Using Formal Analysis and Search-based Techniques to Address the Assurance of Cyber-Physical Systems at the Requirements Level

By

Byron DeVries

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor Of Philosophy

2017

ABSTRACT

Using Formal Analysis and Search-based Techniques to
Address the Assurance of Cyber-Physical Systems at the Requirements Level

By

Byron DeVries

For high-assurance cyber-physical systems (CPS), such as the onboard features in modern transportation systems (e.g., automobiles, trains, and flight systems), ensuring acceptable and safe behavior is of paramount importance. Furthermore, the increasing complexity and the number of onboard features for autonomous vehicles further exacerbates the challenge of guaranteeing safe behavior. The operation of these high-assurance cyber-physical systems depends on the specification, implementation, and verification of those systems. Obstacles to assessing and ensuring assurance for cyber-physical system requirements may occur in many forms, but two significant sources of specification errors are incomplete requirements specifications and undesired feature interactions. In the case of incomplete requirements, it can be challenging to enumerate all the decomposed requirements necessary to satisfy a requirement (i.e., ensuring completeness), especially when considering different combinations of environmental conditions. A feature interaction occurs when two or more features satisfy specific properties in isolation, but no longer satisfy those properties when they are composed together. It may be necessary to analyze an exponential number of feature combinations to detect all possible interactions, resulting in a potentially exponential number of feature interaction results presented to the system developer. Furthermore, the uncertainty created by unexpected system and environmental scenarios exacerbates already difficult requirements specifications problems, many of which involve an exhaustive search for errors and their causes. That is, the exponential number of possibilities represents not only computational growth but also growth in the effort it takes the system designer to assess the results. This doctoral research tackles two key requirements assurance problems that exhibit these characteristics: requirements incompleteness and undesired feature interactions. The work explores

how formal analysis and search-based techniques can be used in a complementary and synergistic fashion to address the assurance of cyber-physical systems facing environmental and system uncertainty, both at design time and run time. Industrial applications are used to demonstrate the respective techniques.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Betty H. C. Cheng, for taking a chance on a non-traditional student who remained in the workforce. Without Dr. Cheng's understanding and support I would have never had such an opportunity, much less completed a doctoral program. I could not have asked for a better advisor, or one I could have learned more from.

I would also like to thank my committee members, especially for their flexibility given their very busy schedules. A heart felt thank you to Dr. Kalyanmoy Deb, Dr. Sandeep Kulkarni, and Dr. Philip McKinley.

Many thanks to those I met along the way, especially those from the SENS lab. A special thanks to Dr. Tony Clark, Dr. Erik Fredericks, and Dr. Jared Moore whose experiences I could learn from as they went through the process before me.

I would be remiss if I did not thank Dr. Paul Jorgensen, a true friend and mentor, for sparking my interest in software engineering in his graduate courses. Additionally, I would like to thank Dr. Greg Wolffe for allowing me to take part in my first real academic research, and Dr. Christian Trefftz for his constant positivity, supportiveness, and providing a very kind introduction to Dr. Cheng.

Most importantly, I would like to thank my wife Angela. Without her, this would have never been possible. I have leaned on her for both practical and emotional support throughout, and she gave liberally of both.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

For high-assurance cyber-physical systems, such as the onboard features in modern transportation systems (e.g., automobiles, trains, and flight systems), ensuring acceptable and safe behavior is important [65]. Furthermore, the increasing complexity and the number of onboard features for autonomous vehicles further exacerbates the challenge of guaranteeing safe behavior [4]. The operation of these high-assurance cyber-physical systems depends on the specification, implementation, and verification of those systems. Obstacles to assurance introduced early in the product life-cycle create a cascading effect that impacts downstream artifacts. Worse, system specifications are often the basis for both implementation and testing artifacts, resulting in specification errors that are not detected until negative consequences occur in deployed systems. The difficulty in identifying specification errors is compounded by both system and environmental uncertainty. The composition of independently-developed features that realize a system introduces uncertainty when considering their cumulative behavior. Similarly, the impact of *dimensionality* (i.e., each of the environmental parameters can be viewed as a dimension, thus increasing the environmental space with each added parameter) from environmental parameters introduces uncertainty when unexpected environmental scenarios occur in an exponential number of possible environmental scenarios. Consequently, automated techniques to detect counterexamples to the assurance of cyber-

physical systems within system specifications is fundamentally important, especially when system and environmental uncertainty abound.

Obstacles to assessing and ensuring assurance for cyber-physical system requirements may occur in many forms, but two significant sources of specification errors are incomplete requirements specifications and undesired feature interactions. In the case of incomplete requirements, it can be challenging to enumerate all the decomposed requirements necessary to satisfy a requirement (i.e., ensuring completeness), especially when considering different combinations of environmental conditions. A feature interaction occurs when two or more features satisfy specific properties in isolation, but no longer satisfy those properties when they are composed together [20]. It may be necessary to analyze an exponential number of feature combinations to detect all possible interactions, resulting in a potentially exponential number of feature interaction results presented to the system developer [5].

1.1 Problem Description

The uncertainty created by unexpected system and environmental scenarios exacerbates already difficult requirements specifications problems, many of which involve an exhaustive search for errors and their causes. That is, the exponential number of possibilities represents not only computational growth but also growth in the effort it takes the system designer to assess the results. This doctoral research tackles two key requirements assurance problems that exhibit these characteristics: requirements incompleteness and undesired feature interactions.

Requirements Completeness: The usefulness of a system specification depends in part on the completeness of the requirements [38]. However, enumerating all necessary requirements is difficult, especially when requirements interact with an unpredictable environment. A specification built with an idealized environmental view is incomplete if it does not include requirements to handle non-idealized behavior [38]. Often incomplete requirements are not

detected until implementation, testing, or worse, after deployment. Even when performed during requirements analysis, detecting incomplete requirements is typically an error prone, tedious, and manual task. Worse yet, a single completeness counterexample may not clearly indicate the extent that incomplete requirements impacts the system or what range of environmental scenarios are affected.

Feature Interaction: Independently-developed features often exhibit overlapping, yet conflicting behavior termed *feature interactions* [40]. Detecting unwanted feature interactions amongst even a moderate number of features can involve the analysis of an exponential number of possible interactions. To combat growth in both computational and designer effort, many researchers limit their analysis to pair-wise interactions [10, 21, 41, 59] where only combinations of two features are analyzed at a time. However, a recent study found interactions greater than pair-wise were found in every system analyzed [5]. This finding indicates that n-way interaction detection techniques are needed that do not overwhelm the system designer with results. A potentially human resource-intensive step is the subsequent effort needed by the system designer to assess each detected interaction to determine the failure caused by the feature interaction. Further, the definition of safety requirements are intended to ensure the safety properties of the system under development. However, safety properties of the system often crosscut functional and non-functional requirements. These cross-cutting concerns are dispersed throughout the requirements. This dispersion renders manual insertion of the safety concerns difficult and error prone.

Key Challenges: In order to address industry-relevant problems with respect to these two areas, we must develop techniques with the following characteristics:

- Automated analysis techniques to reason about the specification of a system,
- Techniques to explore a broad solution space given challenges posed by environmental and system uncertainty, and

- Techniques that can be applied at design time and run time.

1.2 Thesis Statement

This research defines a set of analysis methods to address the generation of counterexamples in model-based definitions of requirements. Specifically, we detect interactions and incomplete requirement decompositions in hierarchical requirements models.

Thesis Statement: Formal analysis and search-based techniques can be used in a complementary and synergistic fashion to address the automated detection and counterexample generation of incomplete requirements and n-way feature interactions in cyber-physical systems facing environmental and system uncertainty.

1.3 Research Contributions

This dissertation research focuses on identifying solutions to two key challenges: identifying incomplete requirements and undesired feature interactions. The research contributions of this work are as follows.

1. **Requirements Incompleteness**: Detect incomplete requirements decomposition in hierarchical requirements models. We illustrate our approach by analyzing a requirements model of an industry-based automotive adaptive cruise control system. We describe the following techniques that we developed to address this problem:

   (a) *Symbolic analysis* to formally define requirements incompleteness and guarantee detection of requirements completeness counterexamples if they exist [37].

   (b) *Evolutionary computation and symbolic analysis* to detect a representative range of requirements completeness counterexamples [38].

   (c) *Run-time analysis* to detect incomplete requirements at run time [39].

2. **Interaction Detection**: Detect unwanted n-way feature interactions and determining their causes at the requirements level. Unlike previous n-way feature interaction detection approaches that attempt to enumerate every set of interacting features, our approach analyzes each feature for its ability to *cause* an interaction with other features, thus reducing designer assessment effort to be linear with respect to the number of features. We illustrate our approach by applying our approach to an industry-based automotive braking system comprising multiple subsystems. We describe the following techniques that we have developed for this dissertation:

   (a) *Symbolic analysis* to formally analyze feature interactions and guarantee detection of interactions if they exist [40].

   (b) *Evolutionary computation and symbolic analysis* to detect a representative range of scenarios where a given feature interaction manifests [40].

   (c) *Run-time analysis* to detect feature interactions at run time.

3. **Non-functional Interaction Detection**: Detect feature interactions that include safety and/or performance non-functional requirements and may include, but are not limited to include, functional requirements. We describe the following techniques that we have developed for this dissertation:

   (a) *Symbolic analysis* to formally analyze non-functional feature interactions and guarantee detection of interactions if they exist.

   (b) *Evolutionary computation and symbolic analysis* to detect a representative range of scenarios where a given non-functional feature interaction manifests.

   (c) *Run-time analysis* to detect non-functional feature interactions at run time.

This dissertation presents a framework comprising multiple automated analysis techniques that collectively identify counterexamples to assurance in the specifications of cyber-physical systems in the presence of system and environmental uncertainty. Specifically, we

analyze requirements models for the existence of counterexamples that identify requirements incompleteness and feature interactions. Throughout this dissertation, we present the theoretical foundation of each technique, a prototype implementation, and the empirical validation on an industrial-based application. The flowchart in Figure 1.1 illustrates the relation between identifying completeness counterexamples and identifying interactions in requirements models. A system designer-defined set of requirements are analyzed for incomplete requirements decomposition. If an incomplete decomposition exists, then the requirements model is redefined by the system designer based on the detected counterexample. When no additional decomposition counterexamples exist, then the requirements are analyzed for interactions. If an interaction exists, then the system designer updates the requirements model to remove the interaction and the process may be reapplied. When no additional interactions exist, then a complete and interaction free requirements model is returned.

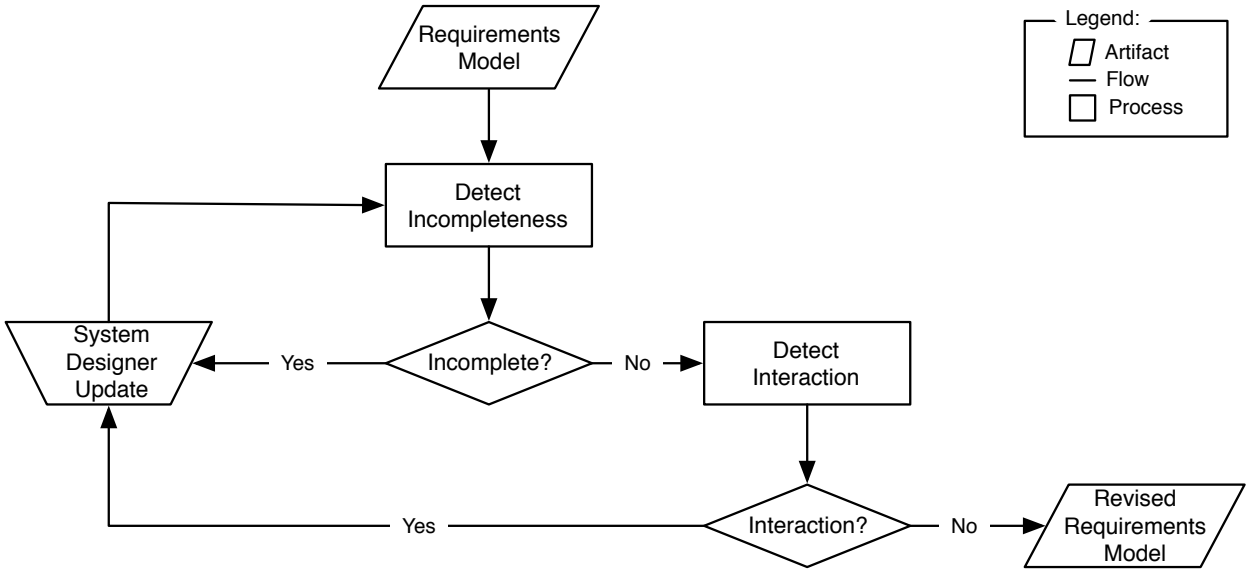


Figure 1.1: Top Level Flowchart

1.4 Organization of Dissertation

The organization of the remainder of this dissertation is as follows. Chapter 2 reviews background material on modeling techniques used in this work, feature interactions, symbolic

analysis, evolutionary computation, and two industrial case studies in automotive braking systems and adaptive cruise control. In Chapter 3, we introduce *Ares*, an approach to detect incomplete requirements decomposition in hierarchical goal models using symbolic analysis, and then apply it to an automotive adaptive cruise control case study. Chapter 4 presents *Ares*-EC, a process of combining both symbolic and evolutionary computation to detect wider ranges of requirements completeness counterexamples when applied to the same case study. Next, in Chapter 5 we describe *Lykus*, a technique of generating run-time detection code for incomplete and inconsistent requirements. We then move to feature interaction detection. In Chapter 6, we describe *Phorcys*, an approach to detect feature interactions and identify their causes; and we illustrate the approach using the automotive braking system case study. Chapter 7 presents *Phorcys*-EC, a process of combining both symbolic and evolutionary computation to detect wider ranges of feature interaction counterexamples in both the feature space and environmental scenario space. Next, in Chapter 8, we describe *Thoosa*, a process of generating run-time detection code for the features that fail due to a feature interaction. Chapter 9 describes *Soter*, a process for detecting non-functional interactions across both non-functional and functional features, including both safety and performance non-functional requirements. Related work is included in each of the individual chapters. Finally, Chapter 10 presents a summary of the contributions of this dissertation and overviews future investigations.

Chapter 2

Background

This chapter provides background information for the dissertation. An overview of goal modeling is provided followed by information on utility functions. Symbolic analysis is covered along with evolutionary computation. Finally, the two industry applications used for this research, automotive braking systems and adaptive cruise control systems, are described.

2.1 Goal Models

While the concepts of this dissertation are applicable to any hierarchical requirements modeling framework, including i^* [96], KAOS goal modeling [87], or simply hierarchical requirements modeling [82], the case studies in this dissertation make use of the KAOS goal modeling notation.[1] True of all hierarchical requirements modeling approaches, requirements are specified by decomposed requirements that are necessary to satisfy their parent requirement. Decomposition continues until some termination criteria is met.

KAOS goal modeling realizes Goal-Oriented Requirements Engineering (GORE) via a graph-based decomposition of high-level goals and objectives that make up the system-to-be. High-level objectives are decomposed into subsequently finer-grained goals and ultimately into system requirements and environmental expectations, or prerequisites satisfied by agents

[1] For this work we do not use the KAOS formal decomposition patterns.

of the environment. Decomposition within KAOS goal modeling allows for both AND and OR decompositions [87]. The satisfaction of an AND-decomposed goal only occurs if all of its aggregate decomposed goals are satisfied. In contrast, the satisfaction of an OR-decomposed goal occurs if any of its aggregate decomposed goals are satisfied. Decomposition terminates when a system requirement can be satisfied by an agent or an environmental expectation can be measured by an agent of the environment [87].

Goals are classified as either functional or non-functional. A functional goal defines a service, or element of functionality, of the system-to-be (e.g., 'stop the car'). Non-functional goals, in contrast, define a constraint on the services that a functional goal provides (e.g., 'stop the car quickly'). Functional goals may be further refined into either invariant or non-invariant goals, denoted '*Maintain*' (e.g., Maintain(Current Speed)) and '*Achieve*,' (e.g., Achieve(Higher Speed)) respectively. Decomposition is terminated when a requirement or expectation can be fully satisfied by a single agent [48]. Since the methods presented are intended to be generally applicable to hierarchical goal and requirements modeling, we refer to goals and requirements interchangeably.

Typically, requirements incompleteness is due to unexpected scenarios for which the system designer did not anticipate and are unsatisfied for the high-level objectives of the system-to-be, more detailed, decomposed requirements do not imply their parent requirement. Formally, when the decomposed requirements $(R_1, R_2, R_3, R_{...}, R_n)$, as well as the domain properties and assumptions, Dom (e.g., an increase in throttle causes an increase in speed), of the parent requirement (R) are satisfied, then the parent requirement is satisfied, as indicated by the entailment operator (i.e., $\vDash$) in the following expression [87]:

$$\{R_1, R_2, R_3, R_{...}, R_n, Dom\} \vDash R. \tag{2.1}$$

If a requirement cannot be discharged or handled by a single agent, then it must be decomposed into additional aggregate requirements that collectively can be combined to satisfy

the parent requirement. A requirement may even be decomposed into a single decomposed requirement if the parent requirement cannot be satisfied by a single agent of the system. A set of aggregate decomposed requirements is one of potentially numerous possible decompositions, and thereby constrains the solution for a given (parent) requirement to the behavior specified by the aggregate decomposed requirements. Necessarily, the parent requirement may be satisfied for any complete decomposition, but not all decompositions are necessarily equivalent with respect to the behavior of any other complete decomposition.

In this dissertation, goal and requirement model labels are in `bold courier` font, while variable names, goal and requirement text, and emphasis are indicated by *italics*.

2.2 Utility Functions

Utility functions [89] have been used for run-time monitoring [45, 47] to assess the satisfaction of requirements. Satisfaction, while typically specified as a Boolean expression, may also be represented as a degree of satisfaction called *satisficement* [92]. Previously, Athena [75] was developed to automatically generate utility functions from specific properties of the environment (**ENV**, **MON**, **REL**), where:

- **ENV** represents environmental properties related to the satisfaction of the goal that may or may not be directly observable (e.g., the expected speed of a vehicle at a future time),
- **MON** represents monitors (e.g., agents and sensors) in the system that are able to monitor specific values (e.g., a GPS speed sensor that measures the speed of a vehicle at the current time), and
- **REL** represents relationships between the monitors and environmental properties that relate to the satisfaction or satisficement of goals (e.g., relating expected future vehicle speed and current vehicle speed to measure the satisficement of a requirement to increase speed).

Athena uses the **REL** properties to compute the degree of satisficement that the utility function returns as either state-, metric-, or fuzzy-logic based satisficement results, represented as Boolean, numeric, or functions of real values, respectively. Utility functions encode domain properties and assumptions (*Dom*) that are present in the **ENV**, **MON**, and **REL** that describe the system. The domain properties are manually specified by the requirements engineer [24]. While the addition of utility functions adds additional properties for each requirement that must be documented, in cases where utility functions are already used to assess run-time satisfaction, the utility functions can be used with no additional documentation required.

2.3 Symbolic Analysis

Satisfiability Modulo Theories (SMT) solvers solve constraints defined in either SMT-LIB [76] or SMT-LIB version 2 [12] standards [27]. SMT solvers are a collection of decidable theories, represented as decision procedures, and a SAT solver, all of which are applied via strategies to solve a diverse range of constraint problems [33]. In this dissertation, we use the Microsoft Z3 SMT Solver [32], given its ubiquity and availability.

2.4 Evolutionary Computation

Genetic Algorithms are a stochastic optimization method often used to optimize complex problems [51, 70]. A population of proposed solutions is randomly initialized and assigned a *fitness* value that represents an evaluation of the performance of the individual in a population. Additional individuals are 'born' using evolutionary operators including crossover and mutation. Crossover mates two or more individuals by combining aspects of their genetic code into a new individual. Mutation modifies an individual randomly. Typically, a portion of the new and existing individuals are kept for the next generation. The process of simulated evolution continues until some upper bound, often a number of generations, is reached.

2.5 Industrial Applications

In this dissertation, we make use of two industrial systems within the automotive domain: braking systems and adaptive cruise control. A general overview of each of these systems is described in this section.

2.5.1 Automotive Braking Systems

Braking systems, such as the one considered in this dissertation, while often viewed as a single subsystem within a vehicle, in fact, comprise two categories of subsystems: 1) control subsystems that *command* a specific brake force and 2) actuator subsystems that *apply* the commanded brake force. For the braking system used in this dissertation, the two subsystems that *apply* a specific brake force are *standard force braking* (i.e., physical hydraulic force applied to brake rotors in disk brakes to cause friction that slows the turning speed of the wheels) and *regenerative braking* in hybrid vehicles (i.e., use of electric generators to reduce speed and store energy). The two subsystems that can *command* the commanded brake force, thus providing active braking, are *continuous braking* (i.e., use of electronic-commanded signals to braking subsystems rather than physical hydraulic force) and *anti-lock braking* (i.e., intermittent application of braking to reduce wheel lockup). Braking is performed by the friction provided by electric generators in regenerative braking, by the friction created by standard disk brakes, or a combination of both. Regenerative braking only provides limited stopping force, thus necessitating standard force braking for larger commanded brake forces. Figure 2.1 provides a graphical overview indicating the brake force from the brake pedal distribution to both the continuous and anti-lock braking features. Both the continuous and anti-lock braking features distribute a commanded brake force to the brake application features (i.e., standard and regenerative force). We differentiate between the features that apply brake force to those that command brake force as a design decision to promote extensibility. A commanded brake force, in the case of larger onboard

systems, may not even be directly tied to the brake force read from a pedal. For example, an adaptive cruise control system may employ brake force to prevent collision with a target car (i.e., car in front of current vehicle). Many cars also use the braking system to allow for traction control on snowy or icy roads, by applying braking to only the wheels that spin out of control.

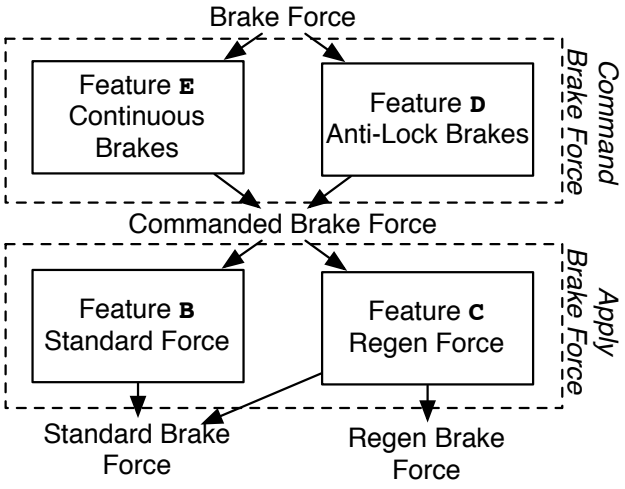


Figure 2.1: Braking System Overview

Onboard vehicle systems, including the braking system, may be viewed as individual features. Similarly, the subsystems of a given system may also be viewed as individual features depending on the level of subsystem decomposition. Herein, we use the term "feature" to refer to the observable services and functionality provided by a given "subsystem" [10], where our analysis focus is at the requirements level of a given feature (i.e., observable functionality). That is, each feature may have multiple requirements and is viewed as a subsystem. For our braking system, we have four subsystems, each of which is considered a feature: standard force braking, regenerative force braking, continuous braking, and anti-lock braking.

2.5.2 Automotive Adaptive Cruise Control

An Adaptive Cruise Control (ACC) system uses radar to adjust vehicle speed ensuring a safe following distance from the car ahead while maintaining as close to the desired speed as

possible. An ACC can be viewed in four parts: cruise control modes (e.g., on, off), increasing speed, lowering speed, and maintaining speed. Speed is increased or lowered to match the desired speed, however speed may also be lowered if there is not a safe distance to the target car (i.e., car immediately in front). Similarly, the speed will not continue to increase if the safe distance is violated. The speed is maintained if both the desired speed is met and the target car is a safe distance.

Chapter 3

Detecting Incomplete Requirements: Using Symbolic Analysis

The usefulness of a system specification depends in part on the completeness of the requirements. However, enumerating all necessary requirements is difficult, especially when requirements interact with an unpredictable environment. A specification built with an idealized environmental view is incomplete if it does not include requirements to handle non-idealized behavior. Often incomplete requirements are not detected until implementation, testing, or worse, after deployment. Even when performed during requirements analysis, detecting incomplete requirements is typically an error prone, tedious, and manual task. This chapter introduces *Ares*, a design-time approach for detecting incomplete requirements decomposition using symbolic analysis of hierarchical requirements models. We illustrate our approach by applying *Ares* to a requirements model of an industry-based automotive adaptive cruise control system. *Ares* is able to automatically detect specific instances of incomplete requirements decompositions at design-time, many of which are subtle and would be difficult to detect, either manually or with testing.

3.1 Introduction

Despite the best efforts and intentions of system developers, developing complete requirements is often a challenge. Exhaustively enumerating all cases sufficient to satisfy expected functionality can be prohibitively difficult, especially when unexpected scenarios arise. Verification of decomposed requirements commonly requires domain expertise to ensure completeness. However, verification is often performed manually, at some expense, and without guarantee. This chapter presents *Ares*,[1] a symbolic analysis approach to automatically identifying counterexamples to completeness in hierarchical requirements models.

The process used to create requirements that are decomposed completely is not straightforward, and detecting incomplete requirements is still an active research question [2, 23, 46, 69, 98]. For example, a requirement for a vehicle may be to stop. In an idealized system, applying brake force (e.g., from hydraulic brakes) would be sufficient. However, in a realistic system, applying the maximum amount of brake force may not be sufficient in the presence of the maximum amount of throttle. In inclement weather, brake force may not be sufficient without anti-lock brakes. Enumerating all decomposed requirements necessary to satisfy a requirement (e.g., to stop) can be challenging especially when considering different combinations of environmental conditions. While formal methods exist to decompose goals and requirements with guaranteed completeness [31], they are not widely used in practice, and system designers are limited to only specific formal decomposition rules. Methods also exist for identifying counterexamples to completeness [2], but are limited to completeness with respect to specific domain properties (i.e., the set of system and environmental variables and states) rather than with respect to decomposition and require manual review for relevance and applicability. Currently, no methods exist that automate the detection of incomplete requirements decomposition without imposing restrictions on how requirements are decomposed or described.

This chapter describes *Ares*, a symbolic analysis approach to automatically identify

[1] *Ares* is the Greek god of war, especially the untamed aspects of war.

environmental configurations where completeness properties are violated in a hierarchical requirements model. Hierarchical requirements satisfaction can be assessed in two ways: its individual satisfaction, or the satisfaction of a requirement's aggregate decomposed requirements (i.e., children or sub-requirements). Given complete decomposition, a requirement should be satisfied whenever its aggregate decomposed requirements are satisfied. For example, given a requirement to stop and a decomposed requirement to brake, the requirement to stop should be satisfied if the aggregated decomposed requirements are satisfied (i.e., to brake). However, if the throttle can overwhelm the brakes, then the requirement to stop may not actually be satisfied. An additional aggregate decomposed requirement is needed when a requirement is not satisfied even though its aggregate decomposed requirements are satisfied. Since the requirement to stop is unsatisfied due to the force created by the throttle, one possible solution would be AND decomposed into two requirements: a requirement to brake and a requirement to use no throttle (e.g., remove foot from gas pedal), both of which are necessary for the satisfaction of the requirement to stop. *Ares* identifies incomplete decompositions in the form of counterexamples that are summarized for the system designer to revise the requirements accordingly.

Ares analyzes individual requirements within a hierarchical requirements model using expressions from utility functions [75] that represent requirements satisfaction. Utility functions translate system-monitored data to scalar values that scale proportionally to the degree a requirement is satisfied. Utility functions are typically used to monitor requirements at run-time [75]. Rather than using the utility functions for run-time monitoring, *Ares* uses the expressions themselves for symbolic analysis. For each requirement's decomposition, *Ares* identifies, via symbolic analysis of the range of possible requirement variables values, whether are any environmental conditions exist that cause a requirement to be unsatisfied while its aggregate decomposed requirements are satisfied. Counterexamples are identified for an industry-based requirements model for an adaptive cruise control system, and incomplete requirements decompositions are summarized and presented to the system designer.

The contributions of this chapter are as follows:

- We introduce *Ares*, a design-time, symbolic analysis approach to automatically detect incomplete decomposition in hierarchical requirements models.
- We present a prototype implementation of the *Ares* symbolic analysis approach.
- We demonstrate the applicability of *Ares* on an industry-based automotive example, an adaptive cruise control system.

The remainder of this chapter is organized into the following sections. Section 3.2 provides an overview of the input model. Section 3.3 details the approach. Section 3.4 describes the results of a case study, and Section 3.5 details related work. Finally, Section 3.6 summarizes the work.

3.2 Adaptive Cruise Control Input Model

The Adaptive Cruise Control system is a cruise control system with radar that adjusts the vehicle's speed autonomously to ensure a safe distance with a target vehicle, while maintaining a desired speed. Figure 3.1 is a KAOS goal model for the ACC system. Keywords *Maintain* and *Achieve* are shortened to *M* and *A*, respectively. Table 3.1 identifies the agents referenced in Figure 3.1. The ACC model defined here was manually developed by the authors in conjunction with automotive industrial practitioners and is intended to be representative of a realistic example of industrial requirements engineering artifacts.[2]

In Figure 3.1, the goals that begin with prefix '`A.`' (top left) contain the overall hierarchy of ACC modes and also includes manual throttle and manual brake response requirements. Goals `A.6` through `A.14` refer to the state of the value of their respective agents. For example, `A.6` indicates that its agent, *Cruise Switch Sensor*, is *Off*. All of `A.6` through `A.14` are

[2]Conversations with automotive industrial practitioners included reviewing our goal modeling approach to automotive systems, including review of the specific ACC requirements model included here, over the course of several meetings.

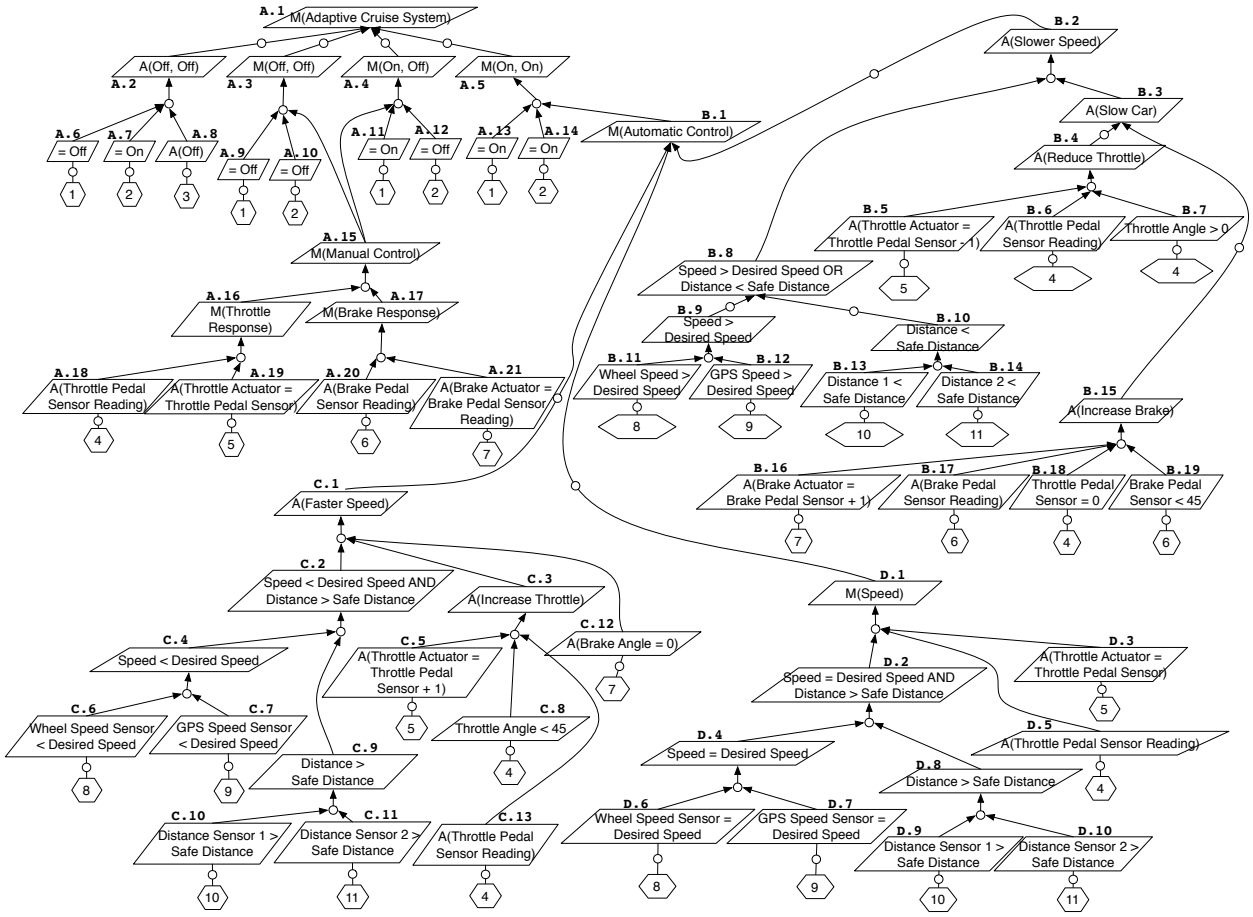


Figure 3.1: Adaptive Cruise Control Goal Model

expectations (i.e., conditions to be handled by the environment or outside the control of the system), except for `A.8` which is a system goal to turn *Off* the *Cruise Active Switch*. The top-level goal, `A.1`, is decomposed into four categories of functionality (`A.2`, `A.3`, `A.4`, and `A.5`), each of these indicates a pair of values from *Cruise Switch Sensor* and *Cruise Active Sensor*, respectively. This functionality is dependent on the values of the *Cruise Switch Sensor* and *Cruise Active Sensor*, which are either maintained or achieved. In goals `A.3` and `A.4`, where the *Cruise Active Sensor* is *Off*, the throttle is controlled manually. If the *Cruise Active Sensor* is *On*, but the *Cruise Switch Sensor* is *Off* (i.e., goal `A.2`), then the *Cruise Active Switch* is turned off via an *Achieve* goal (i.e., `A.8`). In goal `A.5` where both *Cruise Active Sensor* and *Cruise Switch Sensor* are both *On*, the throttle is automatically controlled. The remainder of the goal model, where goals are prefixed with '`B.`', '`C.`', or '`D.`' represents

Table 3.1: Agents used in Goal Model

#	**Agent (Sensor / Actuator)**
1	Cruise Switch Sensor
2	Cruise Active Sensor
3	Cruise Active Switch
4	Throttle Pedal Sensor
5	Throttle Actuator
6	Brake Pedal Sensor
7	Brake Actuator
8	Wheel Speed Sensor
9	GPS Speed Sensor
10	Distance Sensor 1
11	Distance Sensor 2

portions of the automatic control for throttle and braking by changing or maintaining the speed, detailed as follows:

Goal `B.2` : '*A(Slower Speed)*' decreases the speed by reducing throttle via goal `B.4` or increase braking via goal `B.15` when the *Speed* is greater than the *Desired Speed* (goal `B.9`) or the *Distance* measured is less than the defined *Safe Distance* (goal `B.10`).

