecommunications*. ACM, 2009. 21, 43

[188] J. Yang and C.-J. Seger. Introduction to generalized symbolic trajectory evaluation. In *IEEE Transactions on Very Large Scale Integration Systems*, volume 11, 2003. 163

[189] J. yin Zhang, F. chun Yang, and S. Su. Detecting feature interactions in web services with model checking techniques. *Journal of China Universities of Posts and Telecommunications*, 14(3), 2007. 22, 43

[190] T. Yoneda and T. Ohta. A formal approach for the definition and detection of feature interactions. In *Feature Interactions in Telecommunications and Software Systems V*, pages 202–216. IOS Press, 1998. 19, 42

[191] T. Yoneda and T. Ohta. The declarative language STR (state transition rule). In *FIREworks workshop*, pages 197–211. Springer, 2000. 42

[192] P. Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–30, 1993. 2

[193] I. Zibman, C. Woolf, P. O'Reilly, L. Stickland, D. Willis, and J. Visser. Minimizing feature interactions: An architecture and processing model approach. In *Feature Interactions in Telecommunications III*, pages 65–83. IOS Press, 1995. 20

[194] P. A. Zimmer and J. M. Atlee. Categorizing and prioritizing telephony features. In *Feature Interactions in Telecommunications and Software Systems VIII*, pages 327–333. IOS Press, 2005. 20