Goal `C.1` : '*A(Faster Speed)*' increases the speed by increasing the throttle via goal `C.3` while ensuring braking is minimized via goal `C.12` when the *Speed* is less than the *Desired Speed* (goal `C.4`) and the *Distance* measured is greater than the defined *Safe Distance* (goal `C.9`).

Goal `D.1` : '*M(Speed)*' maintains the speed by maintaining the current throttle position via goal `D.3` when the *Speed* is equal to the *Desired Speed* and the *Distance* measured is greater than the defined *Safe Distance* (goal `D.8`).

The **ENV**, **MON**, and **REL** properties for the goal model in Figure 3.1 are defined in Table 3.2. Environmental properties (**ENV**) that are not directly observable via agents or sensors are calculated using a combination of monitors (**MON**) and a relationship (**REL**) to compute their respective values. The **ENV** and **MON** properties are only listed when used in the relationship (**REL**). For example, goals `A.1` through `A.21` can be assessed directly with

monitors (**MON**) without directly accessing environmental conditions. The **ENV**, **MON**, and **REL** properties are generated manually by the requirements engineer. Table 3.3 defines the range and unit for each of the variables defined in Table 3.2.

3.3 Symbolic Analysis Approach

Ares is a method for symbolically analyzing hierarchical requirements models for completeness. Issues with completeness are detectable when the measured satisficement of a requirement is not logically implied by its decomposed requirements. For each decomposed goal, symbolic analysis is used to evaluate the entire applicable range of all environmental configurations and system variables referenced in the parent goal and its decomposed aggregate requirements. Unlike variable testing with a finite number of concrete instantiations where instances of completeness counterexamples may be missed, all possibilities can be simultaneously analyzed symbolically. As shown in Expression (2.1), a requirement is satisfied if its aggregate decomposed requirements are complete and satisfied over the domain properties and assumptions (Dom). For AND-decomposition, the following equation must be true for complete decompositions:

$$(R_1 \wedge ... \wedge R_i \wedge ... \wedge R_n) \wedge Dom \implies R. \tag{3.1}$$

For OR-decomposition, the following equation must be true for complete decompositions:

$$(R_1 \vee ... \vee R_i \vee ... \vee R_n) \wedge Dom \implies R. \tag{3.2}$$

In the case of AND- and OR-decompositions, the set of requirements $(R_1, R_2, R_3, R_{...}, R_n)$ composed by the decomposition operator (OR or AND) implies the parent requirement (R), assuming the domain properties and assumptions (Dom) hold. When using utility functions to assess satisficement, Boolean logic is not sufficient due to

Table 3.2: **ENV**, **MON**, and **REL** Properties

Goal	ENV	MON	REL
A.1		*Cruise Switch Sensor*, *Cruise Active Sensor*	*true*
A.2		*Cruise Switch Sensor*, *Cruise Active Sensor*	*Cruise Switch Sensor* == *Off* $\wedge$ *Cruise Active Sensor* == *On*
A.3		*Cruise Switch Sensor*, *Cruise Active Sensor*	*Cruise Switch Sensor* == *Off* $\wedge$ *Cruise Active Sensor* == *Off*
A.4		*Cruise Switch Sensor*, *Cruise Active Sensor*	*Cruise Switch Sensor* == *On* $\wedge$ *Cruise Active Sensor* == *Off*
A.5		*Cruise Switch Sensor*, *Cruise Active Sensor*	*Cruise Switch Sensor* == *On* $\wedge$ *Cruise Active Sensor* == *On*
A.6		*Cruise Switch Sensor*	*Cruise Switch Sensor* == *Off*
A.7		*Cruise Active Sensor*	*Cruise Active Sensor* == *On*
A.8		*Cruise Active Switch*	*Cruise Active Switch* == *Off*
A.9		*Cruise Switch Sensor*	*Cruise Switch Sensor* == *Off*
A.10		*Cruise Active Sensor*	*Cruise Active Sensor* == *Off*
A.11		*Cruise Switch Sensor*	*Cruise Switch Sensor* == *On*
A.12		*Cruise Active Sensor*	*Cruise Active Sensor* == *Off*
A.13		*Cruise Switch Sensor*	*Cruise Switch Sensor* == *On*
A.14		*Cruise Active Sensor*	*Cruise Active Sensor* == *On*
A.15		*Throttle Pedal Sensor*, *Throttle Actuator*, *Brake Pedal Sensor*, *Brake Actuator*	*Throttle Pedal Sensor* == *Throttle Actuator* $\wedge$ *Brake Pedal Sensor* == *Brake Actuator*
A.16		*Throttle Pedal Sensor*, *Throttle Actuator*	*Throttle Pedal Sensor* == *Throttle Actuator*
A.17		*Brake Pedal Sensor*, *Brake Actuator*	*Brake Pedal Sensor* == *Brake Actuator*
A.18			*true*
A.19		*Throttle Pedal Sensor*, *Throttle Actuator*	*Throttle Pedal Sensor* == *Throttle Actuator*
A.20			*true*
A.21		*Brake Pedal Sensor*, *Brake Actuator*	*Brake Pedal Sensor* == *Brake Actuator*
B.1	$Speed_t$, $Speed_{t+1}$, *Distance*	*Safe Distance*	$Speed_t >= Speed_{t+1} \vee$ *Distance* < *Safe Distance*
B.2	$Speed_t$, $Speed_{t+1}$		$Speed_t > Speed_{t+1} \vee (Speed_t == MIN \wedge Speed_{t+1} == MIN)$
B.3	$Speed_t$, $Speed_{t+1}$		$Speed_t > Speed_{t+1} \vee (Speed_t == MIN \wedge Speed_{t+1} == MIN)$
B.4		*Throttle Actuator*, *Throttle Pedal Sensor*	*Throttle Actuator* < *Throttle Pedal Sensor*
B.5		*Throttle Actuator*, *Throttle Pedal Sensor*	*Throttle Actuator* < *Throttle Pedal Sensor*
B.6			*true*
B.7		*Throttle Pedal Sensor*	*Throttle Pedal Sensor* > 0
B.8	$Speed_t$, *Distance*	*Desired Speed*, *Safe Distance*	$Speed_t >$ *Desired Speed* $\vee$ *Distance* < *Safe Distance*
B.9	$Speed_t$	*Desired Speed*	$Speed_t >$ *Desired Speed*
B.10	*Distance*	*Safe Distance*	*Distance* < *Safe Distance*
B.11		*Wheel Speed Sensor*, *Desired Speed*	*Wheel Speed Sensor* > *Desired Speed*
B.12		*GPS Speed Sensor*, *Desired Speed*	*GPS Speed Sensor* > *Desired Speed*
B.13		*Distance Sensor 1*, *Safe Distance*	*Distance Sensor 1* < *Safe Distance*
B.14		*Distance Sensor 2*, *Safe Distance*	*Distance Sensor 2* < *Safe Distance*
B.15		*Brake Actuator*, *Brake Pedal Sensor*	*Brake Actuator* > *Brake Pedal Sensor*
B.16		*Brake Actuator*, *Brake Pedal Sensor*	*Brake Actuator* > *Brake Pedal Sensor*
B.17			*true*
B.18		*Throttle Pedal Sensor*	*Throttle Pedal Sensor* == MIN
B.19		*Brake Pedal Sensor*	*Brake Pedal Sensor* < MAX
C.1	$Speed_t$, $Speed_{t+1}$		$Speed_t < Speed_{t+1}$
C.2	$Speed_t$, *Distance*	*Desired Speed*, *Safe Distance*	$Speed_t <$ *Desired Speed* $\wedge$ *Distance* > *Safe Distance*
C.3		*Throttle Actuator*, *Throttle Pedal Sensor*	*Throttle Actuator* > *Throttle Pedal Sensor*
C.4	$Speed_t$	*Desired Speed*	$Speed_t <$ *Desired Speed*
C.5		*Throttle Actuator*, *Throttle Pedal Sensor*	*Throttle Actuator* > *Throttle Pedal Sensor*
C.6		*Wheel Speed Sensor*, *Desired Speed*	*Wheel Speed Sensor* < *Desired Speed*
C.7		*GPS Speed Sensor*, *Desired Speed*	*GPS Speed Sensor* < *Desired Speed*
C.8		*Throttle Pedal Sensor*	*Throttle Pedal Sensor* < MAX
C.9	*Distance*	*Safe Distance*	*Distance* > *Safe Distance*
C.10		*Distance Sensor 1*, *Safe Distance*	*Distance Sensor 1* > *Safe Distance*
C.11		*Distance Sensor 2*, *Safe Distance*	*Distance Sensor 2* > *Safe Distance*
C.12		*Brake Actuator*	*Brake Actuator* == MIN
C.13			*true*
D.1	$Speed_t$, $Speed_{t+1}$		$Speed_t == Speed_{t+1}$
D.2	$Speed_t$, *Distance*	*Desired Speed*, *Safe Distance*	$Speed_t ==$ *Desired Speed* $\wedge$ *Distance* > *Safe Distance*
D.3		*Throttle Actuator*, *Throttle Pedal Sensor*	*Throttle Actuator* == *Throttle Pedal Sensor*
D.4	$Speed_t$	*Desired Speed*	$Speed_t ==$ *Desired Speed*
D.5			*true*
D.6		*Wheel Speed Sensor*, *Desired Speed*	*Wheel Speed Sensor* == *Desired Speed*
D.7		*GPS Speed Sensor*, *Desired Speed*	*GPS Speed Sensor* == *Desired Speed*
D.8	*Distance*	*Safe Distance*	*Distance* > *Safe Distance*
D.9		*Distance Sensor 1*, *Safe Distance*	*Distance Sensor 1* > *Safe Distance*
D.10		*Distance Sensor 2*, *Safe Distance*	*Distance Sensor 2* > *Safe Distance*
$Speed_t$		*Wheel Speed Sensor*, *GPS Speed Sensor*	*Wheel Speed Sensor* $\vee$ *GPS Speed Sensor*
$Speed_t$		*Throttle Pedal Sensor*, *Brake Pedal Sensor*	$max($*MIN*, *Throttle Pedal Sensor* $-$ *Brake Pedal Sensor*$)$
$Speed_{t+1}$		*Throttle Actuator*, *Brake Actuator*	$max($*MIN*, *Throttle Actuator* $-$ *Brake Actuator*$)$
Distance		*Distance Sensor 1*, *Distance Sensor 2*	*Distance Sensor 1* $\vee$ *Distance Sensor 2*

Table 3.3: Units and Scaling for Variables in Table 3.2

Variable	**Min**	**Max**	**Unit**
$Speed_t$	0.0	100.0	MPH
$Speed_{t+1}$	0.0	100.0	MPH
Distance	0.0	50.0	Feet
Desired Speed	0.0	100.0	MPH
Safe Distance	0.0	50.0	Feet
Throttle Actuator	0.0	100.0	%
Throttle Pedal Sensor	0.0	100.0	%
Brake Actuator	0.0	100.0	%
Brake Pedal Sensor	0.0	100.0	%
Distance Sensor 1	0.0	50.0	Feet
Distance Sensor 2	0.0	50.0	Feet
Wheel Speed Sensor	0.0	100.0	MPH
GPS Speed Sensor	0.0	100.0	MPH
Cruise Switch Sensor	*Off*	*On*	Boolean
Cruise Active Sensor	*Off*	*On*	Boolean
Cruise Active Switch	*Off*	*On*	Boolean

the real-valued results. Therefore, the minimum satisficement of the decomposed goals is used for AND-decomposition and maximum satisficement of the decomposed goals is used for OR-decomposition [75]. The measure of decompositional completeness for assessing satisficement for Expression (3.1) (i.e., AND decomposition) is captured in Expression (3.3):

$$
\begin{aligned}
&min(Satisficement(R_1), \\
&\quad ..., \\
&\quad Satisficement(R_i), \\
&\quad ..., \\
&\quad Satisficement(R_n)) \implies Satisficement(R),
\end{aligned}
\tag{3.3}
$$

where the *Satisficement* function is the corresponding utility function (i.e., measure of the range of satisfaction from unsatisfied at 0.0 to satisfied at 1.0) for the respective requirement parameter. For example, *Satisficement*(`B.2`) is the **REL** expression from row `B.2` in Table 3.2. Similarly, the measure of decompositional completeness for assessing satisficement

for Expression (3.2) (i.e., OR decomposition) is captured in Expression (3.4):

$$
\begin{aligned}
&max(Satisficement(R_1), \\
&\quad ..., \\
&\quad Satisficement(R_i), \\
&\quad ..., \\
&\quad Satisficement(R_n)) \implies Satisficement(R)
\end{aligned}
\tag{3.4}
$$

In both Expressions (3.3) and (3.4), the utility functions (i.e., *Satisficement*) encode the applicable domain properties and assumptions (*Dom*) via the **ENV**, **MON**, and **REL** properties as specified in Table 3.2. Specifically, the **REL** properties are used to describe the degree of satisficement that the utility function returns as either state- or metric-logic based satisficement results, that is, Boolean or real-value, respectively.

The remainder of this section describes the *Ares* process for detecting incomplete requirements decompositions by finding counterexamples to Expressions (3.3) and (3.4).

3.3.1 Ares Process

Figure 3.2 overviews the *Ares* process in a Data Flow Diagram (DFD). The circles represent processing elements. The parallel horizontal bars represent persistent data. The labeled arrows represent data flows. The boxes represent external entities. *Ares* makes use of Athena [75] to generate utility functions from a goal model (e.g., Figure 3.1) and **ENV**, **MON**, and **REL** properties (e.g., Table 3.2). Step 1 of Figure 3.2 accepts a goal model and the Athena-derived utility functions to produce a set of satisficement functions based on the defined relationships for all the hierarchically decomposed requirements. Step 2 applies the symbolic analysis to the satisficement expressions to identify decomposition counterexamples.

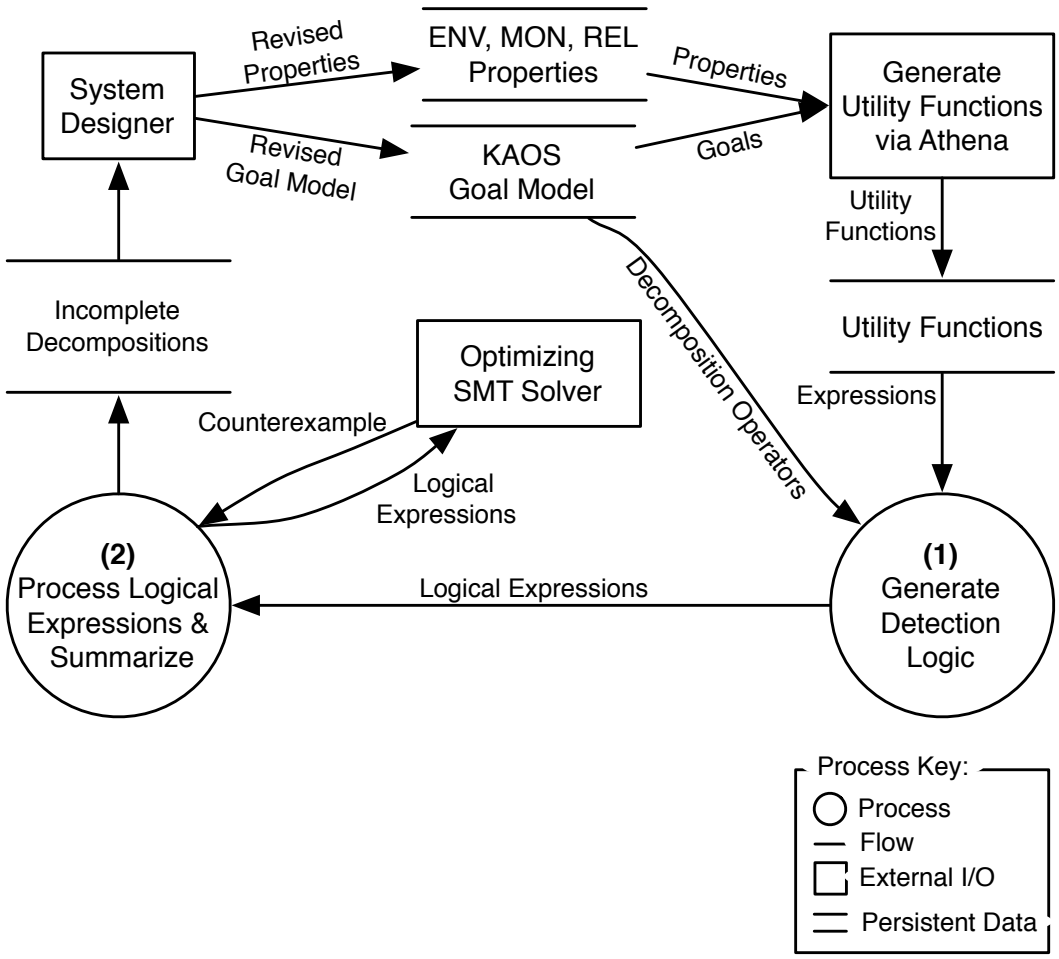


Figure 3.2: *Ares* Data Flow Diagram

DFD Step 1: Generate Detection Logic

Ares uses the expressions generated by Athena, as well as the decomposition operators (either AND or OR) to build an expression that measures the numerical difference in the satisficement of a requirement and its decomposed requirements. Any requirement that does not measure satisficement, but only discrete satisfaction (i.e., satisfied or not-satisfied) is mapped to a satisficement of 1.0 for 'satisfied' and 0.0 for 'not satisfied.' The aggregate decomposed requirements use the respective decomposition operator to define their collective satisficement. For example, the satisficement of any requirement can be measured via its utility function, therefore the satisficement of `B.4` is measured by Expression (3.5) based on the **REL** property specified by the requirements engineer in Table 3.2 and taken from the

generated utility function.

$$Satisficement(\texttt{B.4}) = Throttle\ Actuator < Throttle\ Pedal\ Sensor \tag{3.5}$$

The satisficement of a set of aggregate decomposed goals is dependent on the decomposition operator, where OR-decomposed requirements are satisficed, as a group, at the maximum any individual requirement is satisficed (i.e., the antecedent of the implication in Expression (3.4)). For example, the requirements OR-decomposed from `B.8` are:

$$Satisficement_{Decomp}(\texttt{B.8}) = max(Satisficement(\texttt{B.9}), Satisficement(\texttt{B.10})). \tag{3.6}$$

In contrast, requirements decomposed via an AND operator are satisficed as a group at the minimum that any individual requirement is satisficed (i.e., the antecedent of the implication in Expression (3.3)). For example, the satisficement of requirements AND-decomposed from `B.4` are:

$$Satisficement_{Decomp}(\texttt{B.4}) = min(Satisficement(\texttt{B.5}), Satisficement(\texttt{B.6}), Satisficement(\texttt{B.7})). \tag{3.7}$$

In order to detect completeness decomposition errors, a counterexample must be found where the satisficement of the aggregate decomposed requirements do not imply the satisficement of the parent requirement. For example, analysis of Expression (3.8) is used to determine if goal `B.4` has a completeness counterexample.

$$Counterexample_{\texttt{B.4}} = \neg(Satisficement_{Decomp}(\texttt{B.4}) \implies Satisficement(\texttt{B.4})). \tag{3.8}$$

Since satisficement of the goals is measured in terms of a real value in the range of 0.0 to 1.0, maximizing the difference between the decomposed requirements satisficement and the parent requirement satisficement identifies the most significant (e.g., largest difference between decomposed and parent satisficement) completeness counterexample (in case there is more than one completeness counterexample for a given decomposed goal). For example, maximizing Expression (3.9) identifies the most significant completeness counterexample for the decomposition of `B.4`:

$$Counterexample_{\texttt{B.4}} = Satisficement_{Decomp}(\texttt{B.4}) - Satisficement(\texttt{B.4}). \tag{3.9}$$

Identifying the maximum of Expression (3.9) yields counterexamples to Expressions (3.3) and (3.4), where the parent requirement, `B.4` is satisfied while the decomposed requirements are not.

For all decomposed goals, the corresponding difference expressions (e.g., Expression (3.9) for `B.4`) are encoded in SMT version 2 [12], along with additional optimization commands (e.g., *maximize*) [14, 15] that maximize a designated expression (e.g., Expression (3.9) for `B.4`). The encoded expressions along with the decomposition operator are output to Step 2 in Figure 3.2.

DFD Step 2: Process Logical Expressions & Summarize

The expressions generated from Step 1 are processed, along with the decomposition operator from the goal model, to generate the following values for each decomposition:

- Measures of satisficement for individual requirements using utility functions (e.g., Expression (3.5)),
- Measures of satisficement for requirements based on their decomposed requirement's utility functions (e.g., Expressions (3.6) and (3.7)), and

- Measures of the maximum numerical difference between the satisficement of a requirement and its aggregate decomposed requirements (e.g., Expression (3.9)).

Each counterexample provides the previously generated values for the decomposition along with the concrete values for environmental and system variables specific to the counterexample.

Each expression that is analyzed represents a single requirement and its aggregate decomposed requirements. To analyze the entire goal model, each decomposed requirement must be analyzed for completeness.

Counterexamples representing the maximum satisficement differences are summarized and can be used by the system designer to update the requirements model and/or **ENV**, **MON**, and **REL** properties. Possibly unreachable, but also unguarded, scenarios that indicate incomplete decomposition will be detected. Specific requirement guards (e.g., expectations) should be used to address the incomplete decomposition counterexample, due to possible changes in reachability throughout the life of the system. For example, the limitations of a vehicle engine may bound the acceleration such that an incomplete decomposition related to high acceleration is unreachable (i.e., impossible). However, if the car is modified, then the incomplete decomposition may become reachable.

Incomplete requirements may arise due to a variety of causes, including missing or superfluous requirements for either all environmental configurations or a subset of possible environmental configurations. Alternatively, the parent or decomposed requirements may simply be incorrect. Revising the goal model for each of these causes may require additional levels of decomposition to allow for alternate requirement choices for subsets of cases, the addition or removal of requirements, or correcting individual requirements.

3.3.2 Scalability and Limitations

Detection of incomplete requirements decomposition relies on the assumption that individual requirements are correctly measured for satisficement by the respective utility func-

tions. Without such an assumption, no decompositions, including complete decompositions, could be reasoned about due to possible measurement errors. The generation of utility functions requires **ENV**, **MON**, and **REL** properties for each requirement, thereby increasing the volume of information required by *Ares*. Increases in data or information required is an issue shared by techniques that focus on the relationship of requirements models and the environment [1, 64]. However, in cases where utility functions already exist (e.g., for run-time monitoring) there is no additional information overhead.

Ares processes each decomposition individually, therefore the computational cost of analyzing a goal model grows linearly with respect to the number of decompositions in the context of the worst-case single decomposition. Given that individual decompositions are typically small (e.g., not more than a dozen), *Ares* can provide scalable analysis, even for large requirements specifications (e.g., tens of thousands to hundreds of thousands of requirements). The execution time of the case study detailed in Section 3.4 is less than 3 seconds for each decomposition analysis on a 1.3 GHz Intel Core i5 processor with 4 GB of 1600 MHz DDR3 RAM.

3.4 Symbolic Analysis Case Study

This section describes the case study results of applying *Ares* to the ACC system in Figure 3.1, referred to as goal model M.

3.4.1 Incomplete AND Decomposition: Goal D.1

The ACC system defined in Figure 3.1 identifies numerous decomposed requirements. We start by describing the *Ares* processes for a single decomposed requirement, `D.1`. The decomposition of `D.1` is shown in its entirety in Figure 3.3. The goal `D.1` in goal model M is intended to specify requirements for the ACC system to maintain the current speed by controlling the accelerator. The decomposition requires all three of its decomposed re-

quirements to be satisfied (i.e., D.2, D.3, and D.5). First, in goal D.2, the environmental expectations must be met. The current speed ($Speed_t$) must match the desired speed (*Desired Speed*) set in the ACC and there must be sufficient distance to maintain the current speed safely (*Distance* > *Safe Distance*)[3]. Second, in goal D.3, the current throttle position (*Throttle Pedal Sensor*) is maintained in the expressed throttle position (*Throttle Actuator*). Third, in goal D.5, the current throttle position is read from the responsible agent (i.e., the *Throttle Pedal Sensor*).

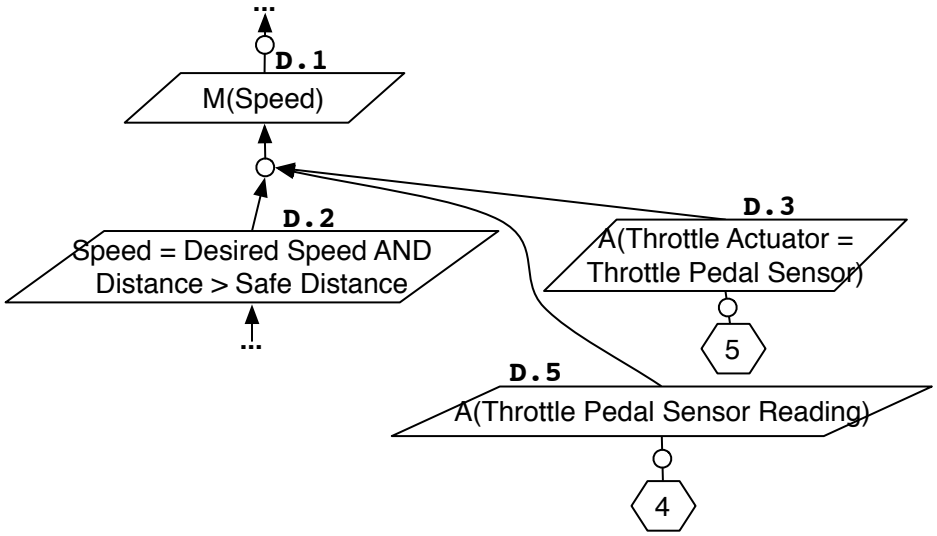


Figure 3.3: Example AND Decomposition in Goal Model M

1-M) Generate Detection Logic for Model M

We start by maximizing a version of Expression (3.9), instantiated for D.1:

$$Counterexample_{\texttt{D.1}} = Satisficement_{Decomp}(\texttt{D.1}) - Satisficement(\texttt{D.1}). \tag{3.10}$$

Expanding $Satisficement_{Decomp}$ via Expression (3.7) (i.e., the expression that defines the satisfaction of a combination of child goals) for D.1 yields the following expression, where

[3]While optimization typically requires an epsilon to be defined for less-than or greater-than, the optimization methods used in this dissertation do not require an explicitly defined epsilon.

the child goals of D.1 are D.2, D.5, and D.3:

$$\begin{aligned} Counterexample_{\texttt{D.1}} = min(&Satisficement(\texttt{D.2}), \\ &Satisficement(\texttt{D.5}), \\ &Satisficement(\texttt{D.3})) - Satisficement(\texttt{D.1}). \end{aligned} \tag{3.11}$$

Employing the utility functions defined in Table 3.2, where *true* is mapped to 1.0 and *false* is mapped to 0.0, Expression (3.12) is generated where the parent requirement is subtracted from the minimum (i.e., equivalent to an AND operation) of the decomposed requirements.

$$\begin{aligned} Counterexample_{\texttt{D.1}} = min(&Speed_t == Desired\ Speed \wedge \\ &Distance > Safe\ Distance, \\ &true, \\ &Throttle\ Actuator == Throttle\ Pedal\ Sensor) - \\ &Speed_t == Speed_{t+1} \end{aligned} \tag{3.12}$$

2-*M*) Process Logical Expressions & Summarize

The completeness counterexample expression (Expression (3.12)) is maximized, while ensuring the domain properties and assumptions (i.e., $Speed_t$, $Speed_{t+1}$, and *Distance*) in Table 3.2 are maintained, resulting in a maximized value of 1.0 (i.e., *true*). This result indicates the existence of a completeness counterexample. The counterexample provides the specific environmental configuration of variables, in both the normalized (from 0.0 to 1.0) values symbolically analyzed and their real world values (see Table 3.4). The counterexample indicates that while goal D.1 is unsatisfied, and therefore the current speed ($Speed_t$) is not maintained, all of the decomposed goals are satisfied. Therefore, it is not sufficient to maintain speed via the throttle. The utility function for goal D.1, $Speed_t == Speed_{t+1}$, indicates

that the current and next speed values will not be the same. The domain properties and assumptions of the environment for both $Speed_t$ and $Speed_{t+1}$, as documented in Table 3.2, depend on the throttle *and brake*. This finding is reasonable, as we would expect the speed of a vehicle to be impacted by the application of the brake. Clearly, the decomposition of goal `D.1` must include braking control.

Table 3.4: Counterexample Variables for `D.1` in Goal Model M

Variable or Requirement	**Normalized Value**	**Value**
$Speed_t$	0.5	50.0 MPH
$Speed_{t+1}$	0.0	0.0 MPH
Distance	0.5	25.0 Feet
Desired Speed	0.5	50.0 MPH
Safe Distance	0.0	0.0 Feet
Throttle Actuator	0.5	50.0%
Throttle Pedal Sensor	0.5	50.0%
Brake Actuator	0.75	75.0%
Brake Pedal Sensor	0.0	0.0%
Goal `D.1`	0.0	Unsatisfied
Goal `D.2`	1.0	Satisfied
Goal `D.3`	1.0	Satisfied
Goal `D.5`	1.0	Satisfied

Following the example of goals `D.3`, and `D.5`, two additional requirements are added to the decomposition in revised goal model M' for brake control. The partial goal model in Figure 3.4 shows the updated portion of goal model M' where requirements `D.New1` and `D.New2` are newly added to the decomposition of goal `D.1`. The new goals, `D.New1` and `D.New2`, set the *Brake Actuator* to the current pedal position and read the current pedal position, respectively.

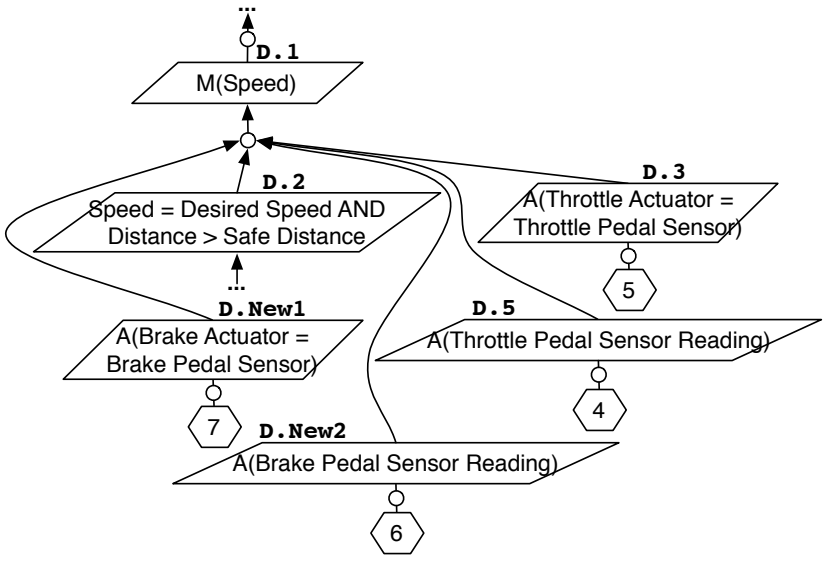


Figure 3.4: Revised AND Decomposition in Goal Model M'

1-M') Generate Detection Logic for Model M'

The associated utility functions for these new requirements model M' are:

$$Satisficement(\texttt{D.New1}) = Brake\ Actuator == Brake\ Pedal\ Sensor \tag{3.13}$$

$$Satisficement(\texttt{D.New2}) = true, \tag{3.14}$$

where $Satisficement(\texttt{D.New1})$ includes two **MON** properties: *Brake Actuator* and *Brake Pedal Sensor*. Goal `D.New2` includes no properties, only requires a sensor reading.

Inserting these additional utility functions (Expressions (3.13) and (3.14)) to those defined in Table 3.2, results in the following completeness counterexample detection expression:

$$\begin{aligned} Counterexample_{\texttt{D.1}} = min(&Speed_t == Desired\ Speed \wedge \\ &Distance > Safe\ Distance, true, \\ &Throttle\ Actuator == Throttle\ Pedal\ Sensor, \\ &Brake\ Actuator == Brake\ Pedal\ Sensor, \\ &true) - Speed_t == Speed_{t+1}. \end{aligned} \tag{3.15}$$

2-M') Process Logical Expressions & Summarize

Maximizing Expression (3.15) for goal `D.1` in goal model M', while ensuring the domain properties and assumptions (i.e., $Speed_t$, $Speed_{t+1}$, and *Distance*) in Table 3.2 are maintained, results in a maximized value of 0.0 (i.e., *false*). This result indicates that no completeness counterexample exists in the updated M' goal model for decomposition from `D.1`.

3.4.2 Incomplete OR Decomposition: Goal B.3

In this section, we describe the *Ares* process for a single decomposed requirement, `B.3`, in goal model M'. The decomposition of goal `B.3` is shown in Figure 3.5. The goal `B.3` is intended to specify requirements allowing the ACC system to reduce the current speed by controlling both the accelerator and brake. The decomposition requires either of its decomposed requirements to be satisfied. First, in goal `B.4`, the throttle (*Throttle Actuator*) must be reduced. Second, in goal `B.15`, the brake (*Brake Actuator*) must be increased. Either one of these, or both, is intended to be sufficient to lower the speed of the vehicle unless the speed ($Speed_t$) is already zero.

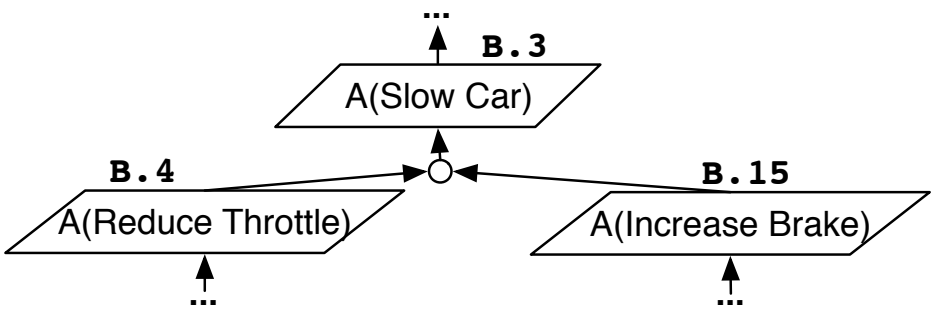


Figure 3.5: Example OR Decomposition in Goal Model M'

1-M') Generate Detection Logic for Model M'

In order to detect a completeness counterexample, we maximize an expression similar to Expression (3.9), but instantiated for `B.3`:

$$Counterexample_{\texttt{B.3}} = Satisficement_{Decomp}(\texttt{B.3}) - Satisficement(\texttt{B.3}). \tag{3.16}$$

Expanding Expression (3.16) via Expression (3.7) and the decomposition specific for goal `B.3` results in:

$$\begin{aligned} Counterexample_{\texttt{B.3}} = max(&Satisficement(\texttt{B.4}), \\ &Satisficement(\texttt{B.15})) - Satisficement(\texttt{B.3}). \end{aligned} \tag{3.17}$$

Using the corresponding utility functions defined in Table 3.2 yields the following incompleteness detection expression:

$$\begin{aligned} Counterexample_{\texttt{B.3}} = min(&Throttle\ Actuator < Throttle\ Pedal\ Sensor, \\ &Brake\ Actuator > Brake\ Pedal\ Sensor) - \\ &(Speed_t > Speed_{t+1} \vee \\ &(Speed_t == MIN \wedge Speed_{t+1} == MIN)). \end{aligned} \tag{3.18}$$

2-M') Process Logical Expressions & Summarize

Maximizing the completeness counterexample Expression (3.18), while ensuring the domain properties and assumptions (e.g., $Speed_t$, and $Speed_{t+1}$) in Table 3.2 are maintained, results in a maximized value of 1.0. This result indicates the existence of a completeness counterexample. The counterexample provides the specific environmental configuration of variables defined in Table 3.5, in both the normalized (from 0.0 to 1.0) values symbolically analyzed and their real world values. The counterexample indicates that goal `B.3` is unsat-

isfied, and therefore the current speed ($Speed_t$) does not decrease even when at least one of the decomposed goals is satisfied (i.e., `B.4`). Therefore, the decomposition is not sufficient to decrease speed. While either decomposed goal would be sufficient to slow the car if there were no other changes, when goal `B.4` decreases the throttle by a fractional amount (in this case 50.0% of the available throttle) the brake can also be reduced by a comparable amount, causing a neutral impact to the vehicle speed.

Table 3.5: Counterexample Variables for `B.3` in Goal Model M'

Variable or Requirement	**Normalized Value**	**Value**
$Speed_t$	0.5	50.0 MPH
$Speed_{t+1}$	0.5	50.0 MPH
Throttle Actuator	0.5	50.0%
Throttle Pedal Sensor	1.0	100.0%
Brake Actuator	0.0	0.0%
Brake Pedal Sensor	0.5	50.0%
Goal `B.3`	0.0	Unsatisfied
Goal `B.4`	1.0	Satisfied
Goal `B.15`	0.0	Unsatisfied

In order for the decomposition of goal `B.3` to be complete, there must be a requirement that ensures the satisfaction of one decomposed goal is not negatively impacted by another decomposed goal. Many specifications would be sufficient and could be represented via additional requirements, including adjusting the throttle and brake proportionally to one another. However, one of the most straightforward possible solutions is to change the decomposition operator from OR to AND. If both decomposed goals are required to contribute to slowing the car, then neither goal can negatively impact the other. Figure 3.6 shows the portion of the updated goal model, M'', where the decomposition of goal `B.3` has been revised from an OR-decomposition to an AND-decomposition.

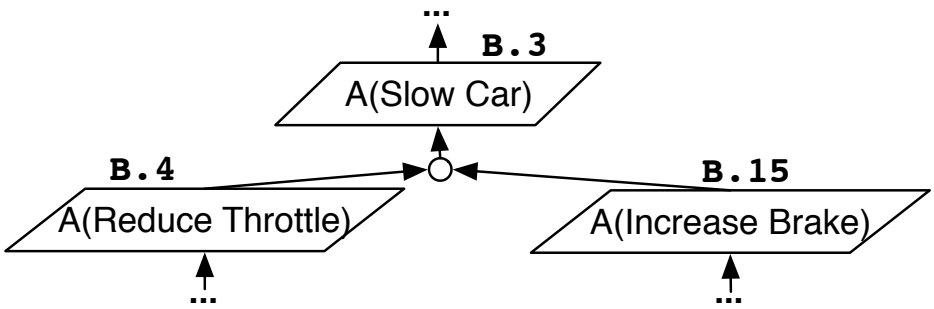


Figure 3.6: Revised OR Decomposition in Goal Model M''

1-M'') Generate Detection Logic for Model M''

While the utility functions do not change, the decomposition operator does. Expression (3.17) is updated to use the minimum of the decomposed requirements (i.e., AND) instead of the maximum (i.e., OR) in the following equation:

$$Counterexample_{\texttt{B.3}} = min(Satisficement(\texttt{B.4}), \\ Satisficement(\texttt{B.15})) - Satisficement(\texttt{B.3}). \tag{3.19}$$

Using the corresponding utility functions defined in Table 3.2 yields the following incompleteness detection expression:

$$Counterexample_{\texttt{B.3}} = min(\,Throttle\ Actuator < Throttle\ Pedal\ Sensor, \\ Brake\ Actuator > Brake\ Pedal\ Sensor) - \\ (Speed_t > Speed_{t+1} \vee \\ (Speed_t == MIN \wedge Speed_{t+1} == MIN)). \tag{3.20}$$

2-M'') Process Logical Expressions & Summarize

Maximizing the updated completeness counterexample expression (i.e., the entire Expression (3.20), where the minimum satisficement of any of the decomposed requirements is used, rather than the maximum satisficement of any of the decomposed requirements) for

goal B.3 in goal model M'', while ensuring the domain properties and assumptions (e.g., $Speed_t$, and $Speed_{t+1}$) in Table 3.2 are maintained, results in a maximized value of 0.0 (i.e., *false*). This indicates that no completeness counterexample exists in the updated M'' goal model for B.3.

3.4.3 Discussion

The application of *Ares* to each of the decompositions in goal model M identifies two incomplete decompositions: an AND decomposition from goal D.1, and an OR decomposition from B.3. Both of these incompleteness causes are requirements specification errors that would be difficult to detect with either manual analysis or testing techniques due to the unexpected simultaneous use of brake and throttle. Neither incomplete decomposition was artificially inserted, instead they were unexpected artifacts that remained even after review by automotive industrial collaborators. The first incomplete decomposition, goal D.1, required additional requirements (D.New1, and D.New2). The revised goal model, M', still contained an incomplete decomposition for goal B.3. This final incompleteness problem was addressed by changing the decomposition operator of B.3 from OR to AND to obtain the final goal model M'', which contained no further incomplete requirements decompositions.

3.4.4 Threats to Validity

Ares depends on the accuracy of the utility functions provided by Athena [75]. However, these utility functions are not guaranteed to be correct. The accuracy of the completeness counterexample detection is bounded by the accuracy of the utility functions and the related domain properties and assumptions.

3.5 Symbolic Analysis Related Work

While a number of projects have dealt with partial specifications and uncertainty about the requirements (e.g., [85, 93]), this work focuses on automatically detecting incomplete requirements in the context of requirements decomposition (i.e., requirements refinement). Several different strategies have been developed to address requirements incompleteness with respect to decomposition. We overview these techniques and compare them to the *Ares* approach.

A search-based technique for the generation of obstacles [2] has been used to create counterexamples to completeness, where the incompleteness is detected with respect to the domain properties. However, this process requires manual review of the obstacles generated to ensure their relevance and applicability. *Ares* automatically generates counterexamples that are guaranteed to indicate incomplete requirement decomposition based on the artifacts provided, and does so within the context of the decomposed requirement rather than the entire domain.

Methods with formal guarantees exist, including analysis of behavioral state-based systems [56] and the application of theorem provers to formally described requirements [87]. However, methods that employ formal proofs are heavyweight solutions that require expertise with theorem proving to describe requirements [87] or low-level functional details [56]. *Ares* supports the use of hierarchical requirements modeling, supplemented by specifications of environmental conditions, described in terms of simple scalar and Boolean expressions involving application variables.

Formal decomposition of requirements that guarantees completeness with respect to decomposition [31] is limited to defined decomposition strategies and requires a high degree of formality. It is possible to add completeness criteria to formal specification languages, though not all criteria can be enforced via language semantics [67], thus potentially allowing incomplete requirements. *Ares* is not limited to specific decompositions or language semantics but supports unrestricted decomposition patterns and requirement descriptions.

Process rigor [98] and analysis of natural language requirements [46, 69] have both been used to lower the likelihood of incomplete requirements, but are unable to provide guaranteed detection or avoidance of incomplete requirements.

Ares is unique as it applies symbolic analysis to automatically-generated utility functions to detect incomplete decompositions. By using symbolic analysis of application and domain specific properties provided by domain experts at design-time, *Ares* supports full analysis of the hierarchical requirements models.

3.6 Summary

In this chapter, we have presented *Ares*, a design-time approach for detecting incomplete requirements decomposition using symbolic analysis of hierarchical requirements models. Unlike previous incomplete requirements detection methods, *Ares* detects incomplete requirements decompositions while not limiting the allowable decomposition strategies.

We demonstrate *Ares* on an adaptive cruise control system developed in collaboration with our automotive industrial collaborators. We show that *Ares* is able to automatically detect incomplete requirements decompositions and provide completeness counterexamples in seconds.

Chapter 4

Detecting Incomplete Requirements: Using Evolutionary Computation

While we previously developed a technique to detect incomplete requirements decomposition with respect to a given environmental scenario, a single completeness counterexample may not clearly indicate the extent that incomplete requirements impacts the system or what range of environmental scenarios are affected. This chapter introduces *Ares-EC*, a design-time approach for detecting incomplete requirements decomposition using a combination of evolutionary computation and symbolic analysis of hierarchical requirements models to detect a set of representative incompleteness counterexamples. We again illustrate our approach by applying *Ares-EC* to the requirements model of an industry-based automotive adaptive cruise control system. *Ares-EC* is able to apply symbolic analysis and evolutionary computation to automatically detect multiple, diverse, and representative sets of requirements incompleteness counterexamples at design time.

4.1 Introduction

Even when requirements incompleteness is automatically identified, a single counterexample may not be not sufficient. Just as it is not sufficient for a system designer to correct a

single environmental scenario impacted by incompleteness, neither is it sufficient to simply indicate that requirements are incomplete. Instead, the range of environmental scenarios impacted by incomplete requirements decomposition should be identified in order to facilitate the task of revising the requirements. For example, a requirement for a vehicle may be to stop. In an idealized system, applying brake force (e.g., from hydraulic brakes) would be sufficient. Applying the brakes may be insufficient if the throttle can overwhelm the braking force, thus indicating that the decomposition is incomplete. However, that single counterexample is not representative of the range of operational scenarios that are impacted by incompleteness. For example, in inclement weather, brake force may not be sufficient without anti-lock brakes. Not only is enumerating all necessary decomposed requirements difficult, it is also challenging to identify the range of impacted scenarios to assess necessary additional requirements.

This chapter describes *Ares-EC*, an approach that combines symbolic analysis and evolutionary computation to automatically identify sets of representative environmental configurations where completeness properties are violated in a hierarchical requirements model. Hierarchical requirements satisfaction can be assessed in two ways: its individual satisfaction or the satisfaction of a requirement's aggregate decomposed requirements (i.e., children or sub-requirements). Given complete decomposition, a requirement should be satisfied whenever its aggregate decomposed requirements are satisfied [37]. *Ares-EC* uses symbolic analysis to identify individual counterexamples and uses evolutionary computation to search for sets of diverse representations of counterexamples based on the previously identified counterexamples. By employing symbolic analysis, *Ares-EC* can guarantee that a single counterexample will be found, if one exists. Evolutionary computation, on the other hand, can identify multiple diverse counterexamples in parallel. *Ares-EC* identifies incomplete decompositions in the form of sets of representative environmental scenarios, or counterexamples, within a valid range of values for the variables in the system in which the incompleteness is expressed. Counterexamples are then summarized for the system designer to revise the requirements accordingly.

Ares-EC applies utility functions [75] to assess individual requirements for completeness within a hierarchical requirements model. Expressions representing completeness counterexamples used in both symbolic and evolutionary computation are defined in terms of these utility functions. While utility functions have been used to measure run-time satisfaction of requirements [50, 75], *Ares-EC* analyzes the utility functions (via expressions representing incompleteness) at design time. For each requirement's decomposition, *Ares-EC* identifies a representative set of counterexamples from the range of possible requirement variables' values. Counterexamples are identified as environmental conditions that cause a requirement to be unsatisfied while its aggregate decomposed requirements are satisfied. Sets of counterexamples are identified for an industry-based requirements model for an automotive application, and the incomplete requirements decompositions along with their representative set of counterexamples are summarized for the system designer.

The contributions of this chapter are as follows:

- We introduce a design-time, symbolic analysis and evolutionary computation approach to automatically detect diverse and representative sets of completeness counterexamples in hierarchical requirements models.
- We present a prototype implementation of the *Ares-EC* approach.
- We demonstrate the applicability of *Ares-EC* on an industry-based automotive example, an adaptive cruise control system.

The remainder of this chapter is organized into the following sections. Section 4.2 details the *Ares-EC* approach. Section 4.3 describes the results of a case study, and Section 4.4 details related work. Finally, Section 4.5 summarizes the work.

4.2 Evolutionary Computation Approach

Ares-EC is an automated method for identifying sets of completeness counterexamples

in a hierarchical requirements model. A requirement is considered to be incompletely decomposed if there exists a case such that a parent requirement is unsatisfied while the set of its decomposed requirements are satisfied [37].

The *Ares-EC* process, as shown in Figure 4.1, generates detection logic using utility functions and the decompositions in a hierarchical goal model. These logical expressions are then processed by a search based method to identify solutions that represent completeness counterexamples. Next, we detail the *Ares-EC* approach and provide a comparison of four search methods that were considered before determining which approach to finally use.

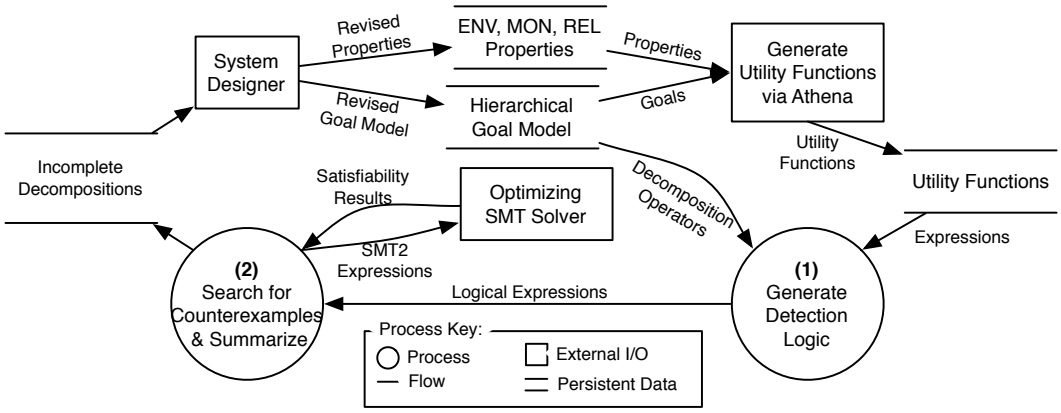


Figure 4.1: *Ares-EC* Data Flow Diagram

4.2.1 Step 1: Generate Detection Logic

Ares-EC makes use of utility functions generated by Athena [75], just as *Ares* does. A completeness counterexample is an unsatisfied parent requirement with a set of satisfied decomposed requirements [37] indicating there are additional child requirements necessary to satisfy the parent requirement. For example, `D.1` is incompletely decomposed if it is not satisfied according to its utility function value, but the set of its decomposed requirements (i.e., `D.2`, `D.5`, and `D.3`) is satisfied (i.e., the minimum of an AND-decomposition) as shown

in Equation (4.1).

$$\begin{aligned} Counterexample_{\texttt{D.1}} = min(&Satisficement(\texttt{D.2}), \\ &Satisficement(\texttt{D.5}), \\ &Satisficement(\texttt{D.3})) - Satisficement(\texttt{D.1}). \end{aligned} \tag{4.1}$$

Each of the requirements referenced in Equation (4.1) may either be satisfied (i.e., 1.0) or unsatisfied (i.e., 0.0) based on their utility function. By instantiating the satisficement expressions in Equation (4.1) with their respective utility values (from Table 3.2), we obtain the expression in Equation (4.2).

$$\begin{aligned} Counterexample_{\texttt{D.1}} = min(&Speed_t == Desired\ Speed \wedge \\ &Distance > Safe\ Distance, \\ &true, \\ &Throttle\ Actuator == Throttle\ Pedal\ Sensor) - \\ &Speed_t == Speed_{t+1} \end{aligned} \tag{4.2}$$

Similarly, OR-decomposed requirements such as `B.3` use the maximum of the decomposed requirements, as shown in Equation (4.3).

$$\begin{aligned} Counterexample_{\texttt{B.3}} = max(&Satisficement(\texttt{B.4}), \\ &Satisficement(\texttt{B.15})) - Satisficement(\texttt{B.3}). \end{aligned} \tag{4.3}$$

A similar expression is generated for every requirement that is decomposed. Identifying an optimum (i.e., a return value of 1.0) for a completeness counterexample expression indicates a counterexample exists.

4.2.2 Step 2: Search for Counterexamples and Summarize

In this work, we compare four methods to identify which provides the largest range of multiple counterexamples. Previously, symbolic analysis has been used to identify single counterexamples of incomplete requirements decomposition [37]. In contrast, here we apply evolutionary computation to search for a range of distributed counterexamples. Since we know that evolutionary computation is not guaranteed to find a solution, especially in 'needle in a haystack' cases, we supplement our evolutionary approach with initial optimal results from our symbolic approach. Finally, we periodically use our symbolic approach to re-seed the evolutionary population as a means to overcome additional 'needle in a haystack' cases.

For simplicity, we configure the parameters for evolutionary computation based on empirical feedback with an emphasis on optimal results and execution time. All instances of evolutionary computation used in this work make use of 200 individuals in the population, a tournament size of 8 is used for mating selection, a tournament of size 4 is used for survival selection, and a mutation rate of 5%. Executions have been limited to 2000 generations and execution is on the order of seconds. Mating selection is performed by an eight-way tournament based on the novelty of individuals within a set of randomly-selected individuals. Crossover for mating is performed by the SBX crossover operator [35] for each real-valued variable (i.e., variable in the completeness counterexample expressions) in the individuals selected. Mutation is performed on five percent of the individuals by randomly modifying a single real-value representing a variable in the genome. Survival via a four-way tournament is used to maintain population size and is based on fitness as measured by the satisfaction of the completeness counterexample expression and is elite preserving as no member of the population will be replaced with a less optimal member. Tournaments of this size were selected in an effort to increase the likelihood that an optimal value takes part in the tournament.

The genetic algorithm emphasizes search diversity via the mating selection and mutation operators (5% chance of mutation), while survival selection optimizes the results of the diverse search. The optimal is a population where each individual is at an optimum *and*

each individual is as far as possible from the other individuals based on the Manhattan distance [28] of the genotype. The genetic evolution described here differs from other genetic algorithms that search for a single optimal individual, since we are looking for a collection of diverse solutions.

Symbolic Analysis Only:

For each requirement, a utility function is used to represent the satisfaction of the requirement based on a set of environmental and system variables that make up the utility function. The parent requirement is symbolically compared to the combined derived requirements, via the completeness counterexample expressions. Detecting a single completeness counterexample is identified via symbolic analysis of the completeness counterexample expressions (e.g., Equation (4.1) or (4.3)) using existing techniques [37]. We include this technique here as a means to establish existence of at least one counterexample and for comparison to the other search-based techniques. Specifically, symbolic analysis, via Microsoft's SMT solver Z3 [32], is used to evaluate the entire range of applicable environmental configurations and system variables used in the completeness counterexample expressions for each decomposed requirement.

In cases where the completeness counterexample expression cannot be satisfied, then no counterexample exists and the requirement is thus complete, thereby alleviating the need to perform additional analysis.

Evolutionary Computation Only:

EC-only employs evolutionary computation in the form of a genetic algorithm to search for *both* novel and optimal results. Instead of searching for a single optimal solution to a counterexample expression, EC-only searches for multiple solutions with optimal phenotype responses (e.g., requirements with utility function values that indicate an incomplete decomposition) and maximized genotype novelty (e.g., a large difference in the environmental

scenario upon which the requirements are applied). The expected results are a population of optimal individuals (i.e., individuals that represent counterexample) distributed across the range of the genotype. The genome is an array of real-valued variables, one for each variable that exists in the completeness counterexample expression.

Symbolic Analysis, then Evolutionary Computation:

While evolutionary computation alone can provide a method of searching with an emphasis on diversity, two issues can occur:

- First, the so called 'needle in a haystack' problem may make finding the optimum solution significantly unlikely to be found.
- Second, expressions without a gradient between satisfied and unsatisfied are likely to take more time to find the optimum, and perhaps degenerate to random search, since the requirements satisfaction is used to calculate the fitness function.

EC-SA alleviates these two problems by utilizing symbolic analysis to identify a single optimum, which is used to seed a portion (10%, or 20 of the 200 individuals) of the initial population. Given an initial optimum, the diverse search is intended to identify a distributed set of optimum values.

Periodic Evolutionary Computation with Symbolic Analysis:

Symbolic analysis may provide a starting point for evolutionary computation, yet despite the guarantee of optimal individuals in the initial population, two issues can still occur:

- First, while the 'needle in a haystack' problem is alleviated for a *single* optimum, additional optima may also be similarly unlikely to be found.
- Second, a change in one variable may require a change in another variable in a single individual to identify another optimum, resulting in additional search time.

PSAEC overcomes these problems by periodically re-analyzing the completeness counterexample expression symbolically, with an added constraint to maximize the distance from a selection of existing individuals in the population. If another optimum is found then that individual is added to the population. We allow PSAEC to select up to 10 random individuals in the population to create a distance constraint from each of them during the first half of the generations. If a new and diverse counterexample is found, then 20 random individuals are replaced with the new counterexample.

4.2.3 Scalability and Limitations

Ares-EC is not guaranteed to identify all completeness counterexamples, even when using evolutionary computation with periodic symbolic analysis. For example, given a set of 'needles in a haystack,' two 'needles' that are close in genotype distance (e.g., when the throttle is at 72% and 74%, but not at 73%) may cause one to be ignored in favor of counterexamples that are further afield. While no guarantees can be made about identifying all completeness counterexamples, the larger the set of counterexamples and greater the diversity found, the more representative the solution set is of the incompleteness. Ultimately, detecting completeness counterexamples is limited to the quality and fidelity of the hierarchical requirements model and utility functions.

4.3 Evolutionary Computation Case Study

This section describes and compares the results of applying the four different methods of identifying counterexamples that satisfy the generated requirement completeness counterexample expressions. These methods are symbolic analysis only (SA), evolutionary computation only (EC), SA-Initialized EC (SAIEC), and Periodic-SA with EC (PSAEC). Each of these methods were executed 50 times[1]. For the SA results, there is no difference between

[1]To ensure reproducibility via statistical measures given the stochastic nature of evolutionary computation.

executions, but for the results that include EC, the results vary across executions. Results are compared for two incomplete requirements (Goals `D.1` and `B.3`, to 'Maintain Speed' and 'Slow Car', respectively) that were previously shown to be incomplete using SA [37]. However, in contrast to previous solutions [37], *Ares-EC* identifies multiple representative counterexamples as measured by a distance metric. Methods are compared based on their ability to return diverse counterexamples. Next, we describe in detail the results from applying each of these four techniques and analyze the results.

4.3.1 Symbolic Analysis

SA identified a single counterexample for both goals `D.1` and `B.3`. Intrinsically, there is no range or diversity in a solution set of one result. SA has been previously used to identify completeness counterexamples [37] and is included here for comparison with the other search-based methods.

4.3.2 Evolutionary Computation

EC attempts to address the fundamental shortcoming with the results from only SA by identifying a population of results, rather than a single result. However, even after 50 executions, no counterexamples could be identified. Unlike the single SA result, EC attempts to identify a range of solutions that are more representative of the scope in which the requirement incompleteness exists. In this case, the lack of variation in fitness (i.e., fitness values are either 0.0 or 1.0) reduces the EC to random search. Significantly larger populations (5000) and generations (20000) were also used with no success.

4.3.3 SA Initialization then EC

While the EC-based method alone was unable to provide counterexamples to a single requirement incompleteness, it is possible to start with a known optimum and search for similar

counterexamples. The SAIEC method results in 200 counterexamples within a population of 200. This result does not mean that there are 200 missing or incomplete requirements, only that this method identified 200 representative counterexamples for each single incomplete requirement. For example, if incomplete requirements decompositions were only found when the brake is depressed more than 50%, then the 200 counterexamples should be in a distribution ranging from being pressed 50% to being pressed 100%. Unlike the EC-only method, providing the EC algorithm with a sample optimum has made it possible to find additional optima resulting in the identification of usable counterexamples from the population.

The SAIEC method is able to find a counterexample for every member of the population for each of the known incomplete requirements (Goals `D.1` and `B.3`). SAIEC is clearly superior to EC alone, as EC alone is unable to identify any counterexamples. SAIEC is also clearly superior to SA, as SA is unable to provide any range or diversity within its counterexamples as SA only identifies a single counterexample.

4.3.4 Periodic SA with EC Results

The additional number of optimal results provided by initializing the EC-based method with a counterexample found from SA still may leave an intrinsic bias to the original optimal set in the results. When multiple variables must change in order to maintain an optimum, it is more difficult to identify additional optima due to the likelihood of a crossover or mutation maintaining the relationship between those variables. In an effort to identify the largest range of counterexamples, it may be that periodically adding an optimal solution outside of the known solutions would improve the overall range of solutions by overcoming the dependencies between variables. Similar to the SAIEC method, the PSAEC identifies 200 diverse counterexamples within a population of 200.

Similar to SAIEC, the PSAEC method is clearly superior to EC alone for the same reason that PSAEC is able to identify counterexamples while using EC only is not. PSAEC is also superior to SA only, as SA only identifies a single counterexample.

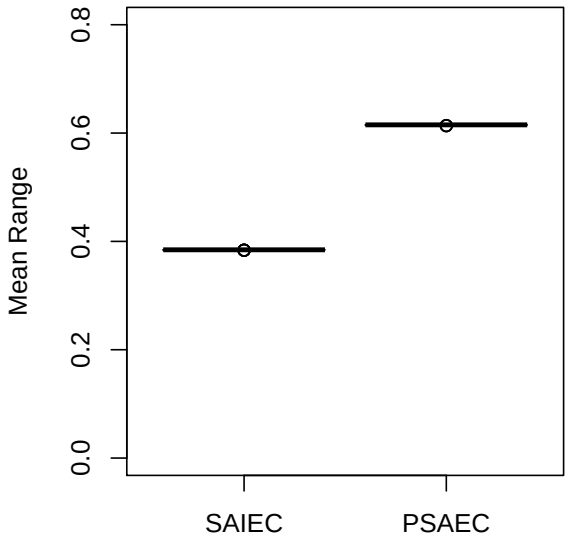


Figure 4.2: Requirement `D.1`

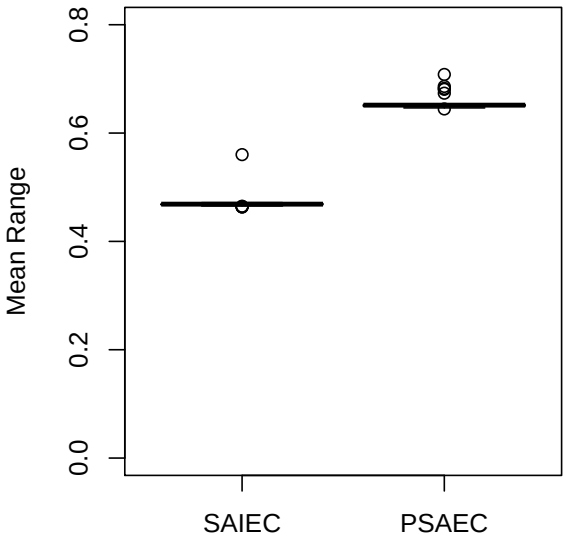


Figure 4.3: Requirement `B.3`

4.3.5 Comparison

While SA can only be used to identify the existence of a requirement completeness counterexample, additional counterexamples provide more information on the range and scope of the incompleteness. EC-based methods can identify additional counterexamples in parallel, but encounter difficulties satisfying expressions with fitness cliffs or 'needle in a haystack' solutions (i.e., in an EC-only method) or difficulties with identifying additional novel solutions due to correlated variables (i.e., in a SAIEC search method). In fact, in this specific case, the EC-only method was unable to identify results due to the lack of a fitness gradient. In the general case, the greatest range of optimal genotype values is provided by the PSAEC that escapes limitations of the SAIEC search methods. Evidence of this finding can be seen in Figures 4.2 and 4.3 for requirements incompleteness for goals `D.1` and `B.3`, respectively, where the mean range of genotype values in each individual can be seen in box plots.

It is necessary to statistically compare the two methods that identified a range of solutions (i.e., SAIEC and PSAEC), since they are both able to identify the same number

of counterexamples. We define the null hypothesis H_0 to state that there is no difference between the range of optimal solutions for the SAIEC and PSAEC methods. We also define an alternative hypothesis, H_1, that states that there is a difference between the range of optimal solutions for SAIEC and PSAEC methods. In both cases, in goals `B.3` and `D.1`, PSAEC achieves statistically significant larger mean range values over 50 executions as measured using the Mann-Whitney U-test ($p < 0.05$ where $p = 2.2 * 10^{-16}$). Therefore we can reject the null hypothesis, H_0, in favor of the alternate hypothesis H_1 due to the statistically significant difference.

4.4 Evolutionary Computation Related Work

This section covers related work for both requirements completeness and search methods that maintain diversity. While there is a broad collection of research into leveraging search-based techniques for requirements-related tasks, many of which are described in surveys [54, 99], to the best of the authors' knowledge, none have explicitly tackled the problem of requirements decomposition incompleteness.

4.4.1 Requirements Completeness

Outside of process rigor [98], formal guarantees of requirements completeness exist in the form of decomposition strategies that are proven to define complete decomposed requirements [31]. Completeness criteria may be added to formal specification languages, though incomplete requirements may still exist due to criteria that cannot be enforced by language semantics [67]. A method exists to detect incomplete decompositions using symbolic analysis, however only a single counterexample for each incomplete requirement is produced [37].

Ares-EC is unique as it applies symbolic analysis and evolutionary computation to automatically-generated utility functions to detect *sets* of representative completeness counterexamples without restricting decompositions to a finite set of formal patterns.

4.4.2 Search for Diversity

Multi-objective optimization (e.g., NSGA-II [36]) identifies multiple solutions, but the solutions represent the Pareto front of a tradeoff between two or more objectives [34]. However, if the objectives are not competing, then the problem collapses to single-objective optimization. Our method of searching for requirement completeness counterexamples does not contain competing objectives, but rather a single objective with multiple solutions.

Novelty search uses evolutionary computation to identify novel behaviors [66] across the genotype. Rather than identifying a population of optimum solutions, novelty search identifies the range of possible solutions from an optimum to the worst solution. Niching is typically used for multi-modal problems [43], rather than problems with an area of optimal results.

The search method used by *Ares-EC* identifies multiple optimum solutions in parallel while maximizing diversity of the solutions.

4.5 Summary

In this chapter, we have presented *Ares-EC*, a design-time approach for detecting incomplete requirements decomposition using symbolic analysis and evolutionary computation to analyze hierarchical requirements models. Unlike previous incomplete requirements detection methods, *Ares-EC* detects representative sets of incomplete requirements decompositions while not limiting the allowable decomposition strategies.

We demonstrated *Ares-EC* on an adaptive cruise control system developed in collaboration with our automotive industrial collaborators. We show that *Ares-EC* is able to automatically detect incomplete requirements decompositions and provide sets of completeness counterexamples in seconds, allowing holistic correction of the fundamental incompleteness issues rather than the correction of single counterexamples. Further, by combining symbolic analysis with evolutionary computation we achieve the benefits of both techniques.

Chapter 5

Detecting Incomplete Requirements: At Run Time

The validity of run-time monitoring of system goals and requirements depends on both the completeness of the requirements, as well as the correctness of the environmental assumptions. Often specifications are built with an idealized view of the environment that leads to incomplete and inconsistent requirements related to non-idealized behavior. Worse yet, requirements may be measured as satisfied at run time despite an incomplete or inconsistent decomposition of requirements due to violated environmental assumptions. While methods exist to detect incomplete requirements at design time, environmental assumptions may be invalidated in unexpected run-time environments causing undetected incomplete decompositions. This chapter introduces *Lykus*, an approach for using models at run time to detect incomplete and inconsistent requirements decompositions at run time [39]. We illustrate our approach by applying *Lykus* to a requirements model of an adaptive cruise control system from our industrial collaborators. *Lykus* is able to automatically detect instances of incomplete and inconsistent requirements decompositions at run time.

5.1 Detecting at Run Time

While run-time requirement monitors are intended to measure and report the satisfaction of the requirements in a software system, they are only effective if the requirements are complete and consistent. Problematically, unexpected environmental scenarios that may lead to incomplete or inconsistent requirements in cyber-physical systems may also allow run-time monitors to erroneously assess requirements satisfaction. For example, if a requirements decomposition is incomplete, then the decomposed requirement would be satisfied even if the missing requirement(s) were unsatisfied. This chapter presents *Lykus*,[1] an approach to automatically detect incomplete requirements coverage at run time.

Detecting incomplete requirements is still an active research area [2, 23, 46, 69, 98, 37], which becomes more complicated when design-time environmental assumptions are found to be invalid at run time. For example, a requirement for a vehicle may be to accelerate. In an idealized system, applying the throttle (e.g., pressing the gas pedal) would be sufficient. However, the assumption that spinning the wheels faster due to an increased throttle affects vehicle speed may be found to be invalid. Given an environmental scenario with low traction (e.g., ice) or where the vehicle's wheels do not touch the ground (e.g., the vehicle is flipped over) would invalidate the earlier assumption when the vehicle did not accelerate. Formal design-time methods to decompose goals and requirements with guaranteed completeness exist [31], though they are limited to only specific formal decomposition rules. Design-time methods exist that are not limited to formal decomposition patterns [37, 38], but make assumptions about the environment that may be shown to be invalid at run time. Problematically, run-time monitoring of incompletely decomposed requirements may indicate satisfaction when the missing requirement(s) would be unsatisfied (e.g., the throttle is increased indicating satisfaction, but the vehicle does not accelerate). Currently, no methods exist to detect incomplete and inconsistent requirements decompositions at run time in order to ensure relevant requirement satisfaction assessment.

[1] *Lykus* is the mortal son of *Ares*, who sacrificed strangers to his father.

This chapter describes *Lykus*, the extension of *Ares*, a design time model-driven technique [37] to detect incomplete and inconsistent requirements decompositions at run time. *Lykus* adapts run-time monitors to ensure relevant assessment in the context of the physical environment and requirements model. While requirements in a hierarchical decomposition may be assessed directly or based on the satisfaction of the decomposed child requirements, neither method is sufficient to assess satisfaction alone in the presence of an incomplete or inconsistent decomposition. However, by using a combination of both assessments, it is possible to detect an incomplete or inconsistent decomposition and calculate the correct requirement satisfaction. For example, given a parent requirement to accelerate a vehicle, it may appear that the requirement is satisfied when all of the decomposed child requirements (e.g., increase throttle) are satisfied. If the vehicle *does not* accelerate, then there is an incomplete requirement decomposition and additional requirements that are unrepresented are unsatisfied. Similarly, if the vehicle *does* accelerate but the decomposed requirements are not satisfied, then the parent requirement to accelerate is not being satisfied in the manner specified by the child requirements. Instead, it could be that the vehicle is rolling down a hill rather than accelerating via throttle. In both cases, the intent of the acceleration requirement is not met, therefore the requirement is unsatisfied. *Lykus* identifies incomplete and inconsistent decompositions at run time to dynamically modify requirement satisfaction monitors in order to provide more environmentally relevant assessments of the requirements unsatisfied. *Lykus* also identifies violated environmental assumptions when incomplete or inconsistent decompositions are identified at run time.

Lykus uses utility functions [75] to analyze individual requirements within the system specification. Rather than return the raw assessment values generated by the utility functions, *Lykus* identifies incomplete requirements (i.e., additional requirements are necessary but not specified) and inconsistent requirements (i.e., the system is not constrained in the manner defined by the requirements) at run time by comparing the utility function values for parent and child requirements. In the case of incomplete requirements, the parent utility

function value is modified to be 'unsatisfied,' as there exists at least one requirement that should have been decomposed (but was not) that is 'unsatisfied.' Similarly, in the case of an inconsistent satisfied requirement, the parent utility function value is also modified to be unsatisfied since the satisfaction did not take place according to the constraints imposed by the decomposed requirements. *Lykus* explicitly uses the decompositions from within a hierarchically decomposed goal model, typically a design-time model, to analyze the decomposition completeness and consistency at run time where specific scenarios representing the actual state of the environment and system are continually processed, or streamed, through the detection system.

The contributions of this chapter are as follows:

- We introduce a run-time approach to automatically detect incomplete and inconsistent requirement decompositions in hierarchical requirements models.
- We adapt the utility function results in the case of incomplete and inconsistent decompositions in order to assess satisfaction with respect to the actual physical environment conditions, as opposed to an environmental model.
- We identify the environmental assumptions that are shown to be invalid and are the contributing factor to the incomplete or inconsistent decomposition.
- We present a prototype implementation of the *Lykus* run-time analysis and requirement assessment approach.
- We demonstrate the applicability of *Lykus* on an adaptive cruise control system implemented on a rover vehicle.

The remainder of this chapter is organized into the following sections. The model used for the example application is defined in Section 5.2. Section 5.3 details the approach. Section 5.4 describes an example application, and Section 5.5 details related work. Finally, Section 5.6 discusses the conclusions and avenues of future work.

5.2 Complete Adaptive Cruise Control Input Model

Figure 5.1 details a hierarchical goal and requirements model of an Adaptive Cruise Control system that uses distance sensors to ensure a safe following distance from the car ahead by adjusting vehicle speed while simultaneously maintaining as close to the desired speed as possible. This goal model is updated from the one presented in Chapter 3 (Figure 3.1) to reflect the modifications performed according to *Ares* and *Ares*-EC analyses. Abbreviations *M* and *A* are used for *Maintain* and *Achieve*, respectively. Leaf nodes (i.e., agents) that are numbered in the requirements model are provided in Table 5.1. The ACC model defined here has been previously shown to be complete [37], given the assumed environmental conditions.

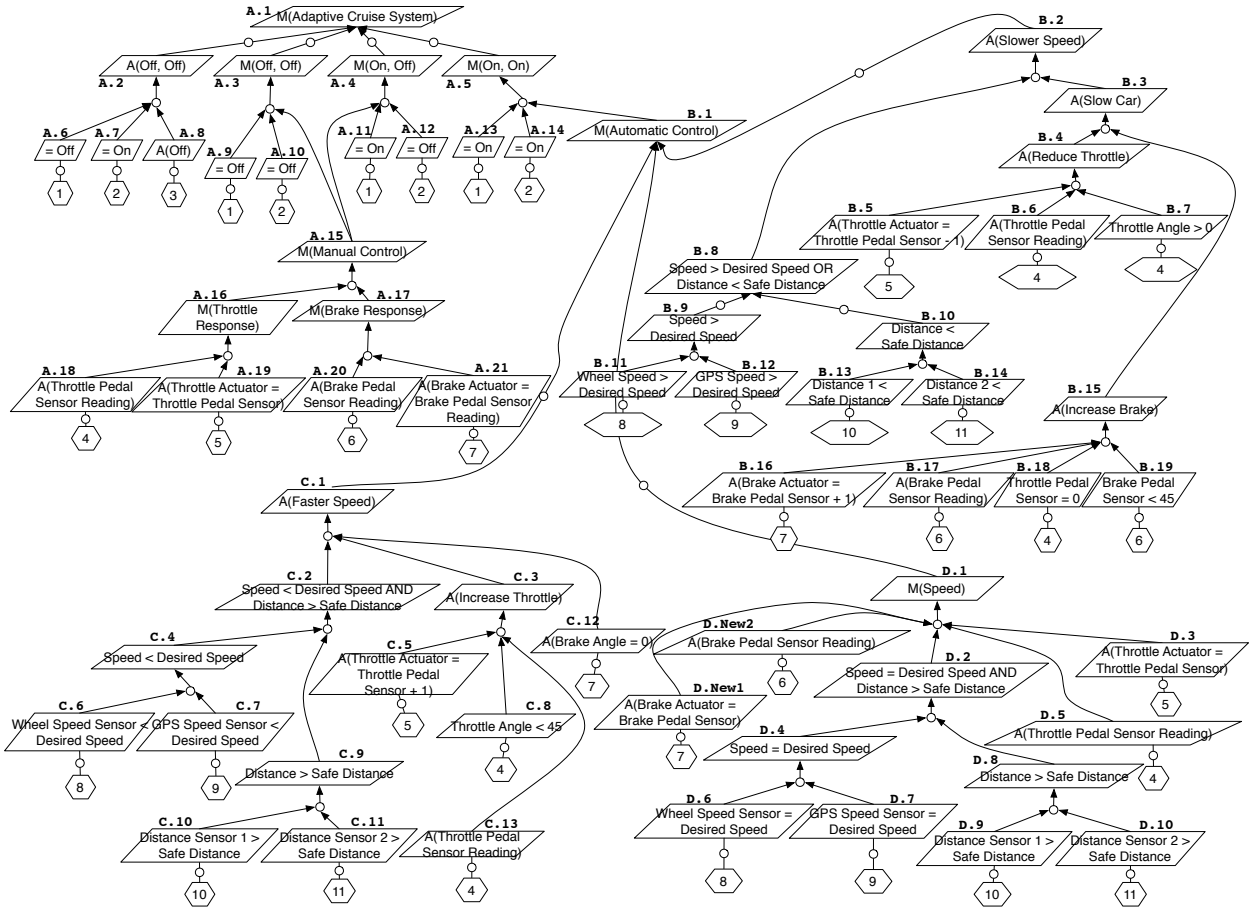


Figure 5.1: Adaptive Cruise Control Goal Model

The ACC model in Figure 5.1 is defined by four primary components: cruise control modes (i.e., goals with a prefix of '`A.`'), speed increase (i.e., goals decomposed from `C.1`),

Table 5.1: Agents used in Goal Model

#	**Agent (Sensor / Actuator)**
1	Cruise Switch Sensor
2	Cruise Active Sensor
3	Cruise Active Switch
4	Throttle Pedal Sensor
5	Throttle Actuator
6	Brake Pedal Sensor
7	Brake Actuator
8	Speed Sensor 1
9	Speed Sensor 2
10	Distance Sensor 1
11	Distance Sensor 2

speed decrease (i.e., goals decomposed from `B.2`), and maintain speed (i.e., goals decomposed from `D.1`). The speed is increased or decreased to maximize speed up to the desired speed while maintaining a safe distance from the target car (i.e., the car immediately in front). In cases where the safe distance is violated, the speed is decreased regardless of the desired speed. In cases that both the desired speed and safe distance are met, the speed is maintained.

The utility functions for the requirements in Figure 5.1 are derived from the **ENV**, **MON**, and **REL** properties in Table 5.2, where, for brevity, only values related to the ongoing example are presented. For example, the utility function for requirement `C.1` is given in Expression 5.1 indicating that goal `C.1` (*'Achieve Faster Speed'*) is satisfied if the speed value at time t, Speed_t, is less than the value at time $t+1$, Speed_{t+1}. Table 5.3 defines the variable value ranges and units.

$$Satisficement(\texttt{C.1}) = Speed_t < Speed_{t+1} \tag{5.1}$$

Table 5.2: **ENV**, **MON**, and **REL** Properties

	ENV	**MON**	**REL**
C.1	$Speed_t, Speed_{t+1}$		$Speed_t < Speed_{t+1}$
C.2	$Speed_t, Distance$	*Desired Speed*, *Safe Distance*	$Speed_t <$ *Desired Speed* $\wedge$ *Distance* $>$ *Safe Distance*
C.3		*Throttle Actuator*, *Throttle Pedal Sensor*	*Throttle Actuator* $>$ *Throttle Pedal Sensor*
C.12		*Brake Actuator*	*Brake Actuator* $==$ *MIN*
$Speed_t$		*Speed Sensor 1*, *Speed Sensor 2*	*Speed Sensor 1* $\vee$ *Speed Sensor 2*
$Speed_t$		*Throttle Pedal Sensor*, *Brake Pedal Sensor*	$max($*MIN*, *Throttle Pedal Sensor* $-$ *Brake Pedal Sensor*$)$
$Speed_{t+1}$		*Throttle Actuator*, *Brake Actuator*	$max($*MIN*, *Throttle Actuator* $-$ *Brake Actuator*$)$
Distance		*Distance Sensor 1*, *Distance Sensor 2*	*Distance Sensor 1* $\vee$ *Distance Sensor 2*

Table 5.3: Units and Scaling for Variables in Table 5.2

Variable	**Min**	**Max**	**Unit**
$Speed_t$	0.0	100.0	MPH
$Speed_{t+1}$	0.0	100.0	MPH
Distance	0.0	50.0	Inches
Desired Speed	0.0	100.0	MPH
Safe Distance	0.0	50.0	Inches
Throttle Actuator	0.0	100.0	%
Throttle Pedal Sensor	0.0	100.0	%
Brake Actuator	0.0	100.0	%
Brake Pedal Sensor	0.0	100.0	%
Distance Sensor 1	0.0	50.0	Inches
Distance Sensor 2	0.0	50.0	Inches
Speed Sensor 1	0.0	100.0	MPH
Speed Sensor 2	0.0	100.0	MPH
Cruise Switch Sensor	*Off*	*On*	Boolean
Cruise Active Sensor	*Off*	*On*	Boolean
Cruise Active Switch	*Off*	*On*	Boolean

5.3 Run-Time Approach

An overview of *Lykus* is presented in Figure 5.2. Similar to *Ares*, *Lykus* makes use of Athena [75] to generate the utility functions that are used as run-time monitors to detect incomplete and inconsistent decompositions of parent requirements in the goal model (e.g., Figure 5.1) using the properties in Table 5.2. Optionally, *Ares* can be used to detect incomplete requirements decompositions that can be identified at design-time [37]. The utility functions and goal model are then used by *Lykus* to generate logical expressions that represent the decompositions within the hierarchically decomposed goal model to detect both incomplete and inconsistent decompositions. These logical expressions are used by *Lykus* to

generate executable monitoring code that detects incomplete and inconsistent decompositions and accurately report parent satisfaction. The executable monitoring code generated by *Lykus* is used at run time to monitor the system operating in its physical environment to detect counterexamples and report satisfaction throughout the system's execution.

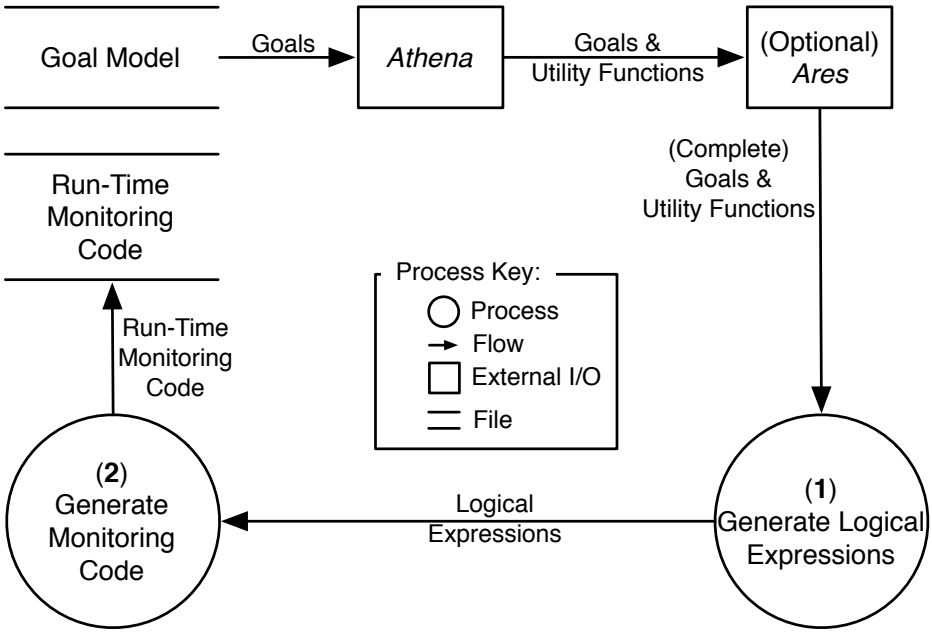


Figure 5.2: *Lykus* Data Flow Diagram

Lykus is applicable to systems that interact with their environment, where the environment is anything that is outside of the system-to-be. The environment may be other systems, not just a physical environment. Additionally, the system must be able to sense its expected impact on the environment to allow utility functions to measure the satisfaction of individual requirements directly, or indirectly using relationships between multiple sensors. Finally, requirements must be defined hierarchically to detect decompositional counterexamples.

5.3.1 DFD Step 1: Generate Logical Expression

The input to Step 1 is a goal model that *may* be analyzed at design-time for incomplete requirement decompositions using *Ares* [37] and utility functions generated by Athena [75]. *Lykus* outputs a set of requirements monitors and additional logic that detects incomplete

Table 5.4: Satisfaction Cases

Row	Utility Function		Updated Parent Requirement
	Parent Requirement	**Child Requirements**	
1	Unsatisfied	Unsatisfied	Unsatisfied
2	Unsatisfied	Satisfied	Unsatisfied (Incomplete)
3	Satisfied	Unsatisfied	Unsatisfied (Inconsistent)
4	Satisfied	Satisfied	Satisfied

and inconsistent requirements decomposition, as well as provides the status of the parent requirement satisfaction.

Unlike design-time solutions that use utility functions for analysis across an entire range of possible scenarios [37], *Lykus* applies the utility value functions at run time for a single scenario that the system is currently experiencing. Based on the results of the comparisons of the realized utility value functions, analysis is performed each time the utility functions are calculated at run time. Counterexamples are identified, and subsequently used immediately for run-time requirement assessments.

Table 5.4 lists the satisfaction of both the parent requirement and the set of decomposed child requirements along with the updated parent requirement satisfaction status to reflect the run-time evaluation of the parent in actual environmental conditions. For each decomposed requirement, two additional checks are performed, one for incomplete decomposition and one for inconsistent composition. If the requirement is neither incomplete nor inconsistent, then the requirement's utility function is unmodified. The possible parent and decomposed child requirement satisfaction results listed in Table 5.4 are discussed in the following subsections. If the parent's and decomposed child requirements' satisfactions do not match, then the parent must be unsatisfied due to either incomplete or inconsistent decomposition. Table 5.4 identifies the possible combinations of parent and decomposed child requirements satisfaction, along with the updated parent satisfaction based on run-time satisfaction measures.

Standard Unsatisfied Requirements

In the case where both the parent and the child requirements are unsatisfied (i.e., Row 1 in Table 5.4), the satisfaction of the parent does not change and the parent requirement remains unsatisfied at run time. Since both the utility function representing the satisfaction of the parent requirement and the combined satisfaction of child requirements are both unsatisfied, then the result is neither incomplete nor inconsistent. The requirement is correctly assessed as unsatisfied by both the parent and child requirements. That is, there is not a known missing requirement due to an unsatisfied parent requirement nor is there an inappropriately satisfied parent requirement due to a solution not specified by the decomposed child requirements. When a parent requirement is unsatisfied, we would normally expect the decomposed child requirements to be unsatisfied.

Unsatisfied Due to Incompleteness

In the case where the parent is unsatisfied, yet the decomposed child requirements are satisfied (i.e., Row 2 in Table 5.4), the parent requirement should remain unsatisfied due to an incomplete decomposition. While the aggregate child requirements indicate that the parent requirement should be satisfied, they only indicate satisfaction due to a missing requirement that would alter the satisfaction of the aggregate set.

Unsatisfied Due to Method Used

In the case where the parent requirement is satisfied, yet the decomposed child requirements are unsatisfied (i.e., Row 3 in Table 5.4), the parent status must be modified to indicate that it is unsatisfied. While the parent requirement utility function indicates that the parent requirement is satisfied, the method of satisfaction is not constrained to the solution provided by the decomposed requirements.

Standard Satisfied Requirement

In the case where the parent requirement and the decomposed child requirements are both satisfied (i.e., Row 4 in Table 5.4), then the decomposition is neither incomplete nor inconsistent. Regardless of the method of assessing the requirement (i.e., directly or via the requirement's decomposed child requirements) the assessment results in satisfaction for both the parent and children.

5.3.2 DFD Step 2: Generate Monitoring Code

The logic for comparing parent requirement and aggregate child requirements satisfactions is generated for each decomposed parent requirement. Neither direct measurement of each requirement (e.g., measuring parent requirements for satisfaction) nor measuring a requirement's aggregate decomposed requirements (e.g., measuring the set of child requirements for satisfaction) is sufficient to assess the satisfaction of the parent requirement in all cases. An implementation of Table 5.4 to calculate the updated satisfaction of requirement `C.1` at run time is given in Figure 5.3. This approach differs from the *Ares* approach, since here we detect at run time if a single specific environmental and system scenario related to the current state of the system includes incomplete or inconsistent requirements decomposition. The requirements, values of the variables, and (when included) environmental assumptions are recorded upon the detection of an incomplete or inconsistent requirements decomposition. Similar functions that update parent requirement status, as measured from the original utility functions, are provided for each of the parent requirements in the goal model in order to detect incomplete and inconsistent decompositions and the associated variables at run time.

```
bool updated_sat_of_c1() {
  // Assess Parent Satisfaction
  bool parent_sat = sat_of_c1();
  // Assess Aggregate Child Satisfaction
  bool child_sat = sat_of_c2() &&
      sat_of_c3() && sat_of_c12();

  if(parent_sat == child_sat)
    return parent_sat;
  else {
    // Save variables and requirement
    record_counterexample(``C.1'');

    // Unsat if incomplete / incorrect
    return false;
  }
}
```

Figure 5.3: Updated Utility Function Listing of `C.1`

5.3.3 Execution Time & Deployment

For each requirements decomposition, the computational effort of *Lykus* grows linearly with respect to the number of decomposed requirements, assuming a constant maximum size of each utility function. Instead of calculating if a requirements decomposition is incomplete or inconsistent in any scenario, *Lykus* calculates if a requirement decomposition is incomplete or inconsistent in only the current scenario (e.g., the agents periodically read from sensors to support the calculation of the utility functions). This is similar to checking a known NP problem, SAT, for a single set of true or false assignments rather than attempting to find which true or false assignments satisfy the Boolean expression.

More formally, identifying incomplete or inconsistent requirements decompositions exists is NP-Complete. That is, identifying an incomplete or inconsistent requirements decompositions is computationally expensive but verifying a specific incomplete or inconsistent requirements decomposition can be done very quickly. *Lykus* leverages the latter property and

verifies the decompositional completeness and consistency with respect to only the current environmental scenario.

Given that utility functions are widely used as run-time monitors [75] (e.g., "sat_of_ $*()$" in Figure 5.3), the additional cost *Lykus* incurs is used to calculate a single Boolean expression based on the concrete satisfaction of the utility functions (e.g., calculation of "parent_sat == child_sat" in Figure 5.3) for each decomposition.

Lykus calculates if each decomposition is incomplete or inconsistent for the current scenario as measured by the utility functions and their respective monitor properties (i.e., agents and sensors). The utility functions are updated as new sensor results are available (as often as 20 times per second) based on the control loop of the software and the speed of the processor and senor responses. When the utility function values are updated, the decompositions that include the requirements with updated utility function values are checked for incompleteness. That is, the decompositions of the requirements specification are evaluated and re-evaluated for incompleteness throughout the execution of the system in which the generated run-time monitoring code is deployed.

5.3.4 Limitations

The accuracy of *Lykus* is only as good as the accuracy of the **ENV**, **MON**, and **REL** properties used to define the utility functions. Additionally, counterexamples may be present in a scenario that occurs only between sensor readings. In such cases, the values calculated by the utility functions never represent an incompleteness or inconsistency and, therefore, no counterexample is detected.

5.4 Run-Time Examples

This section covers examples of an incomplete requirement decomposition and an inconsistent requirement decomposition. *Lykus* is able to generate code to use utility functions

to detect incomplete and inconsistent requirements decompositions at run time to ensure run-time relevant requirements assessment.

5.4.1 Experimental Setup

The updated utility functions and detection logic are executed during the operation of a small autonomous car. The car, shown in Figure 5.4, comprises a 16 MHz ATmega328 microcontroller and two speed controllers driving 4 motors turning 4 wheels. Distance measurements are provided by an ultrasonic sensor, speed measurements are provided by a cumulative accelerometer, and user input is provided by infrared remote control. The small autonomous car was placed into two different environmental scenarios intended to elicit the detection of both incomplete and inconsistent decompositions.

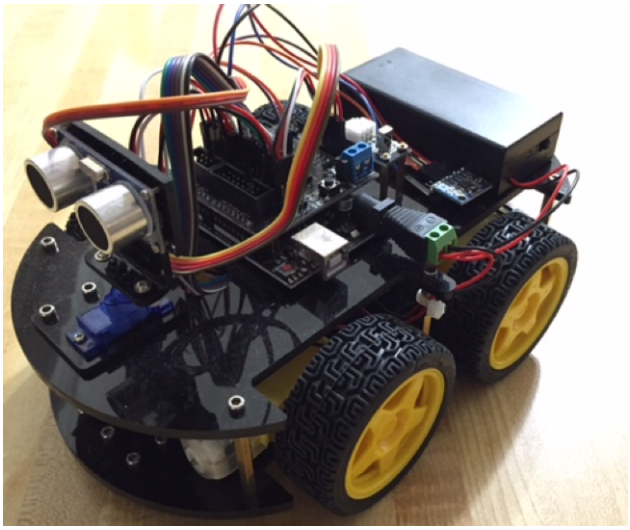


Figure 5.4: Experimental Autonomous Car

5.4.2 Incomplete Requirement Decomposition

In order to demonstrate an incomplete requirement decomposition for a parent requirement, the aggregate child requirements must be satisfied while the parent requirement itself is unsatisfied. Using goal `C.1` as an example, the car must be placed in a position where the utility function is invalid. Specifically, the speed cannot increase over some given time

despite an increase in throttle with no braking. An environmental scenario outside of the idealized environmental model (i.e., as defined in Table 5.2 for rows $Speed_t$, $Speed_{t+1}$, and *Distance*) may elicit previously undetected incomplete requirement decompositions.

In this example, we physically flip the vehicle onto its back, allowing none of the wheels to touch the ground. Since our idealized environmental model never considered the possibility of rolling the vehicle over, the standard method of accelerating does not apply. The updated utility function code generated by *Lykus* detects an incomplete decomposition from `C.1` at run time and records the incomplete decomposition, the system and environmental variables, and violated environmental assumptions.

Importantly, in order to maintain the set speed, the cruise control system *attempts* to accelerate by increasing the throttle (via goal `C.3`) while the brake is not applied (via goal `C.12`); there is a safe distance to any upcoming obstacle and the current speed is less than the desired speed (via expectation `C.2`). Despite these decomposed child goals being satisfied, the speed (both at the current time and in the future) are 0. Due to the detected incomplete decomposition, parent goal `C.1` is reported as *unsatisfied* based on run-time monitored conditions. The variables recorded for the incompleteness are shown in Table 5.5 and are limited to a resolution of 10% for percentage-based values and one tenth (i.e., 0.1) for all other values due to the truncation of sensor and actuator resolution to minimize oscillation and measurement error.

Given the list of environmental assumptions defined in Table 5.2, the values of the counterexample variables can be used to identify violated environmental assumptions at run time. Table 5.6 includes the list of environmental assumptions and indicates if they are valid or not. It is important to note that if the optional design-time completeness detection was not performed by *Ares*, there may be no environmental assumptions documented. In this case, Table 5.6 shows that rows 2 and 3 both include violated environmental assumptions related to the calculation of speed from *only* the brake and throttle, ignoring the possibility of outside influences (e.g., rollovers). This information can be logged for analysis of failures

Table 5.5: Incomplete Decomposition Values

Row	Variable or Goal	Value
1	Brake Actuator	*MIN*
2	Brake Pedal Sensor	*MIN*
3	Desired Speed	*MAX*
4	Distance	*MAX*
5	Distance Sensor 1	*MAX*
6	Distance Sensor 2	*MAX*
7	Safe Distance	6 Inches
8	Speed Sensor 1	*MIN*
9	Speed Sensor 2	*MIN*
10	Speed_t	*MIN*
11	Speed_{t+1}	*MIN*
12	Throttle Actuator	80%
13	Throttle Pedal Sensor	70%
14	Goal `C.1`	Unsatisfied
15	Goal `C.2`	Satisfied
16	Goal `C.3`	Satisfied
17	Goal `C.12`	Satisfied

Table 5.6: Environmental Assumptions: Incompleteness

Row	Environmental Assumption	Valid
1	$Speed_t = Speed\ Sensor\ 1 \vee Speed_t = Speed\ Sensor\ 2$	True
2	$Speed_t = max(MIN,$ $Throttle\ Pedal\ Sensor - Brake\ Pedal\ Sensor)$	False
3	$Speed_{t+1} = max(MIN,$ $Throttle\ Actuator - Brake\ Actuator)$	False
4	$Distance = Distance\ Sensor\ 1 \vee$ $Distance = Distance\ Sensor\ 2$	True

after they occur, or used as inputs to existing adaptation methods beyond the scope of this chapter.

5.4.3 Inconsistent Requirement Decomposition

In order to demonstrate an inconsistent requirements decomposition, the speed must increase despite not increasing the throttle. We achieve this case by allowing the vehicle to drive off a 'cliff,' represented by the edge of a desk. The car accelerates through its fall

Table 5.7: Inconsistent Decomposition Values

Row	Variable or Goal	Value
1	Brake Actuator	*MIN*
2	Brake Pedal Sensor	*MIN*
3	Desired Speed	2 MPH
4	Distance	*MAX*
5	Distance Sensor 1	*MAX*
6	Distance Sensor 2	*MAX*
7	Safe Distance	6 Inches
8	Speed Sensor 1	1.2 MPH
9	Speed Sensor 2	1.2 MPH
10	Speed_t	1.2 MPH
11	Speed_{t+1}	2 MPH
12	Throttle Actuator	20%
13	Throttle Pedal Sensor	20%
14	Goal `C.1`	Satisfied
15	Goal `C.2`	Unsatisfied
16	Goal `C.3`	Unsatisfied
17	Goal `C.12`	Satisfied

despite not increasing the throttle. Just as with the incomplete requirements decomposition, the updated utility function code generated by *Lykus* detects an inconsistent decomposition from `C.1` and records the inconsistent decomposition (i.e., the parent requirement is satisfied in a method outside of what is specified by the decomposed requirements), the system and environmental variables, and violated environmental assumptions.

In this case, the ACC system accelerates without increasing the throttle (via goal `C.3`) while the brake is not applied (via goal `C.12`). While there is a safe distance to any upcoming obstacle, the current speed is not less than the desired speed (via expectation `C.2`). Despite not satisfying these decomposed child goals, the speed increases due to the precipitous drop. Due to the detected inconsistent decomposition, goal `C.1` is reported as *unsatisfied*, as it is not satisfied in the method required by the decomposed child requirements. The variables recorded for the inconsistency are shown in Table 5.7 and are limited to a resolution of 10% for percentage based values and one tenth (i.e., 0.1) for all other values due to the truncation of sensor and actuator resolution to minimize oscillation and measurement error.

Given the list of environmental assumptions previously identified in Table 5.2, Table 5.8 lists the environmental assumptions and indicates their run-time validity. Similar to the incompleteness counterexample, rows 2 and 3 both include violated environmental assumptions related to the calculation of speed from *only* the brake and throttle, ignoring the possibility of outside influences (e.g., drastic changes in terrain). The system designer may choose to adapt to solutions that do not depend on the invalid environmental assumptions, or the invalid environmental assumptions may be recorded for later revision of the system.

Table 5.8: Environmental Assumptions: Inconsistency

Row	Environmental Assumption	Valid
1	$Speed_t = Speed\ Sensor\ 1 \vee Speed_t = Speed\ Sensor\ 2$	True
2	$Speed_t = max(MIN,$ $Throttle\ Pedal\ Sensor - Brake\ Pedal\ Sensor)$	False
3	$Speed_{t+1} = max(MIN,$ $Throttle\ Actuator - Brake\ Actuator)$	False
4	$Distance = Distance\ Sensor\ 1 \vee$ $Distance = Distance\ Sensor\ 2$	True

5.4.4 Threats to Validity

Since *Lykus* is most realistically validated at run time, rather than using a simulation or static analysis, the validation is limited to the finite set of scenarios that occur. We have ensured that both inconsistent and incomplete requirements decompositions can be identified, however additional inconsistent and incomplete requirements decompositions may exist within the requirements model. The validation of the detection classification, however, is done by specific cases (i.e., Table 5.4) within the description of the approach. Additionally, methods that could manage sensor uncertainty (e.g., RELAXed goals and requirements [94]) are not included in the examples of incomplete and inconsistent requirements.

5.5 Run-Time Related Work

This section overviews related work, including requirements completeness and run-time monitoring of requirements. Unlike proposals to use run-time specific models to support run-time analysis [30], we use goal models intended for design time use for run-time analysis.

5.5.1 Requirements Completeness

Multiple methods have been developed to identify incomplete requirements decomposition. We overview these strategies and compare them to *Lykus*. Obstacles to requirements completeness have been generated using search-based techniques [2], however the counterexamples must be manually reviewed for applicability. Formal methods of guaranteeing complete requirements exist for behavioral state-based systems [56], formally described requirements using theorem provers [87], and low-level functional details [56]. Additionally, decomposition with formally-proven completeness properties also exist, where a system defined by repeated application of the formal decomposition patterns guarantees completeness. Problematically, formal methods are intrinsically heavyweight solutions that often require expertise with theorem proving [87] or limit possible solutions to the formally defined patterns.

Tools also exists that identify single [37] or multiple [38] counterexamples (*Ares* and *Ares*-EC, respectively) in hierarchical requirements decompositions at design time without restrictions on decomposition patterns or heavyweight formal descriptions and analysis.

Lykus differs by acknowledging that invalid environmental assumptions made at design time may leave incomplete decompositions intact until they occur at run time. *Lykus* detects these incomplete decompositions at run time rather than at design-time and provides more accurate utility function assessment in the presence of incomplete decompositions.

5.5.2 Run-Time Monitors

Several frameworks exist to monitor requirements at run time [45, 47, 78] that are intended to support instrumentation, diagnosis, and reconfiguration operations within the system. The utility functions whose values we adapt in *Lykus*, as generated by Athena [75], provide the same support with the benefit of automatic generation from a set of environmental properties. *Lykus* differs from existing run-time monitors by detecting incomplete and inconsistent requirements at run time and adapting the results of existing run-time monitors [75] to provide more accurate results.

5.6 *Lykus* Conclusion

In this chapter, we have presented *Lykus*, a run-time approach for detecting incomplete and inconsistent requirements decomposition and using that information to identify invalid environmental assumptions and unsatisfied requirements. *Lykus* reports the issues detected with requirements, as well as the invalid environmental assumptions, while automatically updating the satisfaction results of requirements monitoring.

We demonstrate *Lykus* on an adaptive cruise control system developed in collaboration with our industrial collaborators and implemented on a robotic vehicle. We show that *Lykus* is able to detect incomplete and inconsistent requirements at run time due to invalid environmental assumptions. Specifically, we show that while existing tools identify no incomplete requirements, incomplete decompositions may still exist due to incorrect assumptions that are detectable using run-time monitors.

In the future, we plan to apply *Lykus* to additional examples, including requirements models with RELAXed requirements [94] that make use of fuzzy logic in specifying requirements. Additionally, we plan to investigate how *Lykus* can be used to trigger additional mitigations at run time due to unsatisfied requirements that would not be detected with other methods.

Chapter 6

Detecting Feature Interactions: Using Symbolic Analysis

Independently-developed features often exhibit overlapping, yet conflicting behavior termed *feature interactions*. Detecting unwanted feature interactions amongst even a moderate number of features can involve analysis of an exponential number of possible interactions. A potentially human resource-intensive step is the subsequent effort needed by the system designer to assess each detected interaction. This chapter introduces *Phorcys*, a symbolic analysis, design-time approach for detecting unwanted n-way feature interactions and determining their causes at the requirements level. Unlike previous n-way feature interaction detection approaches that attempt to enumerate every set of interacting features, *Phorcys* analyzes each feature for its ability to cause an interaction with other features, thus reducing designer assessment effort to be linear with respect to the number of features. To the best of the authors' knowledge, *Phorcys* is the only technique to detect both the existence of n-way feature interactions and identify the respective cause. We illustrate our approach by applying *Phorcys* to an industry-based automotive braking system comprising multiple subsystems.

6.1 Introduction

Despite ongoing research since the 1980s, feature interactions (FIs) still pose a significant challenge to modularity and assurance in software development and design [4]. While heavily researched within the telecommunications domain, the increasing complexity and number of features in onboard and cyber-physical systems has created a new generation of challenges [4]. Detecting feature interactions is challenging due to the exponential growth of potential interactions with respect to the number of features [17] and the range of environmental possibilities in cyber-physical systems. More formally, when defining a feature interaction by the minimal set of features that are necessary for the interaction to occur, the number of possible interactions is $\mathcal{O}(2^{|F|})$, where F is the set of features [5]. The development of additional features is ultimately overwhelmed by integrating features and managing the feature interaction problem [18], thus making it impractical for a system designer to assess all possible feature combinations that cause an interaction. The growth in the number possible feature interactions and consequently the number of possible feature interactions that must be assessed can result in latent behavior that impacts the system dependability at run-time. This problem is particularly prominent in cyber-physical systems where unexpected adverse environmental conditions may occur and impact the dependability of the system. This chapter presents *Phorcys*,[1] a method for symbolically detecting each feature that causes an n-way feature interaction in a requirements goal model, thus reducing the system designer's effort to assessing only as many interactions as there are features.

In order to address the computational intractability of feature interaction detection, many researchers have focused on pair-wise interactions [10, 21, 41, 59]. Rather than detecting interactions amongst any number of features, only pairs of features are analyzed. While this approach decreases the number of potential interactions detected to $\mathcal{O}(|F|^2)$, interactions that only emerge when three or more features are present will not be detected [5]. In typical development projects, where features are added incrementally [20], pair-wise fea-

[1] *Phorcys* (pronounced 'forsis') is the Greek god of hidden dangers in the deep.

ture interaction analysis only assesses the newly added feature paired with each existing feature. Given that a recent study found interactions greater than pair-wise in every system analyzed [5], n-way interaction detection techniques are also needed.

This chapter presents *Phorcys*, a method for symbolically analyzing a feature, represented in a goal model, to detect an n-way feature interaction. Rather than analyze every combination of features for an interaction, *Phorcys* analyzes each individual feature for its ability to *cause* an interaction.[2] For example, consider 3 features, $\texttt{F}_1$, $\texttt{F}_2$, and $\texttt{F}_3$. If an interaction exists, then at least one feature ($\texttt{F}_1$, $\texttt{F}_2$, or $\texttt{F}_3$) no longer operates as it did independently. When the inclusion of $\texttt{F}_1$ is impacted by the functionality of $\texttt{F}_2$, $\texttt{F}_3$, or $\texttt{F}_2$ and $\texttt{F}_3$ together, then $\texttt{F}_1$ is the *cause* of the feature interaction. This relationship is not exclusive, if $\texttt{F}_1$ impacts the functionality of $\texttt{F}_3$, it may also be true that $\texttt{F}_3$ impacts the functionality of $\texttt{F}_1$. This concept of the *cause* of the interaction scales to any number of features. Any feature under analysis can be the cause of an interaction if it is impacted by any features in the set of features.

Phorcys analyzes features represented in goal models that hierarchically decompose a high-level goal down to individual requirements [87]. *Phorcys* analyzes each feature by symbolically representing the possible feature combinations of the goal model and uses a constraint solver to check for the existence of any combination of features where the analyzed feature causes conflict in one or more requirements to be satisfied. As such, *Phorcys* is able to check for many feature interaction possibilities in a single analysis. Where previous n-way feature interaction detection methods may present an intractably large (exponential) set of automatically detected interactions for the system designer to assess manually for causes [18, 20], *Phorcys* only presents features that *cause* an interaction and a corresponding counterexample scenario for each of those interactions.

The contributions of this chapter are as follows:

[2] *Phorcys* identifies one counterexample for each feature that causes an interaction (e.g., the number of counterexamples to be reviewed by the designer is linear with respect to the number of features). This does not imply a linear run time, as each analysis may be computationally expensive.

- We introduce a new symbolic approach for analyzing requirements to detect unintended n-way feature interactions and the features that cause them, including proofs of soundness and completeness,
- We present *Phorcys*, a prototype implementation of the approach, and
- We demonstrate the applicability of *Phorcys* on an industry-based application of an automotive braking system based in part on the feature interaction issues in the 2010 Toyota Prius [29, 63], a hybrid vehicle. Analysis shows that *Phorcys* is able to identify feature interaction causes in the braking system goal model, involving two or more features.

The remainder of this chapter is organized as follows. Section 6.2 describes how feature interactions can be formalized, including the *cause* of feature interactions while Section 6.3 covers proofs of soundness and completeness for the formalization. Section 6.4 introduces the *Phorcys* approach to modeling features in goal models. Section 6.5 describes the *Phorcys* process and gives technical details of the detection methods. Section 6.6 provides results of the *Phorcys* process applied to the braking system goal model. Sections 6.7 and 6.8 present related work and a summary, respectively.

6.2 Formally Specifying Feature Interactions

This section formally defines pairwise feature interactions based on previous work [20] and extends the formalization to n-way interactions. The n-way feature interaction problem is then refined in terms of requirement specifications and detecting feature interaction causes.

6.2.1 Standard Feature Interactions

Previously, Calder, et al. [20] described a feature interaction between two features (F_1 and F_2) and their respective properties (ϕ_1 and ϕ_2). The description is as follows. A feature,

F_1, satisfies a property, ϕ_1, denoted as $F_1 \models \phi_1$. Features may be combined, or composed, via a composition operator ('$\oplus$'). Since decomposed goals and features in a goal model are satisfied or unsatisfied in parallel, the composition operator (i.e., '$\oplus$') is a parallel composition operator. Since requirements in goal models are declarative and must be satisfied in parallel to satisfy the root goal, we only consider parallel composition. Future work may include sequential composition. For example, features F_1 and F_2, may be composed as $F_1 \oplus F_2$. When $F_1 \models \phi_1$ and $F_2 \models \phi_2$, the expected composition of F_1 and F_2 should satisfy the conjunction of their respective properties, that is, $F_1 \oplus F_2 \models \phi_1 \wedge \phi_2$. However, if the composition of the features does not satisfy the conjunction of their respective properties, then there exists a feature interaction [20] as shown in Equation 6.1:

$$F_1 \oplus F_2 \not\models \phi_1 \wedge \phi_2. \tag{6.1}$$

6.2.2 N-Way Feature Interaction Extension

This 2-way interaction concept can be extended to represent n-way interactions where any number of composed features result in the satisfaction of their aggregate properties:

$$\bigoplus_{i=1}^{|F|}(F_i) \not\models \bigwedge_{i=1}^{|F|}(\phi_i), \tag{6.2}$$

where F is a set of features that satisfies any overall feature composition requirements and includes features in the full feature set, S, that contains all features of the system-to-be (i.e., $F \subseteq S$).

6.2.3 Requirements-Based Formalization of Feature Interactions

A feature can be defined by its pre-conditions, implementation, and post-conditions [16]. Since the implementation is unknown (i.e., an unimplemented requirement) but assumed to fulfill the specification (i.e., pre- and post-conditions), each feature F_i, where $F_i \in F$

satisfies a property (i.e., ϕ_i), measured as a post-condition such that when the pre-condition is satisfied we expect the post-condition to be satisfied. For a given feature F_i, where $F_i \in F$:

$$P_i \implies R_i, \tag{6.3}$$

where P_i is the precondition for feature F_i, P is the set of all pre-conditions for the system under development (i.e., the system-to-be), R_i is the post-condition for feature F_i, and R is the set of post-conditions. Since we are performing our analysis at the requirements level (rather than implementation), we reason about feature implementation indirectly by assuming that if the pre-conditions are satisfied, then the feature is satisfied. If the feature is satisfied, then the post-conditions of the feature must be satisfied unless there is a conflict between the post-conditions (i.e., a feature interaction between any possible implementations of the features that fulfills the specification). Therefore, the composition of features (conjunction of pre-conditions) fulfills their properties (conjunction of post-conditions) when no feature interaction exists:

$$\bigwedge_{i=1}^{|F|}(P_i) \implies \bigwedge_{i=1}^{|F|}(R_i). \tag{6.4}$$

We detect feature interactions when the properties (conjunction of post-conditions) are not met since no implementation could possibly satisfy conflicting post-condition specifications, leaving at least one feature post-condition unsatisfied. As such, the complement of Equation 6.4 used to detection feature interactions is as follows:

$$\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg \bigwedge_{i=1}^{|F|}(R_i). \tag{6.5}$$

That is not to say that Equation 6.5 is identical to Equation 6.2. Equation 6.2 defines a feature interaction where the implementation of a feature is specified (e.g., FI in code), while Equation 6.5 defines a feature interaction given a specification and assuming an imple-

mentation (e.g., FI in requirements). While Equation 6.2 may detect a feature interaction between two implementations, Equation 6.5 would detect conflicting specifications.

6.2.4 *Phorcys* Feature Interaction Causes

Features in *Phorcys* are specified as a collection of pre-conditions (P) and post-conditions (R) for each feature in the set of features (F). The composition of features and their decomposed components is parallel. Sequential composition and temporal properties are not considered. When pre-conditions (P) are satisfied, the presumed implementation is expected to satisfy the post-conditions (R) (i.e., Equation 6.3) representing the properties (ϕ). We define a feature interaction in the *Phorcys* process to be a parallel composition of features where one feature (F_{cause}) *causes* the interaction by not satisfying its respective property (ϕ_{cause}) assessed by its post-condition (R_{cause}). The *cause* is individually effected such that it *causes* the system as a whole (i.e., the root goal) to be unsatisfied. That is:

- We consider a feature to be satisfied if its pre-conditions are satisfied (i.e., the feature would be executed in an implemented system),
- We expected the properties of a feature, as measured by the post-conditions, to be satisfied when the feature would be executed in an implemented system (i.e., the pre-conditions are satisfied), and
- When the post-conditions cannot be satisfied due to a conflict with other post-conditions (i.e., two or more features are specified to fulfill conflicting properties) we consider a feature interaction to have occurred.

In logical terms, we define Expression 6.6 as a feature interaction for some set of features, F, such that the features in F satisfy the compositional requirements of the system (e.g., the features in F satisfy the top-level goal in a goal model) and $F_{\text{cause}} \in F$:

$$\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg R_{\text{cause}}. \tag{6.6}$$

Equation 6.6 is based on the formal definition for n-way feature interaction detection in Equation 6.5; however, instead of detecting unsatisfied post-conditions in any feature (i.e., $\neg \bigwedge_{i=1}^{|F|}(R_i)$), Equation 6.6 identifies unsatisfied post-conditions in a single feature under analysis (i.e., $\neg R_{\text{cause}}$) as the feature interaction *cause*.

6.3 Proofs of Soundness and Completeness

The feature interaction detection logic defined in the *Phorcys* process is functionally equivalent to existing definitions that do not explicitly detect the *cause* of the feature interaction (i.e., Equation 6.5). We assume a subset of features that satisfies the compositional requirements of the system. In the case of *Phorcys*, the subset of features is defined by the goal model decomposition of the features.

The reasoning behind completeness (i.e., no missing interactions) and soundness (i.e., no superfluous interactions detected) is straightforward. Given definitions for feature interaction detection in Equation 6.5 and Equation 6.6, for *Phorcys* feature interaction detection, a single post-condition is both necessary and sufficient to cause the entire conjunction of post-conditions to be unsatisfied in the presence of a satisfied set of pre-conditions that are assumed to satisfy their properties (i.e., post-conditions), unless otherwise constrained. The proofs of completeness and soundness prove a single post-condition is both necessary and sufficient to be true.

Completeness Theorem

For the purposes of feature interaction detection, completeness is defined such that every feature interaction that is detected via existing methods (i.e., Equation 6.5) is detected via the *Phorcys* method (i.e., Equation 6.6). Therefore, if *Phorcys* feature interaction detection is complete, any feature interaction detected in Equation 6.5 implies a feature interaction is detected in Equation 6.6:

$$\left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg \bigwedge_{i=1}^{|F|}(R_i)\right) \Rightarrow \left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg R_{\text{cause}}\right). \tag{6.7}$$

The completeness proof below shows that in any case where a feature interaction is detected (i.e., the logical conjunction of post-conditions of some set of features are not satisfied) then at least one of the features must be the cause of the interaction (i.e., at least one of the post-conditions must not be satisfied).

Proof. A proof by contradiction will be used. We assume the completeness equation in Equation 6.7 is false. That is:

$$\neg\left(\left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg \bigwedge_{i=1}^{|F|}(R_i)\right) \Rightarrow \left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg R_{\text{cause}}\right)\right). \tag{6.8}$$

Applying logical simplification and converting implications to their logical structures yields:

$$\left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg \bigwedge_{i=1}^{|F|}(R_i)\right) \wedge \left(\neg \bigwedge_{i=1}^{|F|}(P_i) \vee R_{\text{cause}}\right). \tag{6.9}$$

i.) **In the case where R_{cause} is false**, '$\neg \bigwedge_{i=1}^{|F|}(P_i)$' and '$\bigwedge_{i=1}^{|F|}(P_i)$' must both be true, which is a contradiction. Therefore R_{cause} cannot be false.

ii.) **In the case where R_{cause} is true**, '$\bigwedge_{i=1}^{|F|}(P_i)$' and '$\neg \bigwedge_{i=1}^{|F|}(R_i)$' must also be true. Given the assumption that each feature satisfies its property unless otherwise constrained ($P_i \implies R_i$ for every $F_i \in F$ due to Equation 6.3) there must exist a R_{cause} that is not true for '$\neg \bigwedge_{i=1}^{|F|}(R_i)$' to be true. Therefore R_{cause} cannot be true for all features if '$\neg \bigwedge_{i=1}^{|F|}(R_i)$' is true.

Since R_{cause} cannot be false and it cannot be true for all features when a FI is detected (e.g., $\neg \bigwedge_{i=1}^{|F|}(R_i)$), then *Phorcys* cannot be incomplete. □

Soundness Theorem

For the purposes of feature interaction detection, soundness is defined such that every feature interaction that is detected via the *Phorcys* method is also detected using existing methods (i.e., Equation 6.5). Therefore, if *Phorcys* feature interaction detection is sound, any feature interaction detected in Equation 6.6 implies a feature interaction is detected in Equation 6.5:

$$\left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg R_{\text{cause}}\right) \Rightarrow \left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg \bigwedge_{i=1}^{|F|}(R_i)\right). \tag{6.10}$$

The soundness proof below shows that in any case where at least one of the features is the cause of a feature interaction (i.e., at least one of the post-conditions must not be satisfied) then a feature interaction must also be detected (i.e., the logical conjunction of post-conditions of some set of features are not satisfied).

Proof. A proof by contradiction will be used. We assume the soundness equation in Equation 6.10 is false. That is:

$$\neg\left(\left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg R_{\text{cause}}\right) \Rightarrow \left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg \bigwedge_{i=1}^{|F|}(R_i)\right)\right). \tag{6.11}$$

Applying logical simplification and converting implications to their logical structures yields:

$$\left(\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg R_{\text{cause}}\right) \wedge \left(\neg \bigwedge_{i=1}^{|F|}(P_i) \vee \bigwedge_{i=1}^{|F|}(R_i)\right). \tag{6.12}$$

i.) **In the case where $\bigwedge_{i=1}^{|F|}(R_i)$ is false**, '$\neg \bigwedge_{i=1}^{|F|}(P_i)$' and '$\bigwedge_{i=1}^{|F|}(P_i)$' must both be true, which is a contradiction. Therefore '$\bigwedge_{i=1}^{|F|}(R_i)$' cannot be false.

ii.) **In the case where $\bigwedge_{i=1}^{|F|}(R_i)$ is true**, '$\bigwedge_{i=1}^{|F|}(P_i)$' and '$\neg R_{\text{cause}}$' must also be true.

Since $R_{\text{cause}} \in R$ and $' \bigwedge_{i=1}^{|F|}(R_i)$' are both satisfied it is not possible for expression $\neg R_{\text{cause}}$ to be true.

Since $\bigwedge_{i=1}^{|F|}(R_i)$ cannot be false and true at the same time, then *Phorcys* cannot be unsound. □

Given the proofs of both soundness and completeness, it is true that the *Phorcys* feature interaction cause detection is equivalent (i.e., is capable of detecting the same feature interactions) to feature interaction detection with the addition of explicitly identifying the cause of the interaction.

6.4 Goal-Based Modeling of Features

This section details the *Phorcys* approach to modeling features within goal models. This section includes our annotations for features, pre- and post-conditions, as well as an example braking system goal model that is used throughout the remainder of the dissertation.

6.4.1 Features

Goal Models [87] are notationally extended in this work to include the concept of features. While a goal model defines the system-to-be [48], we add an additional notation, '`(f)`,' to denote specific goals and their decompositions representing features within the system. For example, the braking system goal model in Figure 6.1, goals `B`, `C`, `D`, and `E` are annotated with an `(f)` as high-level goals for four different features. While the additional notation to identify features adds information about the system-to-be, it does not change the underlying semantics of a goal model. Any standard goal model can be annotated to identify features, or used as a standard goal model without the additional annotation.

6.4.2 Pre- and Post-Conditions

In this work, we use expectations to define the expected environmental properties that exist *before* and *after* the fulfillment of a requirement. Since elements of the system are used to satisfy a requirement, the element performs some action that causes an observable effect. The expression of these changes is measurable via elements in the environment, and are therefore specified as *expectations*. We differentiate between expectations of the environment *before* the requirement's satisfaction, or *pre-conditions*, and those that are expressed *after* the requirement satisfaction, or *post-conditions*. The use of expectations in this manner is similar to awareness requirements that document domain assumptions [81, 83] and expectations in KAOS goal modeling [87]. Pre- and post-conditions are explicitly included only when necessary to ensure satisfaction of the unique constraints of a requirement, and are shared amongst AND-decomposed siblings. We distinguish these expectations in goal models by the annotations '`(pre)`' and '`(post)`', respectively. For example, in Figure 6.1, expectations `B.3` and `B.1` are annotated as pre- and post-conditions, respectively. The *pre-condition* `B.3` ensures only positive brake forces can be applied via requirement `B.2`. Similarly, the *post-condition* `B.1` indicates that after requirement `B.2` has been satisfied, the standard brake force (SF) is equal to the commanded brake force (CBF). Expectations may expect properties that are changed by the actions of the agent responsible for a requirement. For example, an expectation may be that a car must be in Park to allow a gear shift via a requirement from Park to Drive. However, after the requirement (shift from Park to Drive), the expectation is no longer satisfied. Just as an expectation may be expected *before* a requirement, it may be expected *after* as well. For example, another expectation may be that the car must be in Drive after having shifted from Park to Drive. One expectation is measured before the requirement, and one after. These expectations do not necessarily conflict if one measures the state of the car's transmission now, and one measures state of the car's transmission later. Similar to the feature annotation, this annotation provides additional information about the

system-to-be that could be applied to any goal model or removed from an annotated goal model without consequence on standard goal model semantics.

6.4.3 Braking System Goal Model Example

The goal model in Figure 6.1 represents an automotive braking system with several features. In general, a feature reads the actual brake pedal position and relates that information to a commanded brake force that is translated to an external braking force via a braking mechanism. However, for this system, the braking system has been developed as four individual features (i.e., `B`, `C`, `D`, `E`) rather than as a single, monolithic system. The use of multiple brake features is based on a known 2010 Toyota Prius braking system issue [29]. The braking system features defined here have been independently developed without respect for each other and is intended to be a realistic example of industrial design artifacts at the requirements level, based on our collaboration with automotive industrial practitioners.[3] Specific ratios used in the braking system goal model are commonly included in high-level specifications regarding input and output limitations of line-replaceable components that interface with the system under development.

The features included are standard force braking, regenerative braking, continuous braking, and anti-lock brakes, where the target vehicle is a hybrid (i.e., contains both an electric motor and gas engine). Acronyms used are: PBF (**P**revious **B**rake **F**orce), CBF (**C**ommanded **B**rake **F**orce), SS (**S**lip **S**ensor), RF (**R**egenerative Brake **F**orce), SF (**S**tandard Brake **F**orce), and BF (**B**rake **F**orce). Each of these features was specified without specific regard for the other features, and operates as described below.

Feature `B`: *'Achieve(Standard Force Braking)'* applies (via `B.2`) a brake force through standard force brake methods (e.g., disk brakes on the wheels) based on a commanded brake force using the *Hydraulic Brake Actuator*.

Feature `C`: *'Achieve(Regen Braking)'* applies a brake force using both standard brake

[3]Collaboration with automotive industrial practitioners included reviewing our goal modeling approach to automotive systems, however the specific braking system presented here was not explicitly reviewed.

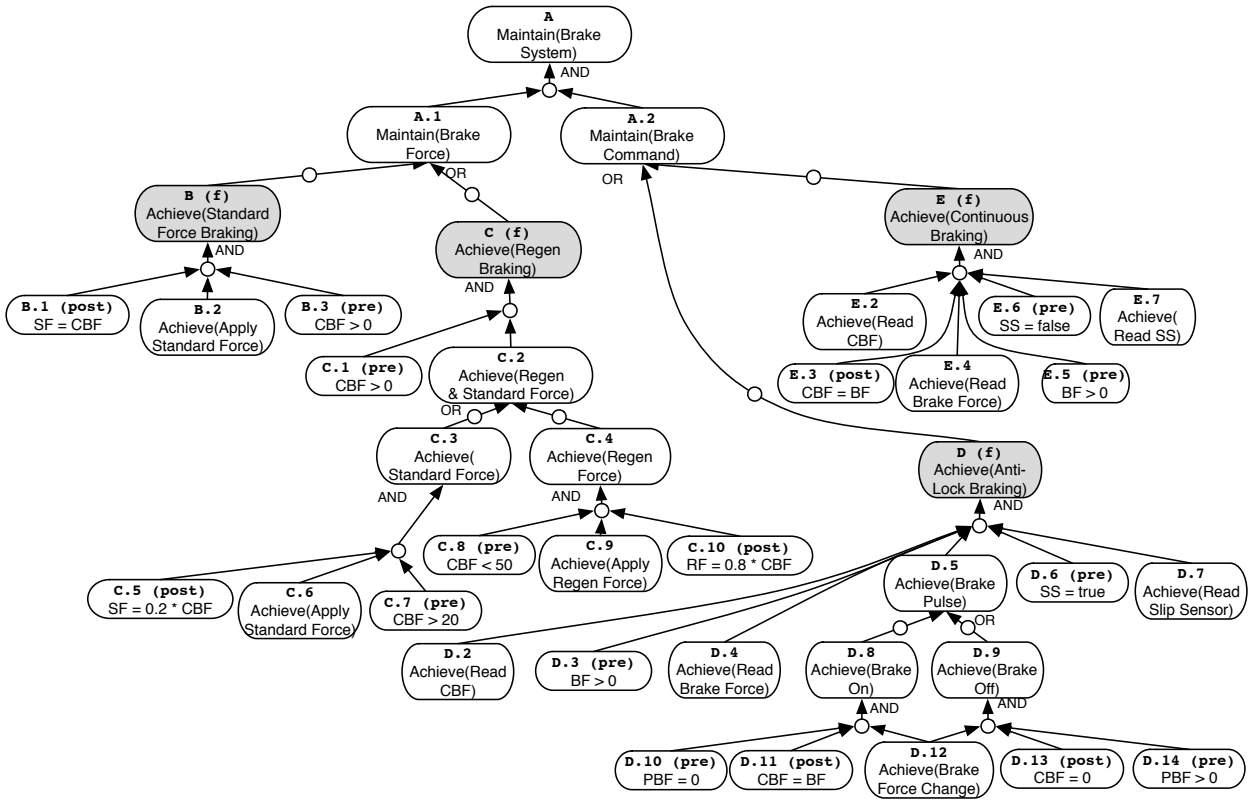


Figure 6.1: Braking System Goal Model

force methods (via C.3) as well as regenerative braking methods (via C.4) that help capture electrical energy from braking. Due to limitations in the amount of braking force that regenerative braking may provide, three combinations are used for braking: regenerative force is used alone for low-braking force needs; a combination of both is used when a mid-range braking force is needed; and standard braking force is used when a large amount of braking force is needed. In all cases, the amount of braking force is dependent on the commanded brake force and is applied using the *Hydraulic Brake Actuator* and/or the *Regeneration Brake Actuator*, depending on the *Commanded Brake Force* value.

Feature D: *'Achieve(Anti-Lock Braking)'* reads the current brake force from the *Brake Pedal Sensor* (D.4) and alternates applying it to the *Commanded Brake Force* (D.8) or applying no commanded brake force (D.9) when the *Slip Sensor* detects slippage of the wheels on the driving surface (D.6 and D.7).

Feature E: *'Achieve(Continuous Braking)'* reads (E.4) the current brake force from

the *Brake Pedal Sensor* and applies it to the *Commanded Brake Force* (`E.2`) when the *Slip Sensor* does not detect any slippage (`E.7`).

These features either *command* the brake force (e.g., `D`, `E`) or physically *apply* the brake force (e.g., `B`, `C`). In order to satisfy the top-level braking goal (`A`), there must be a commanded and applied brake force. Features `D` and `E` are mutually exclusive due to pre-conditions `E.6` (SS = false) and `D.6` (SS = true), since both `E.6` and `D.6` cannot both be satisfied simultaneously. Therefore, due to the AND decomposition of both goal `D` and goal `E`, if either pre-condition `D.6` or `E.6` is unsatisfied, then the entire feature is inactive. The set of features that satisfies the top-level goal is captured via the decompositions in goals `A`, `A.1`, and `A.2`.

6.5 Approach

Phorcys is a method for symbolically analyzing a goal model for n-way feature interactions that cause a conflict in the way requirements satisfy their specifications as defined by their respective pre- and post-conditions. The equations limit symbolically instantiated goal models to concrete instances that contain feature interactions. These counterexamples are generated for each feature that can cause an interaction.

6.5.1 FI Detection Process

Figure 6.2 overviews the *Phorcys* process with a DFD. As shown in Step 1 of Figure 6.2, *Phorcys* accepts a goal model defined in its entirety by the system designer (including feature and pre- and post-condition annotations), such as the one in Figure 6.1. Step 2 analyzes each feature to determine if each feature can cause a feature interaction. The *Phorcys* output from Step 3, contains the features that cause an interaction along with affected features, as well as the concrete values of the variables in the system. Next, we describe each of the steps in the DFD in turn.

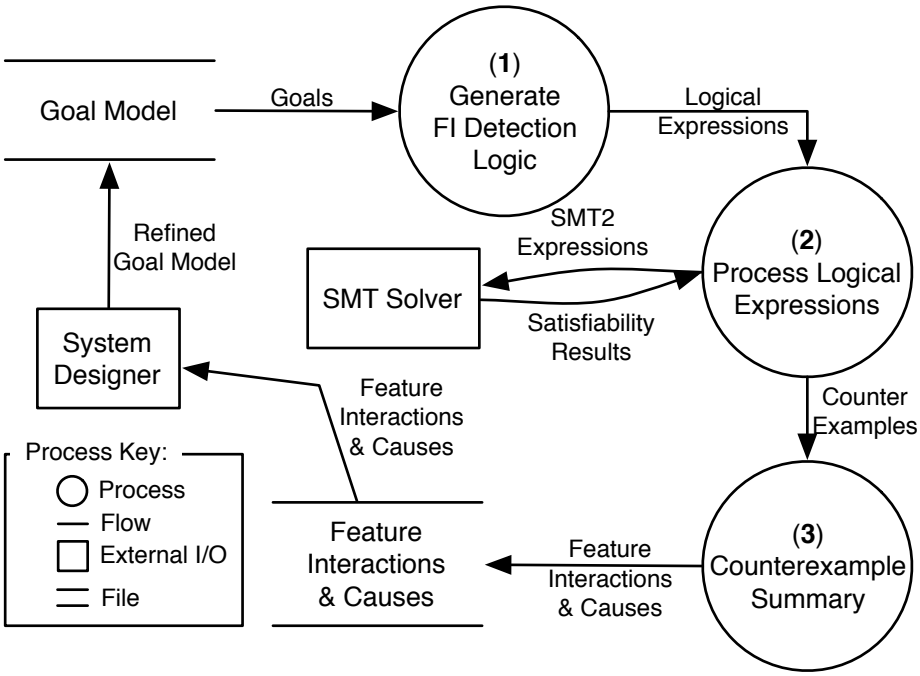


Figure 6.2: *Phorcys* Data Flow Diagram

Step (1): Generate FI Detection Logic

Phorcys feature interaction causes are detected based on generated logical expressions for each feature represented in the goal model provided by the system designer. The logical expressions are generated from the pre- and post-conditions in the goal model that are satisfied when a goal model instance is both i) valid and ii) contains properties of a feature interaction as defined within this step. The specific goal model instances are created by symbolically analyzing the logical expressions.

i) Valid Goal Configurations: *Phorcys* limits the allowable feature sets to those that satisfy the top level goal. For example, given the goal model in Figure 6.1, the pre-conditions of features that, when combined, satisfy goal `A` define the allowable feature sets:

$$(P_{\texttt{B}} \vee P_{\texttt{C}}) \wedge (P_{\texttt{D}} \vee P_{\texttt{E}}), \tag{6.13}$$

where $P_{\texttt{B}}$, $P_{\texttt{C}}$, $P_{\texttt{D}}$, and $P_{\texttt{E}}$ represent the sets of pre-conditions that refer to conditions that should be true before realization of features `B`, `C`, `D`, `E`. For example, $P_{\texttt{B}} = \texttt{B.3} = (\text{CBF} > 0)$,

since the only pre-condition in feature B is B.3. Figure 6.3 shows all possible goal configurations represented at the feature level for the braking system, where shading is used to denote satisfied goals.

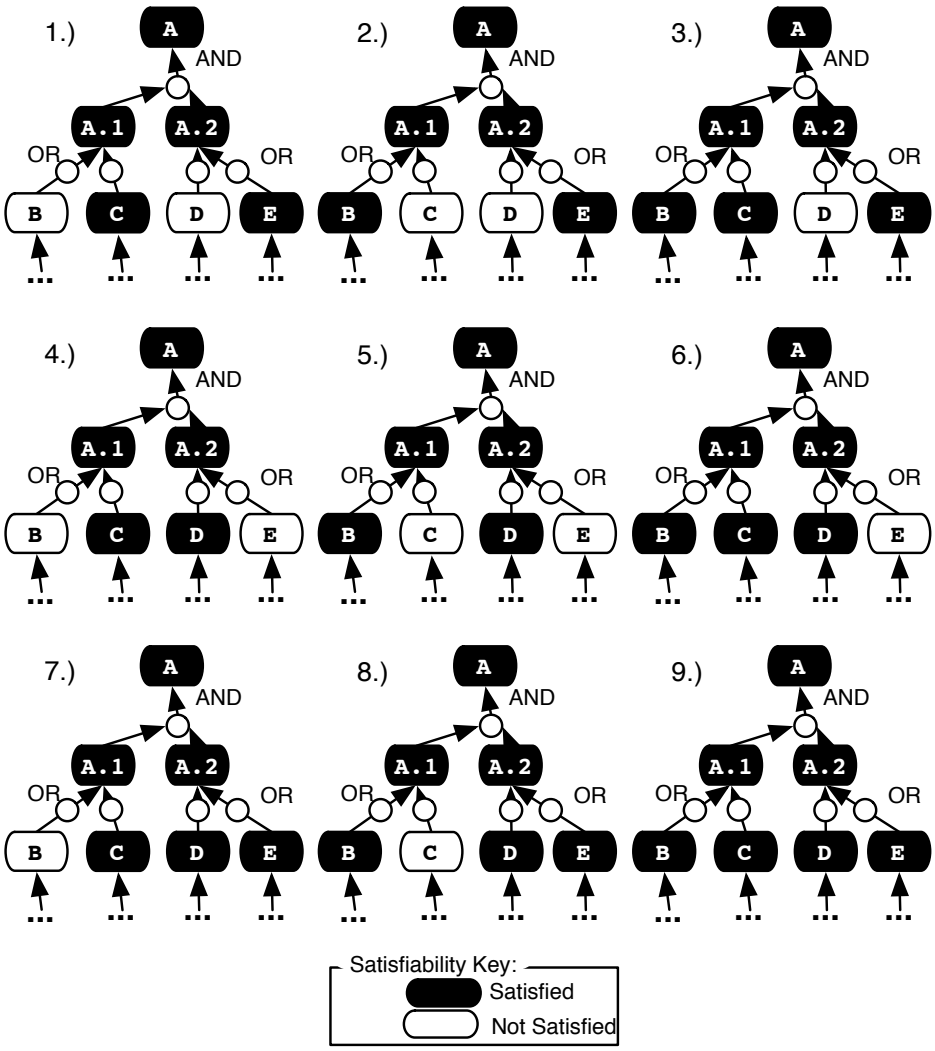


Figure 6.3: Goal Configurations that Satisfy Top-Level Goal

It is important to note that while all the goal configurations shown in Figure 6.3 would satisfy the top level goal (i.e., goal A) at a feature level, not all of those goal configurations are valid and would satisfy Equation 6.13. For example, a goal model configuration with both E.6 and D.6 cannot be feasible since E.6, $SS = false$ while D.6, $SS = true$, and expression $(SS = false) \wedge (SS = true)$ cannot be satisfied. This means that in Figure 6.3, goal configurations that have both D and E satisfied simultaneously are not valid (i.e., goal

configurations 7, 8, and 9). For example, goal configuration 2 in Figure 6.3 must be fully satisfied as shown in Figure 6.4.

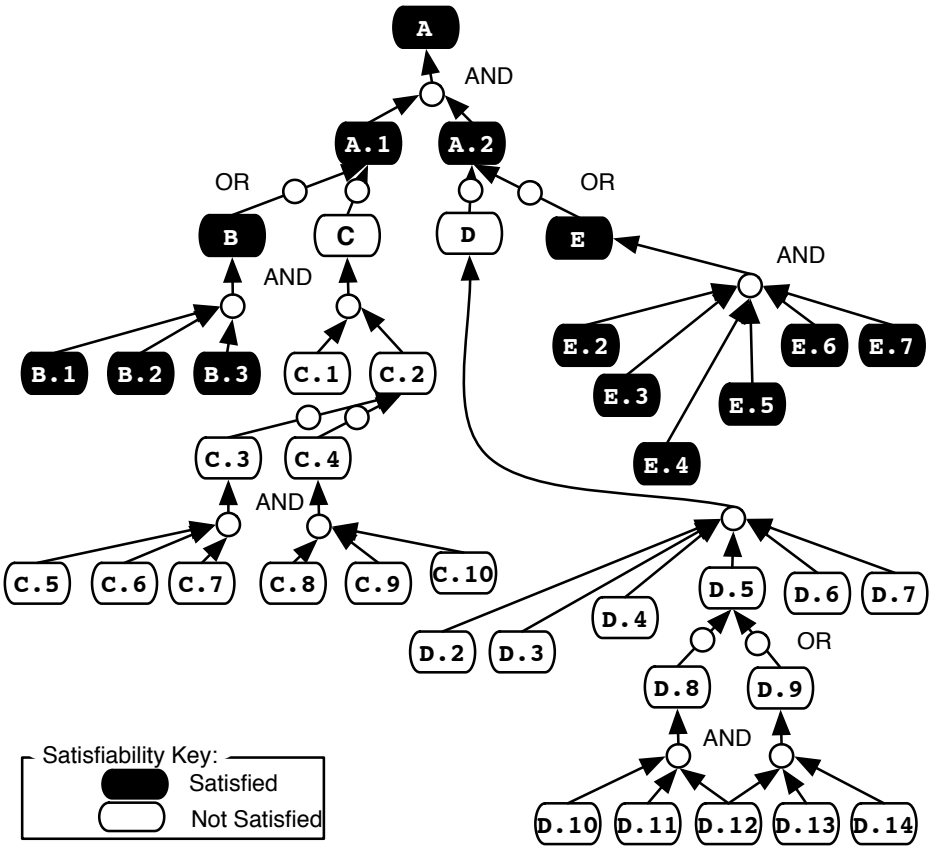


Figure 6.4: Goal Configuration 2 in Figure 6.3

ii) Feature Interaction Properties: Feature interactions are only detected in goal configurations where the top-level goal would be satisfied if it were not for the interaction. For example, when analyzing feature B as the cause (i.e., F_{cause} with post-condition R_{cause}) only goal configurations 2, 3, 5, 6, 8, and 9 in Figure 6.3 are applicable. For the detection analysis, the potential causing feature (e.g., B) is then constrained to satisfy its pre-condition expectations (e.g., B.3) while not satisfying its post-condition expectations (e.g., ¬B.1). For example, feature B is evaluated for satisfaction of Expression 6.14 that is derived from Expression 6.6, an example of a feature interaction property:

$$\bigwedge_{i=1}^{|F|}(P_i) \wedge \neg R_{\text{B}}, \tag{6.14}$$

where F can be any of the sets of features that satisfy Equation 6.13 and include the feature under analysis (i.e., at least feature B and one of feature D or E). That is, the causing feature (i.e., B) would be satisfied (i.e., implemented to the specification) if it were not for the presence of the other features in the set. Given a feature interaction analysis for feature B as the cause, the possible sets of features that satisfy the top-level goal and satisfy their pre-conditions (i.e., Equation 6.13) are: {B, E}, {B, C, E}, {B, D}, {B, C, D}. Feature sets that do not include feature B cannot have feature B as a feature interaction cause, and we already know that feature D and feature E are mutually exclusive. In order to detect a feature interaction, the set of features must constrain the possible implementations of feature B such that feature B cannot be implemented according to its specifications. Importantly, for each feature's pre-condition (P_i) that is satisfied, the post-condition (R_i) is satisfied unless otherwise constrained (i.e., Equation 6.3) For example, the pre- and post-conditions for feature set {B, E} are:

$$P_{\mathtt{B}} = (\text{CBF} > 0) \tag{6.15}$$

$$P_{\mathtt{E}} = (\text{SS} = \text{false}) \wedge (\text{BF} > 0) \tag{6.16}$$

$$R_{\mathtt{B}} = (\text{SF} = \text{CBF}) \tag{6.17}$$

$$R_{\mathtt{E}} = (\text{CBF} = \text{BF}). \tag{6.18}$$

In order for Equation 6.14 to detect a feature interaction caused by feature B given the feature set of {B, E}, three conditions must be satisfied. First, the pre-conditions of both features (i.e., $P_{\mathtt{B}}$ and $P_{\mathtt{E}}$) must be satisfied. Second, the properties (post-conditions) of the features with satisfied pre-conditions must be satisfied (i.e., $R_{\mathtt{B}}$ and $R_{\mathtt{E}}$), unless otherwise constrained by another expectation (i.e., pre- or post-condition). Finally, the feature under analysis as the cause (i.e., feature B) must violate its properties (i.e., $\neg R_{\mathtt{B}}$) due to the pre- or post-condition of another feature in the feature set. However, since SF is not constrained

by any other pre- or post-condition, then feature B cannot be a feature interaction cause given the feature set $\{\mathtt{B}, \mathtt{E}\}$. In summary, for feature set $\{\mathtt{B}, \mathtt{E}\}$ with feature B under analysis, the following equation representing the feature interaction properties must be satisfiable to detect a feature interaction caused by feature B:

$$(P_{\mathtt{B}} \wedge P_{\mathtt{E}}) \wedge \neg R_{\mathtt{B}}, \tag{6.19}$$

where both $R_{\mathtt{B}}$ and $R_{\mathtt{E}}$ must also be satisfied, unless constrained by any other pre- or post-condition (i.e., elements of P, and R, respectively).

However, in the case of $\{\mathtt{B}, \mathtt{C}, \mathtt{E}\}$, the pre- and post-conditions are:

$$P_{\mathtt{B}} = (\text{CBF} > 0) \tag{6.20}$$

$$P_{\mathtt{C}} = ((\text{CBF} > 0) \wedge ((\text{CBF} > 20) \vee (\text{CBF} < 50)) \tag{6.21}$$

$$P_{\mathtt{E}} = (\text{SS} = \text{false}) \wedge (\text{BF} > 0) \tag{6.22}$$

$$R_{\mathtt{B}} = (\text{SF} = \text{CBF}) \tag{6.23}$$

$$R_{\mathtt{C}} = (\text{SF} = 0.2 * \text{CBF}) \vee (\text{RF} = 0.8 * \text{CBF}) \tag{6.24}$$

$$R_{\mathtt{E}} = (\text{CBF} = \text{BF}). \tag{6.25}$$

In order for Equation 6.14 to detect a feature interaction caused by feature B given the feature set of $\{\mathtt{B}, \mathtt{C}, \mathtt{E}\}$, three conditions must be satisfied. First, the pre-conditions of both features (i.e., $P_{\mathtt{B}}$, $P_{\mathtt{C}}$, and $P_{\mathtt{E}}$) must be satisfied. Second, the properties (post-conditions) of the features with satisfied pre-conditions must be satisfied (i.e., $R_{\mathtt{B}}$, $R_{\mathtt{C}}$, and $R_{\mathtt{E}}$), unless otherwise constrained by another expectation (i.e., pre- or post-condition). Finally, the feature under analysis as the cause (i.e., feature B) must violate its properties (i.e., $\neg R_{\mathtt{B}}$) due to the pre- or post-condition of another feature in the feature set. Unlike the set $\{\mathtt{B}, \mathtt{E}\}$, in the set $\{\mathtt{B}, \mathtt{C}, \mathtt{E}\}$ the variable *SF* is constrained by both $R_{\mathtt{B}}$ and $R_{\mathtt{C}}$.

Importantly, in order for a post-condition to violate its properties due to another pre- or post-condition, several things must be true. That is, a post-condition must fail due to the constraint imposed by another pre- or post-condition where 1) the variables have all been constrained by another pre- or post-condition *and* 2) the pre- and post-conditions contribute (or would contribute if not for the FI) to the satisfaction of the top level goal. This necessitates that the pre- and post-conditions are decomposed from a satisfied parent, applied recursively until the top-level goal is satisfied (i.e., each parent is either decomposed from another satisfied parent or is the top-level goal). Similarly, the variables must be constrained by variables that are either similarly constrained *or* initialized by reading the variables from the environment (e.g., in requirements `E.2`, `E.4`, `E.7`, `D.2`, `D.4`, and `D.7`) where those pre- or post-conditions or requirements also contribute to the satisfaction of the top level goal.

Figure 6.5 identifies one possible configuration of satisfied goals that could lead to the violation of feature `B`'s post-condition. It is important to note that the pre-condition expression (i.e., $P_{\texttt{C}}$) for feature `C` is satisfied, despite the satisfaction of only one of its decomposed goals (i.e., goal `C.3`). Further, since each pre- and post-condition must contribute to the satisfaction of the top-level goal, the pre- and post-condition expressions for feature `C` (i.e., $P_{\texttt{C}}$ and $P_{\texttt{C}}$, respectively) must be satisfied by expectations `C.5` and `C.7`, `C.8` and `C.10`, or both due to their respective decompositions from goals `C.3` and `C.4`. The post-condition for feature `B` (i.e., `B.1`) is unsatisfied due to the conflict between itself and post-condition `C.9`. Post-condition `B.1` is exempt from contributing to the top-level goal due to its violation from another pre- or post-condition, an exemption which is applicable to any post-condition, not just those within the feature under analysis.

Specific values identified by *Phorcys* when analyzing feature `B` are presented in the feature interaction detection examples section (Section 6.6). The analysis logic presented here is generated for each feature.

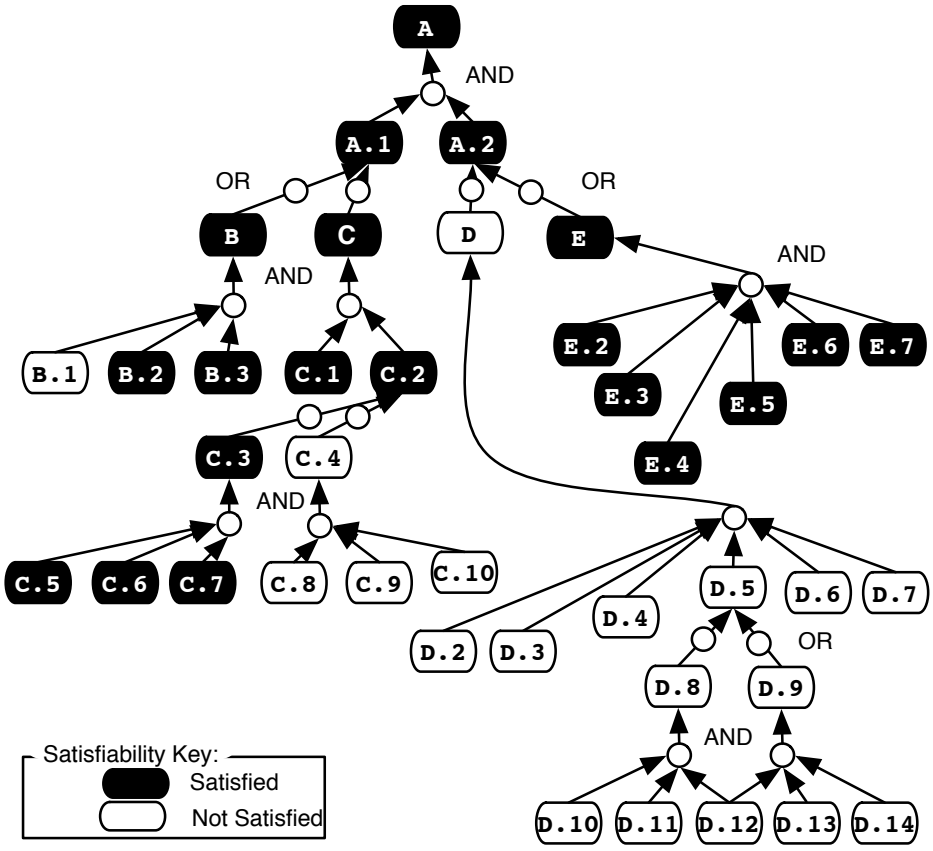


Figure 6.5: Feature Interaction in Goal Configuration Set $\{\mathtt{B}, \mathtt{C}, \mathtt{E}\}$

Step (2): Process Logical Expressions

The logical expressions defined by the process in Step 1, of the DFD in Figure 6.2 are translated into SMT v.2 [12] for analysis by a Satisfiability Modulo Theories (SMT) solver. SMT solvers follow the SMT-LIB [76] or SMT-LIB v.2 [12] standards to determine the satisfiability or validity of defined constraints [27]. In this work, we make use of the Microsoft Z3 SMT Solver [32].

For each feature, the valid goal configurations and feature interaction properties (i.e., the conjunction of Equations 6.13 and 6.14) specific for the feature under analysis are submitted to the SMT solver, and counterexamples are returned after processing the logical expressions. Therefore, a maximum of one counterexample is detected for each feature that acts as the cause of a feature interaction. If a conflict is detected, then it is added to the set of counterexamples to be summarized in Step 3.

Step (3): Counterexample Summary

For each feature that causes an interaction, the following information is summarized from the counterexample:

- The feature that *caused* the interaction,
- The set of features involved in the interaction, and
- The values of the instantiated variables.

Importantly, the features satisfied in the goal model may not be necessary for the feature interaction to occur but they are sufficient for the interaction to occur. There is no guarantee of minimal sets of features that cause an interaction, however the feature that *causes* an interaction is identified. Summarized counterexamples are stored for review by the system designer.

Update Goal Model by System Designer

The summarized counterexamples are intended to be used by the system designer to revise the input goal model. The root cause of a feature interaction may include one or more of the following incomplete list of possible problems:

- Incorrect pre- or post-conditions for a requirement,
- Incorrect decomposition of goals that do not meet desired requirements specification, or
- Incorrect decomposition of goals that indicate incorrect requirements specification.

The features identified as the cause of a feature interaction may be revised by: modifying the requirement and/or its associated pre- and post-conditions, or revising the decomposition of the goal model to restrict the feature interaction from occurring. Additional means of mitigating feature interactions are outside the scope of this chapter. A revised goal model

may then be analyzed again by *Phorcys* to detect newly introduced or remaining feature interactions.

6.5.2 Assumptions and Limitations

While a goal model, or its collection of pre-conditions, can be verified for satisfiability in a specific case by checking each expectation, identifying a specific satisfiable case is not always as computationally tractable. Using expectations (e.g., pre-conditions), a goal model is capable of representing a conjunction of disjunctions over variables to represent any instance of 3-SAT. Due to this reduction, the properties of 3-SAT also apply to goal model satisfaction in the expectations. Given that 3-SAT is an NP-Complete problem [62], goal model satisfiability is as well. Further, based on the exponential time hypothesis [57], such a problem is hypothesized not to be solvable in sub-exponential time in the worst case. Given the exponential nature of the feature interaction problem, this worst case is not unexpected. In fact, previous methods of n-way feature interaction detection using satisfiability have also been exponential in the worst case [9]. In general, reliably detecting n-way interactions requires exponential enumeration of possible interactions [5]. As such, *Phorcys* is no worse than existing feature interaction detection techniques in the worst case. However, the worst case is dependent on the system defined. Other hierarchically decomposed models, such as feature models, have been shown to typically be computationally inexpensive to symbolically analyze for realistic systems [68].

Similar to other feature interaction detection methods [41], *Phorcys* only analyzes a single step in time for feature interactions. This strategy may allow for unguarded, but still unreachable states to be detected as feature interactions. Similarly, cyclical or sequential changes, such as over and under shooting for a desired state, are undetectable as feature interactions even if the cycling is caused by competing features.

6.6 Examples

This section describes the results from applying *Phorcys* to the braking system goal model in Figure 6.1. *Phorcys* is used to illustrate two different types of feature interaction scenarios, involving more than two features: incorrect requirements and a typographical error, both of which could be extremely difficult to detect and identify using traditional testing techniques if these errors were propagated to program code.

6.6.1 Specification Error Causes

The braking system defined in Figure 6.1 identifies two potential sources for driving brake commands (features `D` and `E`) that provide the inputs to two methods of physically applying the brakes (features `B` and `C`). This initial braking system model is referred to as M. Table 6.1 provides the results of applying *Phorcys* to each of the features (`B`, `C`, `D`, and `E`) in goal model M to determine if any are the cause of a feature interaction.

Table 6.1: FI Causes in Goal Model M

Set of Features	Feature Under Analysis	Causes FI
{B, C, E}	B	Yes
{B, C, E}	C	Yes
N/A	D	No
N/A	E	No

After analysis, both features `B` and `C` are shown to cause a feature interaction. In both cases, the full set of features involved in the interaction are features `B`, `C`, and `E`, as shown in the 'Set of Features' column that identifies the features involved in the interaction.

Analyzing Feature `B` as Cause

The values of the variables in the goal model counterexample are shown in Table 6.2. Figure 6.6 illustrates the applicable parts of the goal model (at the feature level) for the 3-way feature interaction caused by feature `B` where the red text ($SF = ?$) indicates the

conflict detected. (The set of features represented in Figure 6.6 is goal configuration 6 in Figure 6.3). The brake force ($BF = 21.0$) is provided externally by the driver pressing on the brake pedal. This value is read via the *'Brake Pedal Sensor'* and converted to a commanded brake force ($CBF = 21.0$) by feature `D`. The features that apply the brake force, `B` and `C`, read this commanded brake force (CBF) from memory and use it to apply external brake forces. Feature `B` applies, via `B.2`, the entire commanded brake force to standard braking force ($SF = 21.0$) (e.g., drum or disk brakes on the wheels). Feature `C` applies a portion (defined by `C.5`) of the commanded brake force to the standard braking force ($SF = 4.2$) and a portion (defined by `C.10`) to the regenerative braking force ($RF = 16.8$). The interaction occurs due to the conflict where features `B` and `C` attempt to set SF to different values, clearly $SF = 21.0 \wedge SF = 4.2$ cannot be true simultaneously. *Phorcys*, in two separate analysis runs, is able to identify that both `B` or `C` can act as the cause of the interaction.

Table 6.2: Variables for Cause `B` Counterexample in Goal Model M

Variable	**Value**
SF	$21.0/5.0 = 4.2$
CBF	21
PBF	0.0
RF	$84.0/5.0 = 16.8$
BF	21.0
SS	*true*

Analyzing Feature `C` as Cause

A counterexample with a similar type of cause as the interaction caused by feature `B`: a conflict between `B` and `C` attempting to set SF to different values. Since the SMT solver identifies a different counterexample, the values identified for the goal model variables are different (as shown in Table 6.3), where feature `C` is the cause.

The interaction caused by feature `C` is a three-way feature interaction (i.e., `B`, `C`, `D`) where two features (i.e., `B`, `C`) are performing conflicting actions due to the commands of a third feature (i.e., `D`). This unwanted feature interaction is a requirements specification flaw

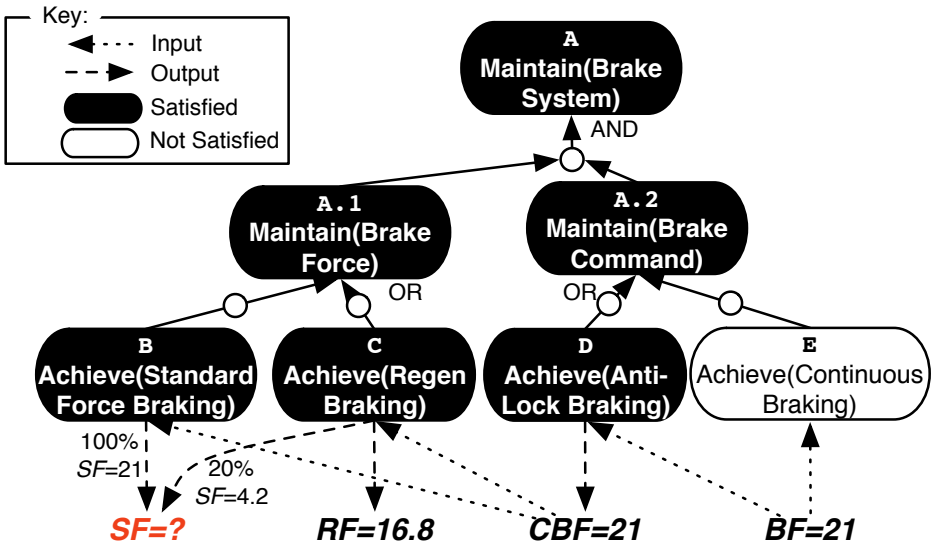


Figure 6.6: 3-Way FI in Figure 6.1 Caused by `B` or `C`

Table 6.3: Variables for Cause `C` Counterexample in Goal Model M

Variable	Value
SF	21.0
CBF	21.0
PBF	0.0
RF	$84.0/5.0 = 16.8$
BF	21.0
SS	*true*

(e.g., incorrect requirements) in the specification of the goal model. Two conflicting values should not be applied to SF. However, it may not be obvious that such an interaction exists due to a requirements specification flaw when features are developed independently. This specification flaw was not purposefully injected into the goal model and was not obvious to the authors when the features were developed independently and without any consideration for other features.

6.6.2 Modeling Error Introduces FI

One method of handing conflicts due to specification flaws, such as the one that exists in Figure 6.1, is strengthening the pre-condition expectations for one or both of the causing features. This strategy reduces, or even eliminates, the overlap between features when

conflicting goals are satisfied. For example, it may be reasonable to enforce standard force braking only, as provided by feature B, when the commanded brake force is over 80%. Conversely, C would be applicable under a commanded force of 80%. The reasoning behind such a change may be related to safety where standard force brakes have known historical failure rates while regenerative braking is a relatively new technology. Standard force braking (feature B) may be more reliable than a combination of braking methods (feature C) that include regenerative braking. Therefore, if the commanded brake force is high (greater than 80%) it may be reasonable to assume braking is safety related, thus justifying the use of only standard brake force using feature B. However, braking forces that are lower (under 80%) may indicate a less safety critical need and justify the electrical power capture from regenerative braking in feature C.

The system designer adds additional pre-condition expectations to both goals B and C to remove the interaction. The additional pre-condition expectation for goal B is B.0, $CBF \geqslant 80.0$, and the additional pre-condition expectation for goal C is C.0, $CBF \leqslant 80.0$. This revised goal model is referred to as M'.

Phorcys is then used to process the goal model M' to identify any remaining or newly introduced feature interactions. Executing *Phorcys* shows that features B and C still cause feature interactions, as shown in Table 6.4.

Table 6.4: FI Causes in Goal Model M'

Set of Features	Feature Under Analysis	Causes FI
{B, C, D}	B	Yes
{B, C, E}	C	Yes
N/A	D	No
N/A	E	No

Analyzing Feature B as Cause

In the revised goal model, the set of features in the counterexample are B, C, and D. Despite the pre-condition expectation B.0, an interaction is still caused by feature B.

However, as shown in Table 6.5a, *CBF* is equal to 80.0, exactly the intended cutoff between features `B` and `C`. Importantly, a modeling error (i.e., typo) has been inadvertently introduced by the system designer while manually updating the goal model that allows features `B` and `C` to overlap due to both expectations including the value 80.0, thus allowing an interaction to continue to exist when $CBF = 80.0$.

Similar to the previous interaction causes in model M, the interactions in model M' pass a value from the driver set brake force ($BF = 80.0$) that is then passed on to the commanded brake force ($CBF = 80.0$) via feature `D`. Just as before, features `B` and `C` both attempt to set the standard brake force (SF) to differing values (80.0 and 16.0 via expectations `B.1` and `C.5`, respectively). In this case, the additional expectations (`B.0` and `C.0`) only allow features `B` and `C` to occur simultaneously when the commanded brake force (CBF) is exactly 80.0.

Analyzing Feature `C` as Cause

The counterexample includes features `B`, `C`, and `D`. As shown in the counterexamples variable values in Table 6.5b, *CBF* is equal to 80.0 just as it was when `B` was the cause. The difference is which feature sets *SF* to a value. In the case where feature `C` is the cause of the interaction, standard braking force is set to 80.0. In the case where feature `B` is the cause, standard braking force is set to 16.0. Thus, the causing feature is the one that is not satisfied, and thus causes the top-level goal to be unsatisfied, in the presence of the other feature.

Both of these detected FIs are present in the goal model shown in Figure 6.7, a detailed view of goal configuration 6 from Figure 6.3 representing one of the 9 possible goal configurations.

Instead of correcting the feature interaction in model M by adding pre-condition expectations to the features that caused the interactions, the M' revision allowed for a single point of overlap between the additional pre-conditions at a commanded brake force of exactly 80.0.

Table 6.5: Counterexamples for FIs in Goal Model M'

Variable	Value
SF	16.0
CBF	80.0
PBF	0.0
RF	65.0
BF	80.0
SS	*true*

(a) Cause `B` Variables

Variable	Value
SF	80.0
CBF	80.0
PBF	-1.0
RF	0.0
BF	80.0
SS	*false*

(b) Cause `C` Variables

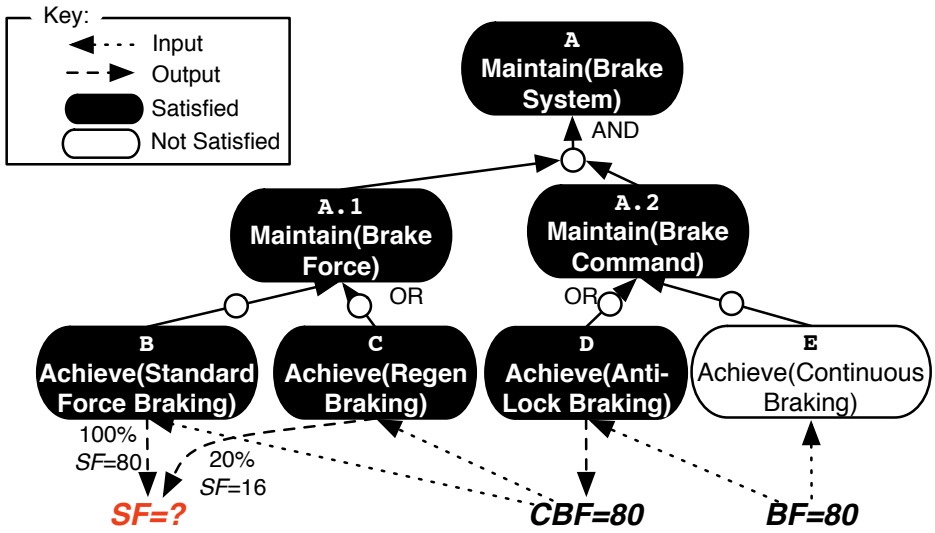


Figure 6.7: 3-Way FI in M' Caused by `B` or `C`

This form of feature interaction is not a specification error, but rather a modeling error that is still caught by *Phorcys*. Again, this error is subtle and difficult to detect with existing state of practice techniques.

6.6.3 Feature Interaction Free Model

We can mitigate this feature interaction by revising of the pre-conditions of the causing feature `C`. Specifically, expectation `C.O` is revised to $CBF < 80.0$ to yield model M''. Applying *Phorcys* to M'' detects no further interactions as shown in the results in Table 6.6.

Table 6.6: Detected FI Causes in Goal Model M''

Feature	Causes FI
`B` Achieve(Standard Force Braking)	No
`C` Achieve(Regen Braking)	No
`D` Achieve(Anti-Lock Braking)	No
`E` Achieve(Continuous Braking)	No

6.6.4 Discussion

Phorcys is able to detect n-way feature interactions with more than two features and provide counterexamples to support specification updates. Due to the feature-oriented requirements specification, an initial requirements specification flaw is uncovered in model M that could produce an unwanted interaction when the features are combined. In the revised model M', a subsequent modeling error (e.g., typo) was inadvertently introduced by the system designer leading to a single possible 3-way unwanted interaction. Finally, an updated model (M'') is shown to be interaction free. All executions were performed on a laptop with a 1.3 GHz Intel Core i5 processor and 4 GB of 1600 MHz DDR3 RAM in fractions of a second.

6.7 Related Work

Significant work has been done in the area of representing features as well as detecting, avoiding, and resolving feature interactions. This section overviews and compares related work. More comprehensive surveys on feature interaction have been presented elsewhere [4, 5, 20].

6.7.1 Feature Interaction

In general, feature interactions have been addressed by three main categories of techniques: detection, avoidance, and resolution [20], each of which is detailed below.

Detection

Typically, the detection of feature interactions focuses on identifying minimal sets of features that are necessary for a feature interaction to take place [41, 59, 6, 7]. Often, these methods are limited to pair-wise detection [21, 41], where only interactions between two features can be detected. For n-way feature interaction detection techniques, these minimal sets require a potentially exponential number of feature interactions to be detected [7]. *Phorcys* detects which features can *cause* an interaction thus providing a linear number of representative interactions that the system designer must assess (not to be conflated with a linear *run-time*). Search-based techniques for discovering latent properties, including feature interactions, exist, but require manual analysis of discovered properties [52, 60]. *Phorcys* differs from existing feature interaction detection methods by *combining* three key elements:

- it operates at the requirements level,
- it detects n-way feature interactions, and
- it identifies the *cause* of the interaction.

Avoidance

Feature Interaction avoidance composes features using operators that are conflict-free [55, 58] or makes use of frameworks that ensure separation of potentially conflicting concerns, eliminating specific classes of feature interactions [86]. Using these methods, the design and development of features is limited to the operators or frameworks employed. Detection methods, including *Phorcys*, could be employed to ensure no unexpected feature interactions occur due to limitations of the composition operators or frameworks.

Resolution

Feature Interaction resolution is often performed at run-time, and includes techniques such as prioritization [22] and negotiation [53] between features. Recently, methods to resolve

interactions based on the conflicting variables have been presented to reduce the number of resolutions [17], though dependencies between variables (e.g., speed and acceleration) are not considered for interactions. This method does not provide guarantees that the resolution provides desired behavior for the system. Therefore, while resolution at run-time may be better than allowing a conflict to occur, detecting and removing an interaction at design-time with *Phorcys* allows for verification of the modified behavior. When resolutions must be defined at design-time, detection methods like *Phorcys* can be used to identify which resolutions are necessary, thus eliminating excess resolutions.

6.7.2 Feature Representation

In this dissertation, we use annotated goal models to represent features, however features have also been represented as various behavioral models [41], logical representations [21, 9], or product line models [26, 61]. Often product lines are represented as feature models [59]. While goal modeling, product lines, and feature models are high-level representations of systems, behavioral and logical specifications necessitate specific implementation details. Feature models only decompose to the feature level and use cross-tree constraints that are limited to the feature level [13]. Therefore, feature models can only model conflicts between entire features and miss requirements-level conflicts that often cause feature interactions [6, 84]. For example, in Figure 6.1, features `B` and `C` conflict for only a range of values. The cross-tree constraints represented in feature models cannot adequately constrain the features without also constraining non-interacting functionality. The annotated goal models presented in this dissertation allow features to be decomposed to the requirements level, allowing for the detection of interactions that occur only in a portion of a feature's functionality while allowing for detection before implementation artifacts are created.

6.8 Summary

In this chapter, we have presented *Phorcys*, a design-time approach for detecting unwanted n-way feature interactions and determining their causes at the requirements level. Unlike previous n-way feature interaction detection approaches that attempt to enumerate every set of features that interact, *Phorcys* reduces the effort of the system developer by analyzing each feature for its ability to *cause* an interaction with other features. The method we present offers several advantages to reducing the cost of feature interaction detection. First, *Phorcys* reduces the computational and system designer effort by detecting interactions at the requirements level rather than implementation specifications. Second, *Phorcys* reduces the system designer effort by identifying which features cause an interaction rather than reviewing a potentially exponential set of interactions. Third, *Phorcys* reduces the computational effort to detect feature interactions for a subset of feature interaction detection problems by applying optimized constraint solvers in the form of SMT solvers to feature interaction detection. We demonstrated *Phorcys* on a cyber-physical system represented as a braking system goal model comprised of several features that are composed to realize an automotive braking system. We show that *Phorcys* is able to automatically detect the causes of n-way feature interactions due to specification and modeling flaws in fractions of a second.

Chapter 7

Detecting Feature Interactions: Using Evolutionary Computation

Ensuring acceptable and safe behavior is of paramount importance for high-assurance systems across the full range of functionality, not just a single scenario. However, detecting whether each feature can cause of a feature interaction only provides a single counterexample for each feature. This chapter introduces *Phorcys*-EC, a design-time approach for detecting unwanted failures caused by n-way feature interactions at the requirements level using both symbolic analysis and evolutionary computation to identify multiple counterexamples. Unlike previous n-way feature interaction detection approaches that look for single features that cause an interaction, *Phorcys*-EC uses a combination of symbolic analysis and evolutionary computation to identify multiple counterexamples for each feature, thus providing more guidance for mitigation (e.g., revising specifications, adding constraints, etc.). To the best of the authors' knowledge, *Phorcys*-EC is the only technique to detect failures caused by n-way feature interactions using a combination of symbolic analysis and evolutionary computation. We illustrate our approach by applying *Phorcys*-EC to an industry-based automotive braking system comprising multiple subsystems.

7.1 Introduction

This chapter presents *Phorcys*-EC,[1] a method for 1) identifying whether a feature fails due to a feature interaction and 2) returning a collection of diverse counterexamples across features and environmental properties using a combination of symbolic analysis and evolutionary computation.

N-way feature interaction detection methods typically detect a potentially exponential number of feature interactions (i.e., $\mathcal{O}(|F|^2)$, where F is the set of features). In order to address the computational intractability of n-way feature interaction detection, many researchers have focused on pair-wise interactions [10, 21, 41, 59]. While this approach decreases the number of potential interactions detected to $\mathcal{O}(|F|^2)$, interactions that only emerge when three or more features are present will not be detected [5]. Methods to detect which features cause an interaction without "creating all possible feature interactions" [5] have previously been considered unreliable, but such methods that are sounds and complete are a necessity. Additionally, a single counterexample may be insufficient

This chapter presents *Phorcys*-EC, an extension of the symbolic approach *Phorcys*. *Phorcys*-EC detects feature interactions using a combination of symbolic analysis and evolutionary computation. That is, feature interaction counterexamples are identified via a feature-oriented analysis based on whether a feature can fail due to an interaction. For example, consider 3 features, $\texttt{F}_1$, $\texttt{F}_2$, and $\texttt{F}_3$. If an interaction exists, then at least one feature ($\texttt{F}_1$, $\texttt{F}_2$, or $\texttt{F}_3$) no longer operates as it did independently. However, where *Phorcys* would provide up to a single counterexample for each feature, *Phorcys*-EC identifies multiple counterexamples with diverse feature sets and environmental scenarios for each feature.

Phorcys-EC analyzes features represented in goal models that hierarchically decompose a high-level goal down to individual requirements [87]. *Phorcys*-EC, using *Phorcys*, symbolically analyzes each feature with respect to different possible feature combinations of the goal model and uses a constraint solver to check for the existence of any combination of

[1] *Phorcys*-EC (pronounced 'forsis') is the Greek god of hidden dangers in the deep.

features (i.e., feature set) where the analyzed feature can fail due to a conflict in one or more requirements to be satisfied. The initially identified feature interaction counterexample is used to pre-populate a genetic algorithm that searches for additional diverse counterexamples. Periodically, the symbolic analysis will be re-applied to re-seed the genetic algorithm during evolution. Where previous n-way feature interaction detection methods may present an intractably large set of automatically detected interactions that are only diverse across features (i.e., do not include additional counterexamples with diverse environmental scenarios for a single interacting feature set) for the system designer to assess manually as to what environmental conditions are necessary for the feature interaction to occur [18, 20], *Phorcys*-EC identifies sets of feature interaction counterexamples for each feature that can fail due to an interaction. The diverse nature of the counterexamples provides more information (i.e., a larger range of environmental scenarios and/or a larger set of features) that can be utilized when revising the requirements and assumptions to mitigate the feature interaction.

The contributions of this chapter are as follows:

- We introduce a new approach for analyzing requirements to detect unintended feature failures caused by n-way feature interactions using symbolic analysis and evolutionary computation,
- The new approach can identify multiple counterexamples that represent diverse sets of features *and* diverse environmental scenarios for a given feature that can cause a feature interaction,
- We present *Phorcys*-EC, a prototype implementation of the approach, and
- We demonstrate the applicability of *Phorcys*-EC on an industry-based application of an automotive braking system based in part on the feature interaction issues in the 2010 Toyota Prius [29, 63], a hybrid vehicle. Analysis shows that *Phorcys*-EC is able to identify feature failures caused by feature interactions in the braking system goal model involving two or more features.

The remainder of this chapter is organized as follows. Section 7.2 provides background information on the automotive braking system used for feature interaction analysis in this chapter. The *Phorcys*-EC approach, examples, and results are included in Sections 7.3 and 7.4, respectively. Work related to *Phorcys*-EC is presented in Section 7.5, and our conclusions are described in Section 7.6.

7.2 Input Model

The goal model in Figure 7.1 represents an automotive braking system with several features. This model is similar to previous braking system goal models; however it has been updated such that the variables are mapped appropriately to a range of zero to one to allow for a more direct application of evolutionary computation in the range of zero to one. This not only causes some changes to the goal model, but subsequently to the results themselves. In general, a feature reads the actual brake pedal position and relates that information to a commanded brake force that is translated to an external braking force via a braking mechanism. However, for this system, the braking system has been developed as four individual features (i.e., `B`, `C`, `D`, `E`) rather than as a single, monolithic system. The use of multiple brake features is based on a known 2010 Toyota Prius braking system issue [29]. The braking system features defined here have been independently developed without consideration for each other and is intended to be a realistic example of industrial design artifacts at the requirements level, based on our collaboration with automotive industrial practitioners.[2]

7.3 Evolutionary Computation Approach

This section presents the *Phorcys*-EC approach and a comparison of analysis approaches.

[2]Collaboration with automotive industrial practitioners included reviewing our goal modeling approach to automotive systems, however the specific braking system presented here was not explicitly reviewed.

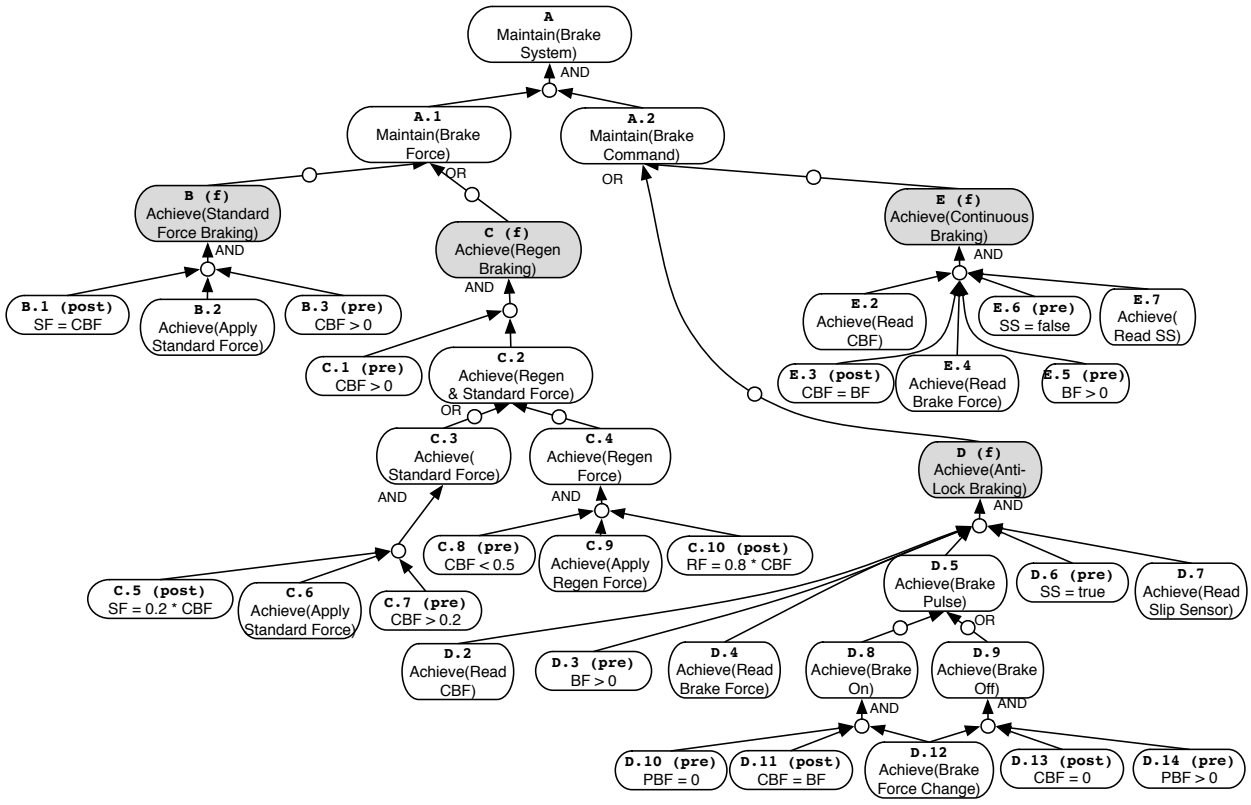


Figure 7.1: Braking System Goal Model

7.3.1 *Phorcys*-EC Approach

Figure 7.2 overviews the *Phorcys*-EC process with a DFD. As shown in Step 1 of Figure 7.2, *Phorcys*-EC accepts a goal model defined by the system designer (including feature and pre-/post-condition annotations), such as the one in Figure 7.1. Step 2 analyzes each feature to determine if a feature can fail due to a feature interaction. The *Phorcys*-EC output from Step 3 contains the features that fail due to an interaction along with affected features, as well as the concrete values of the variables in the system. Next, we describe each of the steps in the DFD in turn.

Step (1): Generate FI Detection Logic

Features that can fail due to a feature interaction, or the *causes* of feature interactions, are detected based on generated logical expressions for each feature represented in the goal

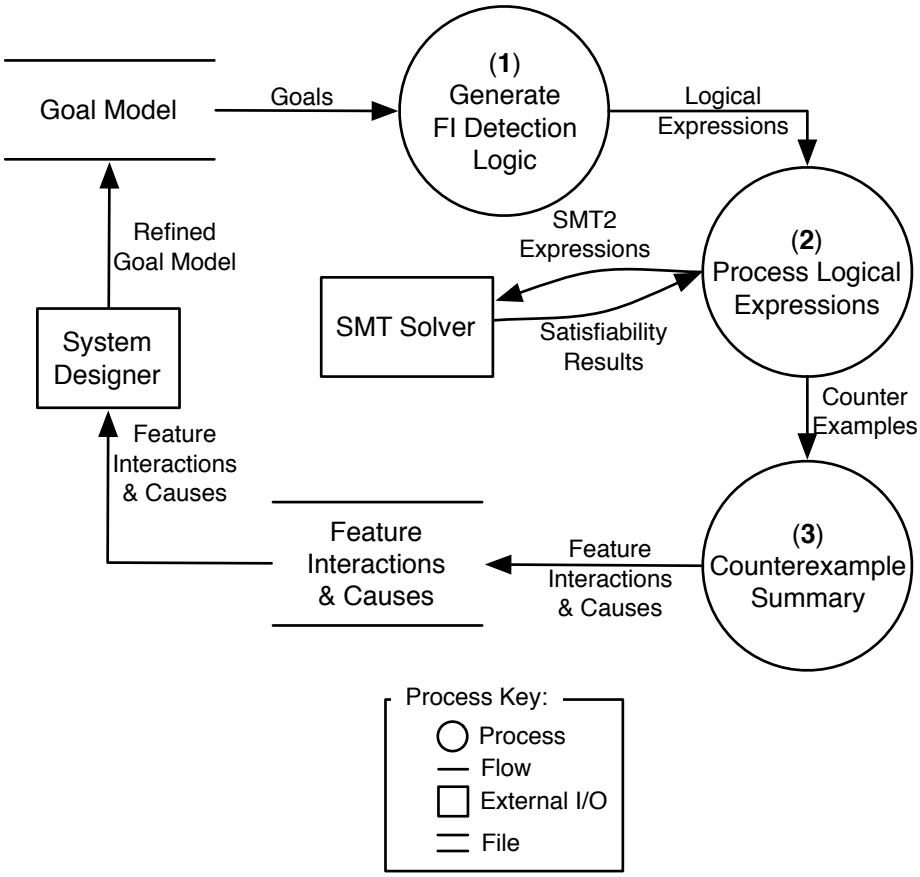


Figure 7.2: *Phorcys*-EC Data Flow Diagram

model provided by the system designer. The logical expressions are generated from the pre- and post-conditions in the goal model that are satisfied when a goal model instance is both i) valid and ii) contains properties of a feature interaction as defined within this step. The specific goal model instances are created by symbolically analyzing the logical expressions (i.e., Equation 6.6).

The feature interaction detection logic generated is identical to that of *Phorcys*, and is used for both symbolic analysis and evolutionary computation. Specific values identified by *Phorcys*-EC when analyzing feature `B` are presented in the Examples section (Section 7.4). The analysis logic is generated for each feature.

Step (2): Process Logical Expressions

The logical expressions generated by *Phorcys*-EC are processed by one of three methods that are compared in this chapter: Symbolic Analysis (SA), Evolutionary Computation (EC), or Symbolic Analysis and Evolutionary Computation (SA+EC).

Step (3): Counterexample Summary

For each feature that can fail due to an interaction, the following information is summarized from the counterexample(s):

- The feature that failed due to the interaction,
- The set of features involved in the interaction, and
- The values of the instantiated variables.

Importantly, the features satisfied in the goal model may not be necessary for the feature interaction to occur but they are sufficient for the interaction to occur. There is no guarantee of minimal sets of features that cause an interaction, however as the analysis identifies if a feature can cause a feature interaction (i.e., fail due to a feature interaction) the cause is intrinsically identified. Summarized counterexamples can be reviewed by the system designer.

Update Goal Model by System Designer

The summarized counterexamples are intended to be used by the system designer to revise the input goal model. The root cause of a feature interaction may include one or more of the following incomplete list of possible problems: incorrect pre- or post-conditions for a requirement; incorrect decomposition of goals that do not meet desired requirements specification; or indicate incorrect requirements specification.

The features identified as failing due to a feature interaction may be revised by: modifying the requirement and/or its associated pre- and post-conditions or revising the decomposition of the goal model to restrict the feature interaction from occurring. A revised goal

model may then be analyzed again by *Phorcys*-EC to detect newly introduced or remaining feature interactions.

7.3.2 Comparison of Analysis Approaches

This subsection provides details of three implementations used to analyze the logical expressions, as shown in Step (2) in the *Phorcys*-EC DFD in Figure 7.2.

Symbolic Analysis (SA)

As illustrated in Figure 7.3, the symbolic analysis instantiation of Step (2) in Figure 7.2 accepts the logical expressions defined by the process in Step (1). These logical expressions are translated into SMT version 2 [12] for analysis by a Satisfiability Modulo Theories (SMT) solver and the results are returned. SMT solvers follow the SMT-LIB [76] or SMT-LIB version 2 [12] standards to determine the satisfiability or validity of defined constraints [27] in a diverse range of constraint problems [33]. In this work, we make use of the Microsoft Z3 SMT Solver [32].

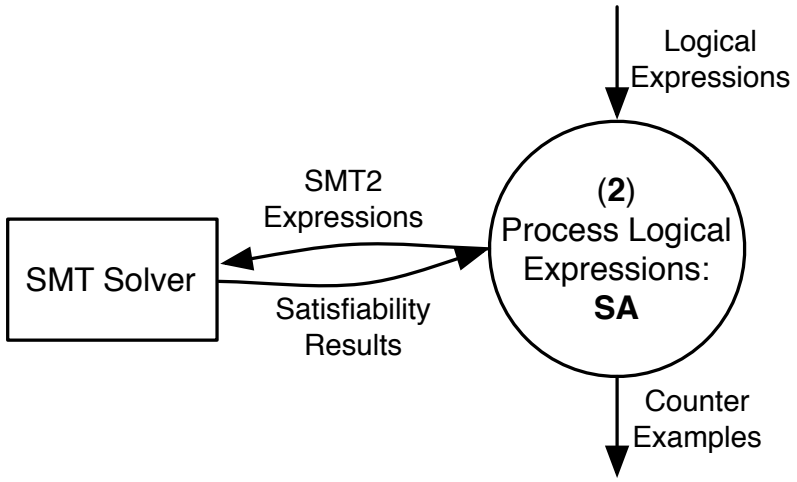


Figure 7.3: *Phorcys*-EC Data Flow Diagram: Step 2, SA

For each feature, the valid goal configurations and feature interaction properties (i.e., the conjunction of Equations 6.13 and 6.14) specific for the feature under analysis are submitted

to the SMT solver, and counterexamples are returned after processing the logical expressions. Therefore, a *maximum of one* counterexample is detected for each feature that can fail due to a feature interaction. If a conflict is detected, then it is added to the set of counterexamples to be summarized in Step 3.

Evolutionary Computation (EC) Analysis

While symbolic analysis can identify individual counterexamples, EC naturally lends itself to identifying multiple counterexamples (i.e., optimum). The genome itself is represented by the variables in the logical expressions generated by *Phorcys*-EC, where each variable in the expression is represented by a real value.

The EC configuration parameters in this work are based on empirical feedback where both optimal results and execution time were emphasized. Both analysis approaches using evolutionary computation (i.e., the EC-only analysis approach, as well as the combined SA+EC analysis approach) make use of a population of 200, a novelty proportionate tournament of size 8 for mating selection, a fitness proportionate tournament of size 4 is used for survival selection, as well as a 5% mutation rate applied by randomly modifying a single real-value representing a variable in the genome. Mating is performed by the SBX crossover operator [35] for each real-value in the individuals selected. Novelty and fitness measures are based on minimum Manhattan distance [28] of the selected genotype values (e.g., features included in the feature set or environmental variables) from any other individual and the existence of a failure for the feature under analysis caused by a feature interaction, respectively. The number of generations is limited to 500. The intention of this genetic algorithm is to emphasize diversity using both the mating and mutation operators while survival selection optimizes the results by culling from the diverse individuals.

Therefore, evolutionary computation is used to search for both optimal (i.e., a feature interaction is detected via the *Phorcys*-EC generated detection logic) *and* novel results (e.g., counterexamples including diverse sets of features or diverse environmental scenarios). In-

stead of searching for a single counterexample identifying a feature interaction, EC looks for multiple solutions using the entire population of the genetic algorithm. The expected results are optimal individuals spread across a range of genotypes.

SA+EC Analysis

While EC can search for multiple solutions with diversity, the following limitations and problems can occur:

- A single or multiple 'needle(s) in a haystack' may exist as the optimum and be prohibitively difficult to find.
- There may be no gradient between optimal and non-optimal results providing no basis for improvement related to changes in fitness resulting in a degradation to random search.
- Dependencies between variables may require changes in additional variables to maintain optimality.

Phorcys-EC overcomes these issues by combining symbolic analysis with evolutionary computation. As illustrated in the DFD in Figure 7.4, *Phorcys*-EC initially and then periodically performs symbolic analysis to identify a single diverse counterexample. That symbolically-identified counterexample is used to supply or replace a portion of the genetic algorithm's population. In order to select this single diverse counterexample, *Phorcys*-EC selects 10 random individuals in the population to create a distance constraint from each of them during the first half of the generations. If a new and diverse counterexample is found, then 20 random individuals are replaced with the new counterexample. This process is performed initially without the existing random chosen individuals and another 4 times (for a total of 5 symbolic analyses, selected based on empirical feedback), evenly distributed throughout the first half of the genetic algorithm's generations.

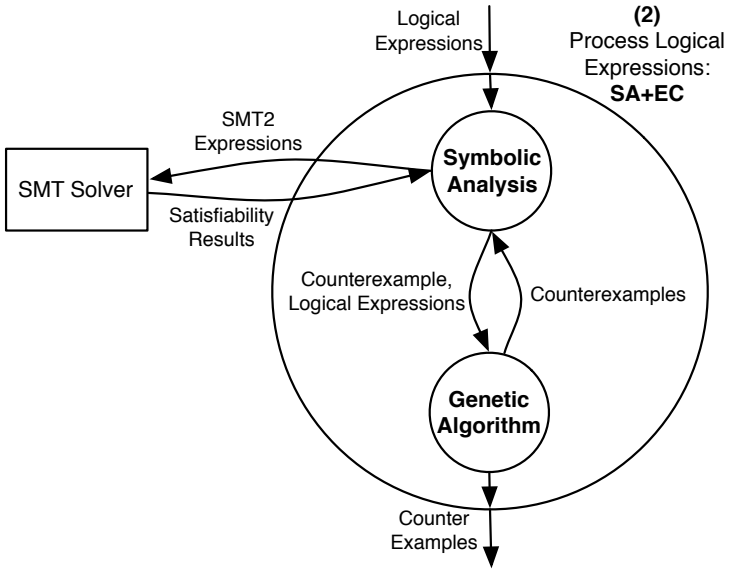


Figure 7.4: *Phorcys*-EC Data Flow Diagram: Step 2, SA+EC

7.4 Evolutionary Computation Case Study

This section provides the results of the *Phorcys*-EC tool when applied with each of the three methods of analyzing logical expressions (i.e., Step 2 in the *Phorcys*-EC DFD in Figure 7.2), as well as a comparison of the three methods.

7.4.1 Symbolic Analysis (SA)

The braking system defined in Figure 7.1 identifies two potential sources for driving brake commands (features `D` and `E`) that provide the inputs to two methods of physically applying the brakes (features `B` and `C`). Table 7.1 provides the results of applying *Phorcys*-EC to each of the features (`B`, `C`, `D`, and `E`) in the goal model to determine if any are the cause of a feature interaction.

Table 7.1: FI Causes in the Goal Model

Set of Features	Feature Under Analysis	Fails Due to FI
{B, C, E}	B	Yes
{B, C, E}	C	Yes
N/A	D	No
N/A	E	No

After analysis, both features B and C are shown to fail due to a feature interaction. In both cases, the full set of features involved in the interaction are features B, C, and E, as shown in the 'Set of Features' column that identifies the features involved in the interaction.

Analyzing Feature B for Failure Due to FI

The values of the variables in the goal model counterexample are shown in Table 7.2. Figure 7.5 illustrates the applicable parts of the goal model (at the feature level) for the 3-way feature interaction caused by feature B where the red text ($SF = ?$) indicates the conflict detected. (The set of features represented in Figure 7.5 is goal configuration 6 in Figure 6.3). The brake force ($BF = 0.35$) is provided externally by the driver pressing on the brake pedal. This value is read via the *'Brake Pedal Sensor'* and converted to a commanded brake force ($CBF = 0.35$) by feature D. The features that apply the brake force, B and C, read this commanded brake force (CBF) from memory and use it to apply external brake forces. Feature B applies, via B.2, the entire commanded brake force to standard braking force ($SF = 0.35$) (e.g., drum or disk brakes on the wheels). Feature C applies a portion (defined by C.5) of the commanded brake force to the standard braking force ($SF = 0.07$) and a portion (defined by C.10) to the regenerative braking force ($RF = 0.28$). The interaction occurs due to the conflict where features B and C both attempt to set SF to different values, clearly $SF = 0.35$ and $SF = 0.07$ cannot be true simultaneously. SA, in two separate analysis runs, is able to identify that both B or C can fail due to the interaction.

Analyzing Feature C for Failure Due to FI

A counterexample with a similar type failure of feature C: a conflict between B and C attempting to set SF to different values. Since the SMT solver identifies a different counterexample, the values identified for the goal model variables are different (as shown in Table 7.3), where feature C fails due to the interaction.

The interaction caused by feature C is a 3-way feature interaction (i.e., B, C, D) where

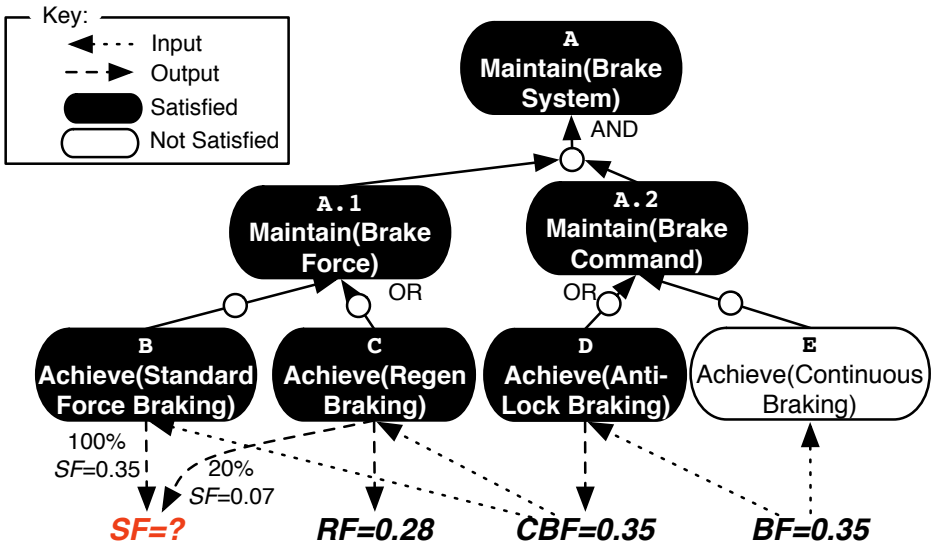


Figure 7.5: 3-Way FI in Figure 7.1 Caused by `B` or `C`

Variable	Value
SF	0.07
CBF	0.35
PBF	0.0
RF	0.28
BF	0.35
SS	*true*

Table 7.2: Feature `B`

Variable	Value
SF	0.35
CBF	0.35
PBF	0.0
RF	0.28
BF	0.35
SS	*true*

Table 7.3: Feature `C`

Variables for Failure

two features (i.e., `B`, `C`) are performing conflicting actions due to the commands of a third feature (i.e., `D`). This unwanted feature interaction is a requirements specification flaw (e.g., incorrect requirements) in the specification of the goal model. Two conflicting values should not be applied to *SF*. However, it may not be obvious that such an interaction exists due to a requirements specification flaw when features are developed independently of one another. This specification flaw was not purposefully injected into the goal model and was not obvious to the authors when the features were developed independently and without any consideration for other features.

7.4.2 Evolutionary Computation (EC)

The EC method uses a genetic algorithm to generate variable instantiations for the logical expressions to detect feature interactions in an effort to address the limitations of a single counterexample result identified by symbolic analysis. However, EC is unable to identify a single counterexample, much less a set of counterexamples, within 50 full executions of the genetic algorithm. Even significantly larger generations (2000) and larger populations (500) did not lead to success for the genetic algorithm. The most likely reason for the lack of results is the lack of a fitness gradient (i.e., either 0.0 or 1.0) for the detection of feature interactions causing the genetic algorithm to function as random search.

7.4.3 SA+EC

SA+EC detected both features `B` and `C` as features that can fail due to feature interactions, just as they were identified in the SA-only method. In the cases of features `B` and `C`, the SA+EC method is always able to identify a set of diverse counterexamples. In all 50 executions of the genetic algorithm, at least 97% of the population succeeded in finding a counterexample (i.e., feature interaction).

Feature Diversity

In the cases of both features `B` and `C`, the possible feature sets that are involved in the interaction are both `B` and `C` as well as one of either `D` or `E`. All executions of SA+EC, using feature diversity, identified all possible feature combinations that could provide the feature interaction that causes the feature under analysis to fail. That is, each of the 50 executions for both features `B` and `C` identified the exhaustive list of possible feature sets.

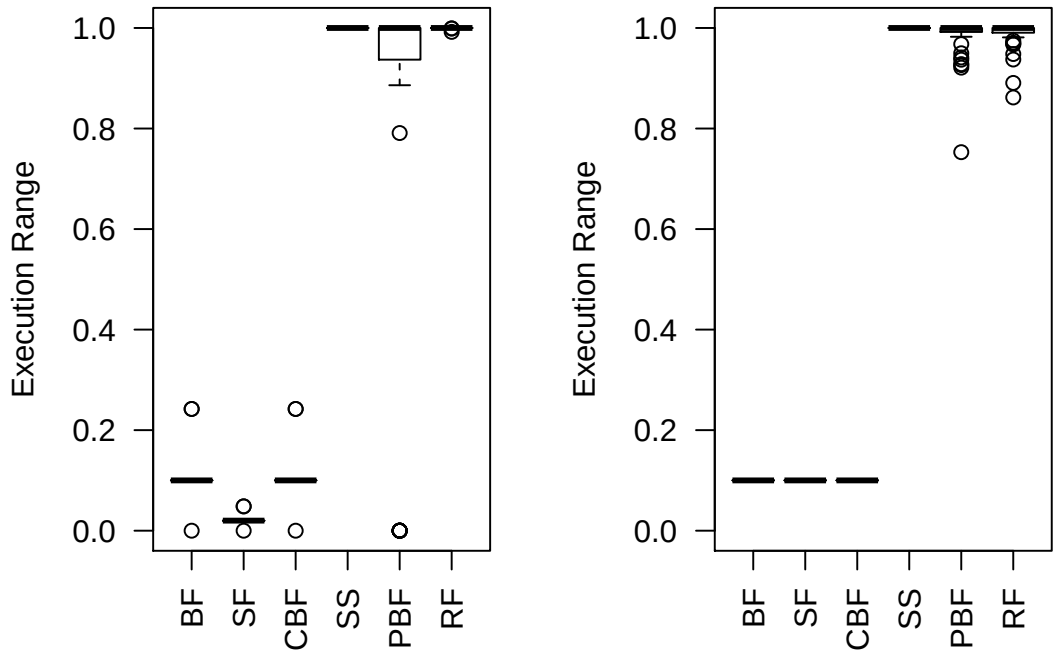
Environmental Diversity

Regardless of which feature, or features, failed due to the feature interaction, at least 99.9% of the feature interactions identified by SA+EC analysis included unique environmen-

tal variable values. That is, the feature interaction counterexamples identified were almost always completely unique and every set of counterexamples (i.e., in a population of 200 identified by the SA+EC) included significant diversity as measured by the Manhattan distance of the environmental scenarios. This diversity is illustrated in Figures 7.6a and 7.6b that show the range (distance from minimum to maximum) each environmental variable took across each set of 200 individuals in the SA+EC analysis, where the box plot represents all 50 executions of the SA+EC. It should be noted, that it is expected that some of the ranges are limited since they represent the ranges of environmental variable values necessary for feature interactions to cause a feature to fail. In both cases, we can see that *SS*, *PBF*, and *RF* have a large range of values while the remainder (i.e., *BF*, *SF*, and *CBF*) are limited due to the smaller range of those environmental variable values when feature interactions occur. With just the box plots of the ranges, it is clear that the feature interaction will be related to the variables *BF*, *SF*, and *CBF*. System designers may infer that the values with a more restricted range must be within those ranges in order for a feature interaction to occur, while variables with a larger range are not limited to specific values for a feature interaction to occur.

7.4.4 Comparison

The SA+EC approach used by *Phorcys*-EC is preferred over either SA or EC alone. Though SA can only identify a single counterexample for every feature that can fail due to a feature interaction, the failure is guaranteed to detect the interactions. This analysis differs from EC where there is no guarantee that a counterexample will be found even when one is known to exist. This weakness is especially apparent when the EC approach failed to detect *any* of the feature interactions due to the discrete nature of the fitness assessment. Therefore, while SA is limited to a single counterexample, it provides a guaranteed result and is the preferred solution when comparing between a SA approach or an EC approach. However, SA+EC is not only able to identify failures due to feature interactions as consis-



(a) `B` Failing

(b) `C` Failing

Figure 7.6: Environmental Variables

tently as SA, but SA+EC *also* identifies a diverse and unique set of counterexamples. Since SA+EC uses the SA approach, the guarantees of SA are also the guarantees of SA+EC since the EC portion is elite preserving and does not remove true counterexamples for anything but a more diverse true counterexample. Additionally, SA+EC was able to identify an exhaustive list of feature sets that can take part in a feature interaction *and* over 99.9% of counterexamples identified were unique. Therefore, SA+EC is preferable to the SA approach alone as SA+EC includes both the completeness and soundness guarantees of SA, as well as providing diverse results that better inform how to mitigate the feature interaction with respect to the requirements and/or expectation revisions.

7.5 Evolutionary Computation Related Work

Significant work has been done in the area of representing features as well as detecting, avoiding, and resolving feature interactions. This section overviews and compares related work limited to those that address detection, the focus of this work. More comprehensive surveys on feature interaction have been presented elsewhere [4, 5, 20].

Typically, the detection of feature interactions focuses on identifying minimal sets of features that are necessary for a feature interaction to take place [41, 59, 6, 7]. Often, these methods are limited to pair-wise detection [21, 41], where only interactions between two features can be detected. For n-way feature interaction detection techniques, these minimal sets require a potentially exponential number of feature interactions to be detected (i.e., every possible feature set) [7], while *Phorcys* detects which features can fail due to an interaction, up to only a single counterexample for each feature is provided. Search-based techniques for discovering latent properties, including feature interactions, exist, but require manual analysis of discovered properties [52, 60]. *Phorcys*-EC differs from existing feature interaction detection methods by *combining* several key elements:

- it operates at the requirements level,
- it determines if a feature can fail due to a feature interaction automatically (i.e., without manual analysis),
- it employs a combination of both symbolic analysis and evolutionary computation to generate a representative and diverse range of counterexamples for each feature that can fail due to a feature interaction.

7.6 Summary

In this chapter, we have described *Phorcys*-EC, a design-time approach for detecting unwanted n-way feature interactions and determining their causes at the requirements level

using symbolic analysis and evolutionary computation to identify diverse counterexamples. Unlike *Phorcys*, *Phorcys*-EC uses evolutionary computation and symbolic analysis to detect multiple counterexamples that represent diverse feature sets and environmental scenarios for each feature without sacrificing guaranteed detection. We have demonstrated *Phorcys*-EC on a braking system goal model comprising several features to realize an automotive braking system.

Future research will continue to explore how *Phorcys*-EC can be extended to address additional assurance challenges posed by computing-based systems as they interact with the environment. For example, we will explore how to detect feature interactions in the face of uncertainty due to environmental conditions as they impact system conditions. Additionally, while *Phorcys*-EC detects feature interactions at design-time, we will also investigate run-time feature interaction detection and mitigation strategies, just as existing SA+EC requirements completeness approaches [38] have been extended to run-time approaches [39]. We will also explore the use of RELAXed goals [94], whose utility functions are evaluated according to fuzzy logic expressions, or applying other transformations to introduce a gradient to requirements satisfaction in support of evolutionary search.

Chapter 8

Detecting Feature Interactions: At Run Time

The validity of systems at run time depends on the features included in those systems operating as specified. However, when feature interactions occur, the specifications no longer reflect the state of the run-time system due to the conflict. While methods exist to detect feature interactions at design time [40], conflicts that cause features to fail may still arise when new detected feature interactions are considered unreachable, new features are added, or an exhaustive design-time detection approach is impractical due to computational costs. This chapter introduces *Thoosa*, an approach for using models at run time to detect features that can fail due to feature interactions at run time. We illustrate our approach by applying *Thoosa* to an industry-based automotive braking system comprising multiple subsystems.

8.1 Introduction

While systems are expected to satisfy their specifications at run time, these specifications can only be satisfied if they are logically sound and do not include conflicts that result in feature interactions. Detecting feature interactions is challenging due to the exponential growth of potential interactions with respect to the number of features [17] and the range of

environmental possibilities in cyber-physical systems. More formally, when defining a feature interaction by the minimal set of features that are necessary for the interaction to occur, the number of possible interactions is $\mathcal{O}(2^{|F|})$, where F is the set of features [5]. The growth in the number possible feature interactions and consequently the number of possible feature interactions that must be assessed can result in latent behavior that impacts the system dependability at run time. This problem is particularly prominent in cyber-physical systems where unexpected adverse environmental conditions may occur and impact the dependability of the system. However, run-time detection of feature interactions reduces the possible feature and environmental combinations to a single concrete instantiation that must be analyzed for feature interactions. This chapter presents *Thoosa*,[1] a method for detecting each feature that causes an n-way feature interaction in a requirements goal model at run time.

This chapter presents *Thoosa*, a method for detecting each feature that fails at run time due to feature interactions using executable code generated from a symbolic representation of feature interactions. That is, feature interaction counterexamples are identified via a feature-oriented analysis based on whether a feature can fail due to an interaction given the concrete instantiation of the system at a given time. For example, consider 3 features, $\texttt{F}_1$, $\texttt{F}_2$, and $\texttt{F}_3$. If an interaction exists, then at least one feature ($\texttt{F}_1$, $\texttt{F}_2$, or $\texttt{F}_3$) no longer operates as it did independently. *Thoosa* reduces the computational effort by only analyzing a single scenario or set of features, at run time, while still detecting n-way feature interactions.

Thoosa analyzes features represented in goal models that hierarchically decompose a high-level goal down to individual requirements [87]. *Thoosa* executes generated logic, in the form of C++ code, that analyzes each feature with respect to the current feature combinations of the goal model and identifies if the analyzed feature can fail due to a conflict in one or more requirements to be satisfied at run time. Each feature is analyzed for failure due to a feature interaction. Where previous run-time feature interaction detection techniques

[1] *Thoosa* is a Greek sea nymph associated with swiftness and the daughter of *Phorcys*.

indicate that a feature interaction exists, *Thoosa* identifies which features fail due to the feature interaction.

The contributions of this chapter are as follows:

- We introduce a new approach for analyzing requirements to determine if they fail at run time due to a feature interaction,

- We present *Thoosa*, a prototype implementation of the approach, and

- We demonstrate the applicability of *Thoosa* on an industry-based application of an automotive braking system based in part on the feature interaction issues in the 2010 Toyota Prius [29, 63], a hybrid vehicle. Analysis shows that *Thoosa* is able to identify feature failures caused by feature interactions in the braking system goal model involving two or more features.

The remainder of this chapter is organized into the following sections. Section 8.2 details the approach. Section 8.3 describes an example application, and Section 8.4 details related work. Finally, Section 8.5 discusses the conclusions and avenues of future work.

8.2 Run-Time Approach

Figure 8.1 overviews the *Thoosa* process with a DFD. As shown in Step 1 of Figure 8.1, *Thoosa* accepts a goal model defined by the system designer (including feature and pre-/post-condition annotations), such as the one in Figure 6.1, and generates feature interaction detection logic. Step 2 generates monitoring code that can be executed at run time based on the feature interaction detection logic. The *Thoosa* output from Step 2 contains the generated monitoring code that can detect feature failures due to feature interactions. Next, we describe each of the steps in the DFD in turn.

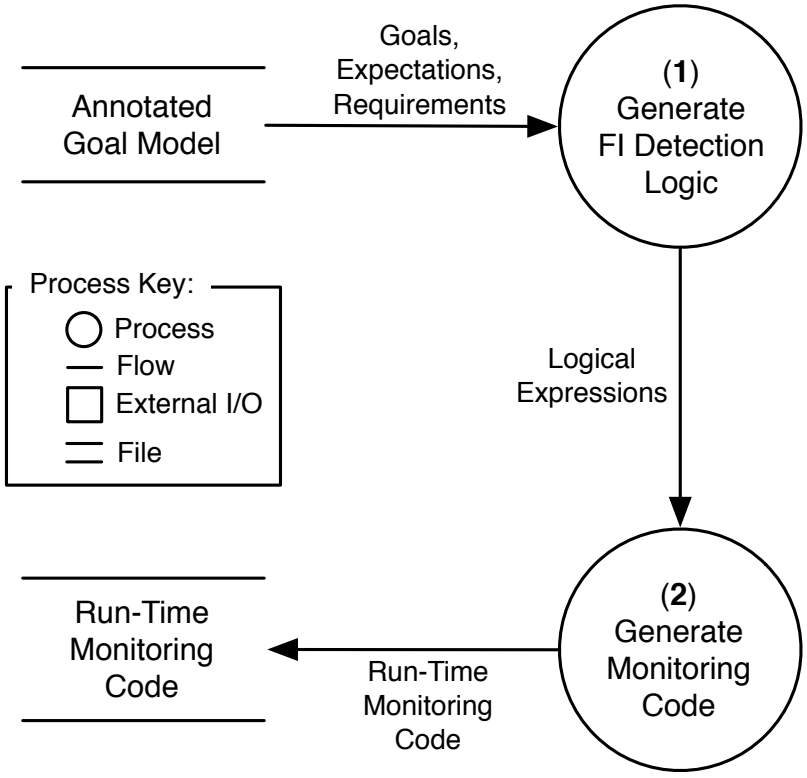


Figure 8.1: *Thoosa* Data Flow Diagram

8.2.1 Step (1): Generate FI Detection Logic

A failure of a specific feature is identified when that specific feature's properties are unsatisfied for a specific composition of features that includes the feature under analysis, as indicated in Equation (6.6). Assuming all features are analyzed for failure due to a feature interaction, Equation (6.6) is both complete (i.e., no missing interactions) and sound (i.e., no superfluous interactions detected). In this chapter, we employ Equation 6.6 to detect if a feature can fail due to a feature interaction where we measure the inclusion of a feature based on that feature's pre-conditions, while the feature properties are measured by the satisfaction of the feature's post-conditions.

Features that can fail due to a feature interaction are detected based on generated logical expressions for each feature represented in the goal model provided by the system designer. The logical expressions are generated from the pre- and post-conditions in the goal model that are satisfied when a goal model instance is both i) valid and ii) contains properties of a

feature interaction as defined within previous work on symbolically analyzing goal models for feature interactions in Chapter 6. That is, the feature interaction detection logic generated by *Thoosa* is the same as that generated by *Phorcys*.

8.2.2 Step (2): Generate Executable Code

Thoosa automatically generates executable code based on the logic generated to detect failures that are caused by feature interactions in Step (1). Specifically, the following items are calculated and assigned to variables based on the current state of goal model variables (i.e., the values those variables take on at run time while executing the implemented system):

- The satisfaction of the goal model based on the decomposition of the top-level features,
- The satisfaction of each of the feature pre-conditions (i.e., corresponding to *if* the feature should be included in the feature set and satisfied), and
- The satisfaction of each of the features including their pre- and post-conditions.

An example of the generated code is shown in the partial code listing in Figure 8.2, where the top-level goal satisfaction, feature pre-condition satisfaction, and feature satisfaction are all calculated. A full listing of the generated code is included in the Appendix.

Importantly, *all* features are analyzed for failure for the generated code instead of analyzing features one at a time. That is, each feature is analyzed for its current ability to cause a feature interaction (e.g., 'B_fails,' 'C_fails,' 'D_fails,' and 'E_fails' in Figure 8.2). The system designer must update the propagation, or 'freshness,' of any variables in the system by updating functions starting with a '*V_*' to return true when the variable has been updated, and false when it has not. For example, Figure 8.3 includes the sample code listing for '*V_SS*' that must be updated to return true when the slip sensor variable has been read from the sensor, and false when it is out of date. In a typical system, sensors would likely be read continuously while actuators would only propagate when they are specifically set to a value by the system.

```
// Calculate the top-level goal satisfaction
(topLevelSatisfation = ((B() || C()) && (D() || E())));

// Calculate the satisfaction of each feature's
//      preconditions
(B_precondition = B_P());
(C_precondition = C_P());
(D_precondition = D_P());
(E_precondition = E_P());

// Calculate
(B_fails = B_PP());
(C_fails = C_PP());
(D_fails = D_PP());
(E_fails = E_PP());
```

Figure 8.2: Variables Calculated for FI Detection in Figure 6.1

```
// Calculate variable propagation:
bool V_SS() {
    return true;
}
```

Figure 8.3: Propagation Function for *SS* (Slip Sensor) Figure 6.1

8.2.3 Record Failures and/or Adapt

Based on the values that are returned from the run-time detection of feature interactions and the features that fail due to the interaction, a partial list of appropriate systems actions that could be manually defined by the system designer are:

- The feature interaction along with the failing feature(s) could be recorded for later analysis. For example, a recorded feature interaction may be used by the system designer to update the system to mitigate that feature interaction.

- Specific mitigations may be put in place to allow fail-safe or fail-silent operation.

- Specific feature failures due to feature interactions and their satisfied pre-conditions

may be used as guidance or input to run-time adaptations to adapt from the feature set that causes a feature interaction.

8.2.4 Limitations

Counterexamples may be present in a scenario that occurs only between sensor readings. In such cases, the values calculated by the utility functions never represent an incompleteness or inconsistency and, therefore, no counterexample is detected.

8.3 Run-Time Case Study

This section provides the results of applying the *Thoosa* tool with simulated run-time inputs. The braking system defined in Figure 6.1 identifies two potential sources for driving brake commands (features `D` and `E`) that provide the inputs to two methods of physically applying the brakes (features `B` and `C`). Previous analysis of the braking system has shown that both features `B` and `C` can cause an interaction. We apply the variables identified in the symbolic approach in Chapter 6, to ensure the run-time approach of *Thoosa* correctly identifies features that can fail due to an interaction.

Run-Time Analysis of Feature `B` for Failure Due to FI

The values of the variables in the goal model counterexample when Feature `B` is the cause of a feature interaction are shown in Table 8.1 and are applied to the run-time system for detection. Figure 8.4 illustrates the applicable parts of the goal model (at the feature level) for the 3-way feature interaction caused by feature `B` where the red text ($SF = ?$) indicates the conflict that should be detected. The brake force ($BF = 0.35$) is provided externally by the driver pressing on the brake pedal. This value is read via the *'Brake Pedal Sensor'* and converted to a commanded brake force ($CBF = 0.35$) by feature `D`. The features that apply the brake force, `B` and `C`, read this commanded brake force (CBF) from memory

and use it to apply external brake forces. Feature `B` applies, via `B.2`, the entire commanded brake force to standard braking force ($SF = 0.35$) (e.g., drum or disk brakes on the wheels). Feature `C` applies a portion (defined by `C.5`) of the commanded brake force to the standard braking force ($SF = 0.07$) and a portion (defined by `C.10`) to the regenerative braking force ($RF = 0.28$). The interaction occurs due to the conflict where features `B` and `C` both attempt to set SF to different values, clearly $SF = 0.35$ and $SF = 0.07$ cannot be true simultaneously.

Table 8.1: Variables for Failure of Feature `B`

Variable	Value
SF	0.07
CBF	0.35
PBF	0.0
RF	0.28
BF	0.35
SS	*true*

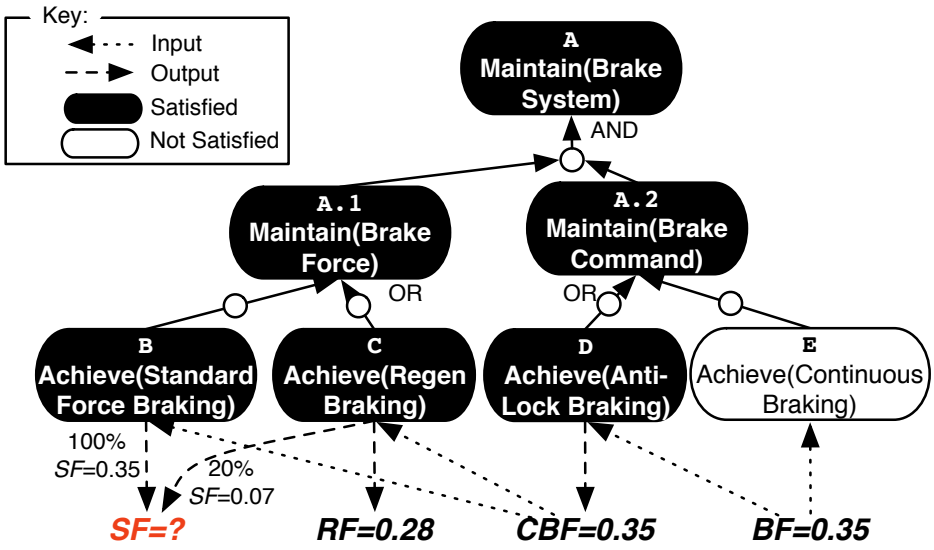


Figure 8.4: 3-Way FI in Figure 6.1 Caused by `B` or `C`

Thoosa does identify that feature `B` fails due to a feature interaction, along with the remainder of the run-time results shown in Table 8.2. That is, when a known counterexample is used to initialize the system variables, the detection code generated by *Thoosa* is able to detect the known feature interaction. Note, the top level goal is still satisfied because

either goal `B` or goal `C` is required, and goal `C` is still satisfied. Therefore, despite the feature interaction the system still succeeds, but not as intended. In this case the system designer may desire this feature interaction to be recorded, but adaptation is not strictly necessary due to the top-level satisfaction of the system. Other strategies are proposed in Sub-Section 8.2.3.

Table 8.2: Run-Time Results for Feature `B`

Identifier	**In Code**	**Result**
Top-Level (Goal `A`)	topLevelSatisfaction	Satisfied
Feature `B` Fails	B_fails	True
Feature `C` Fails	C_fails	False
Feature `D` Fails	D_fails	False
Feature `E` Fails	D_fails	False
Pre-Conditions for `B`	B_precondition	True
Pre-Conditions for `C`	C_precondition	True
Pre-Conditions for `D`	D_precondition	True
Pre-Conditions for `E`	E_precondition	False

Run-Time Analysis of Feature `C` for Failure Due to FI

A counterexample with a similar type of failure is the interaction failure of feature `C`: a conflict between `B` and `C` attempting to set *SF* to different values. The variable values applied to the run-time system, as shown in Table 8.3, result in the detection of a failure of feature `C` due to the interaction.

Table 8.3: Variables for Failure of Feature `C`

Variable	**Value**
SF	0.35
CBF	0.35
PBF	0.0
RF	0.28
BF	0.35
SS	*true*

Thoosa does identify that feature `C` fails due to a feature interaction, along with the remainder of the run-time results shown in Table 8.4. Note, the top-level goal is still satisfied because *either* goal `B` or goal `C` is required, and goal `B` is still satisfied. Therefore, despite

the feature interaction the system still succeeds, but not as intended. This is similar to the top-level satisfaction shown when feature `B` fails due to a feature interaction. Again, no adaptation is strictly required, however recording the feature interaction is one of several possibilities proposed in Sub-Section 8.2.3.

Table 8.4: Run-Time Results for Feature `C`

Identifier	**In Code**	**Result**
Top-Level (Goal `A`)	topLevelSatisfaction	Satisfied
Feature `B` Fails	B_fails	False
Feature `C` Fails	C_fails	True
Feature `D` Fails	D_fails	False
Feature `E` Fails	D_fails	False
Pre-Conditions for `B`	B_precondition	True
Pre-Conditions for `C`	C_precondition	True
Pre-Conditions for `D`	D_precondition	True
Pre-Conditions for `E`	E_precondition	False

8.4 Run-Time Related Work

Significant work has been done in the area of detecting, avoiding, and resolving feature interactions. This section overviews and compares related work limited to those that address run-time detection, the focus of this work. More comprehensive surveys on feature interactions have been presented elsewhere [4, 5, 20].

Often run-time detection techniques are paired with an effort at run-time resolution, including techniques such as prioritization [22] and negotiation [53] between features. Recently, methods to resolve interactions based on the conflicting variables have been presented to reduce the number of resolutions [17, 100], though dependencies between variables (e.g., speed and acceleration) are not considered for interactions. This method does not provide guarantees that the resolution provides desired behavior for the system. Therefore, while resolution at run-time may provide an acceptable solution, it is not guaranteed. *Thoosa* operates at the requirements level and identifies which features fail due to the feature interaction and

allows for mitigations to be created that adapt from an interacting set of features while still satisfying the top-level goal.

Typically, the design-time detection of feature interactions focuses on identifying minimal sets of features that are necessary for a feature interaction to take place [41, 59, 6, 7]. Often, these methods are limited to pair-wise detection [21, 41] or other small feature sets [44], where only interactions between two features can be detected to reduce computational cost. *Thoosa*, however, detects n-way feature interactions at run-time, and only inspects a single scenario (i.e., set of variables and features) at a time.

Thoosa differs from existing feature interaction detection methods by *combining* three key elements:

- it operates at the requirements level,
- it detects n-way feature interactions, and
- it identifies which features can fail due to an interaction, and
- it only inspects a single scenario (i.e., set of variables and features) at run time, reducing computational cost.

8.5 Summary

In this chapter, we have described *Thoosa*, a run-time approach for detecting unwanted n-way feature interactions and determining features that they can cause to fail. *Thoosa* is an important practical assessment of feature interactions at run time when the design-time computational costs of n-way interactions are infeasible. Unlike previous n-way feature interaction detection approaches that detect feature interactions at run time, *Thoosa* also identifies the features that can fail due to the feature interaction that is detected. *Thoosa* reduces the effort to log errors or adapt run-time behavior by identifying which features fail in an interaction rather than simply detecting that an interaction has occurred. We have

demonstrated *Thoosa* on a braking system goal model comprising several features to realize an automotive braking system.

Chapter 9

Detecting Interactions of Non-Functional Properties

Non-functional requirements, including safety requirements, are intended to ensure the non-functional properties of the system under development. However, non-functional properties of the system often crosscut functional and non-functional requirements. These cross-cutting concerns are dispersed throughout the requirements. This dispersion renders manual insertion of the non-functional requirements difficult and error prone. One approach to address this challenge is to take an aspect-oriented modeling approach, where the cross-cutting concerns (i.e., aspects) are modeled separately from the functional concerns and then woven together, thus facilitating traceability and maintainability. This chapter introduces *Soter*, a method for aspect-oriented modeling of non-functional requirements and properties, which applies a symbolic analysis, evolutionary computation and symbolic analysis, and run time-based approaches to detect unwanted interactions between non-functional properties and/or functional requirements for cyber-physical systems. We illustrate our approach by applying *Soter* to detect unwanted interactions in the aspect-oriented safety and performance models and requirements of an industry-based automotive braking system.

9.1 Detecting Feature Interactions

Including safety requirements in a requirements specification can be challenging, especially in cyber-physical systems. The safety requirements may be violated due to unwanted system behavior in response to adverse or unexpected environmental conditions. Safety requirements, often cross-cutting, must be meticulously applied to any system scenario that could cause a violation. For high-assurance systems (e.g., automotive braking systems), ensuring safety requirements are satisfied is of paramount importance. The increasing complexity and the number of onboard features in modern vehicles further exacerbates the challenge of guaranteeing safety requirements. Aspect-Oriented Requirements Engineering (AORE) [11] provides a method of modeling non-functional requirements, including safety requirements, independent of the overall requirements model. However, issues of composition, weaving, traceability, and analyzability must be addressed to ensure safety across the suite of requirements. This chapter presents *Soter*,[1] an automated approach to detect non-functional requirement violations, including safety violations, and unwanted interactions between cross-cutting non-functional requirements and functional requirements at design time and run time.

Current methods of defining non-functional goals and properties in Goal-Oriented Requirements Engineering require explicitly defined links between non-functional requirements and functional goals and requirements in the GORE model. Goals to avoid non-functional violations, or mitigations, can be included in a requirements decomposition [87]. For example, obstacles for a given goal can be directly linked to requirements within a KAOS goal model. Mitigation of obstacles can be performed at or above the level of abstraction of the obstructed requirement to allow for both weak and strong mitigation, respectively [87]. Non-functional goals can be linked to existing goals and requirements based on their contribution to functional goals and requirements [95]. Each of these methods requires a direct connection to specific functional goals or requirements without taking into account how interactions on

[1] *Soter* is the Greek spirit of safety, preservation, and deliverance from harm.

requirements impact the subsequent contribution to other requirements in the system. While methods for defining non-functional requirements as aspects exist, they have not been used to explicitly represent important non-functional requirements like safety requirements [77], require manual detection processes [25], or only assist in identifying cross-cutting concerns [97] in a model.

This chapter describes *Soter*, an approach to detecting violated cross-cutting non-functional requirements, including safety and performance requirements to detect feature interaction at design time, using a symbolic analysis and evolutionary computation-based approach, and run time. Currently, only a limited number of methods exist to detect aspect interactions at the requirements level, despite the similarity to feature interactions [71]. *Soter* leverages the similarity between aspect and feature interactions by translating the aspect-oriented safety requirements into features representing the non-functional requirements. These additional non-functional features are woven into the existing GORE model. Feature interaction detection analysis is applied to each of the functional features and non-functional features to determine if they cause an interaction or non-functional property violation [40]. The counterexamples for each detected interaction amongst the safety features are classified according to the safety model properties and provided to the system designer to guide the revision of the functional and safety requirements. For example, an Adaptive Cruise Control may have a safety requirement to avoid collision with other cars depending on a specific proximity, while the requirements model for the ACC also has requirements to maintain the driver's desired speed. The safety requirement to avoid a collision, (i.e., a non-functional requirement) can be violated when the driver's desired speed is maintained and therefore an interaction exists between the requirements to avoid collision and maintain the driver's desired speed.

Soter provides a method for modeling aspect-oriented non-functional goals, including safety and performance goals, that includes goal decomposition strategies that provide func-

tional mitigations. *Soter* recombines the non-functional property and an optional mitigation decomposition to create non-functional features that represent one of the following cases:

- **Non-Functional Properties**: non-functional invariants of the system with no mitigation,
- **Weak Mitigations**: mitigations that are applied when non-functional properties are violated, and
- **Strong Mitigations**: mitigations that are applied to ensure non-functional properties are never violated.

These non-functional features are defined separately (i.e., as aspect models) from the traditional requirements decomposition hierarchy and are subsequently woven into the relevant portions of the requirements model automatically. Counterexamples representing interactions between functional and non-functional features are generated by a feature interaction detection tool, *Phorcys* [40], and categorized by *Soter* as to whether the non-functional properties, the mitigation objectives, or both were violated.

The contributions of this chapter are as follows:

- We propose a goal modeling approach to modeling non-functional requirements as cross-cutting concerns,
- We introduce an approach to automatically detect conflicts between non-functional requirements and functional requirements by exploiting the similarity between aspect and feature interactions, and
- We present a prototype implementation of the *Soter* approach, and demonstrate the applicability of *Soter* on an industry-based automotive braking system using non-functional safety and performance goals and requirements.

The remainder of this chapter is organized as follows. Section 9.2 describes and gives examples of the *Soter* aspect-oriented requirements model. Section 9.3 introduces the *Soter*

approach. Section 9.4 describes the application of *Soter* to an industry-based example. Section 9.5 covers related work, and Section 9.6 discusses conclusions and future work.

9.2 Representation of Non-Functional Requirements

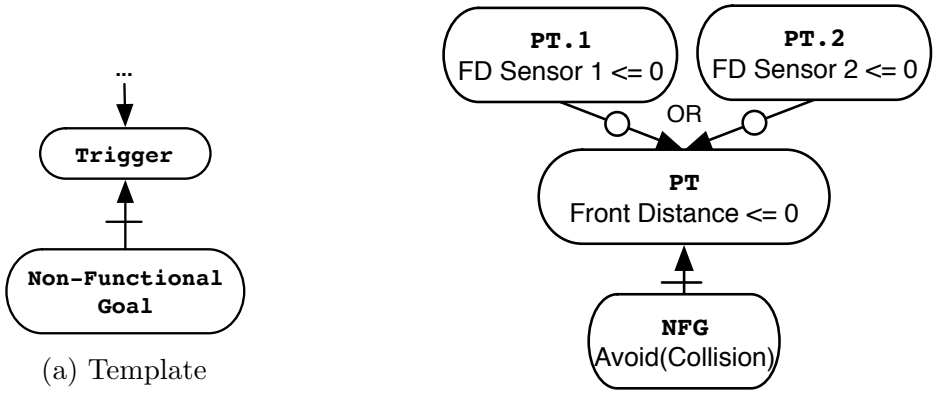
Soter supports the modeling and the analysis of three kinds of non-functional requirements models: non-functional properties, weak mitigation (of non-functional property violation), and strong mitigation (of non-functional property violation). Each of these three comprise a non-functional property trigger that assesses non-functional property violations and for the latter two model types, a goal decomposition representing the mitigation requirements. Importantly, each of these non-functional requirement models are based on a cross-cutting non-functional goal that represents a cross-cutting non-functional concern. Each of the three kinds of non-functional requirements models may be applied to a functional goal model individually or in combination with any number of additional non-functional requirements models.

9.2.1 Non-Functional Properties

Non-functional properties specify non-functional goals (e.g., 'Avoid collision') and are represented by non-functional models that use a non-functional property trigger to detect a non-functional property goal violation. For example, in Figure 9.1a, the `Non-Functional Goal` is violated when the non-functional property `Trigger` is satisfied, denoted by a single hashed directed edge from the non-functional property trigger to the non-functional goal. The non-functional property `Trigger` may be decomposed as needed.

The system, as a whole, is intended to continuously satisfy the `Non-Functional Goal` by never satisfying the non-functional property `Trigger`. For example, in Figure 9.1b, goal `PT` is the cross-cutting safety goal intended to avoid front collisions. Goal `PT`, a safety property trigger, indicates a safety goal violation if the *'Front Distance'* is zero or less, which is true

if either of the front distance sensors (*'FD Sensor 1'* or *'FD Sensor 2'*) report a value of zero or less (i.e., via goals `PT.1` or `PT.2`, respectively).



(a) Template

(b) Safety Instantiation

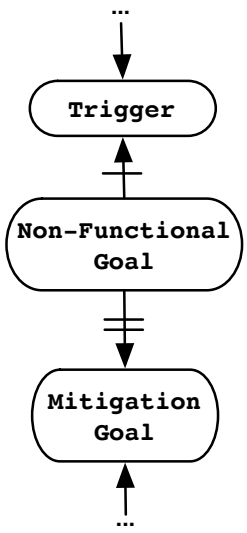
Figure 9.1: Safety Model: Non-Functional Properties

9.2.2 Weak Mitigation

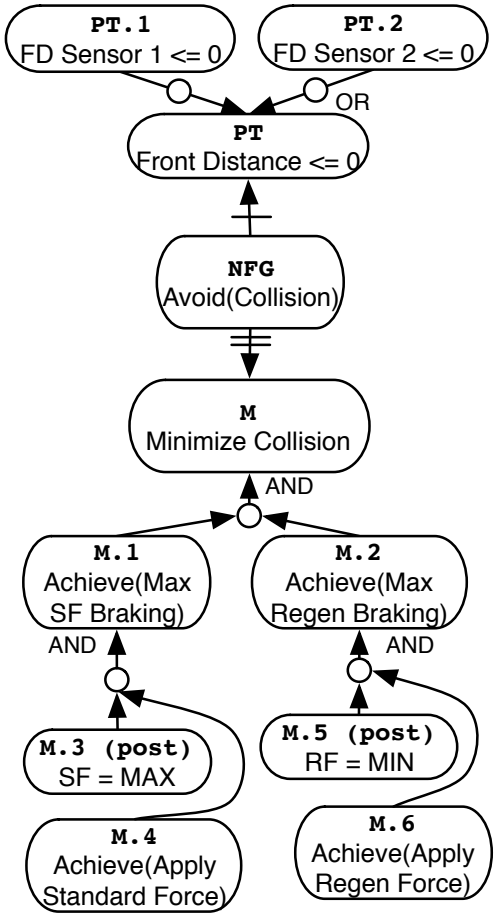
Weak mitigations are represented by non-functional requirement models that provide alternative functionality when the non-functional goal is violated as detected by the non-functional property trigger. For example, in Figure 9.2a, goal `Non-Functional Goal` is still the cross-cutting non-functional goal. The non-functional property trigger, goal `Trigger` and its derived expectations, measures violations of the non-functional goal. For this non-functional requirements model, when the non-functional goal is violated, the `Mitigation Goal` provides alternate behavior to mitigate the impact of the non-functional goal violation. The `Mitigation Goal` is identified by a double hashed edge from the `Non-Functional Goal` to the `Mitigation Goal`. These mitigation requirements may attempt to minimize the adverse impact of the violation of the non-functional goal or provide fail-safe behavior when the non-functional goal is known to be violated.

Weak mitigations are used to model non-functional requirements whose violation cannot be prevented, but necessitate an alternative behavior when the non-functional goal is violated. For example, in Figure 9.2b, a fail-safe mitigation applies full brakes in the event

of an imminent collision. The safety property trigger, decomposed from goal PT, indicates a violation of the non-functional safety goal when the distance measured by either of the front distance sensors (i.e., *'FD Sensor 1'* or *'FD Sensor 2'*) is zero. When the non-functional safety property violation occurs, the mitigation (i.e., Goal M) and its decomposed goals and requirements apply the maximum amount of standard braking (i.e., Goals M.1, M.3, and M.4) while minimizing regenerative braking (i.e., Goals M.2, M.5, M.6) to prevent the battery from being charged during a collision. Goals M.3 and M.5 are expectations expressed as post-conditions of the M.1 and M.2 requirements, respectively.



(a) Template



(b) Safety Instantiation

Figure 9.2: Non-Functional Model: Weak Mitigation

9.2.3 Strong Mitigation

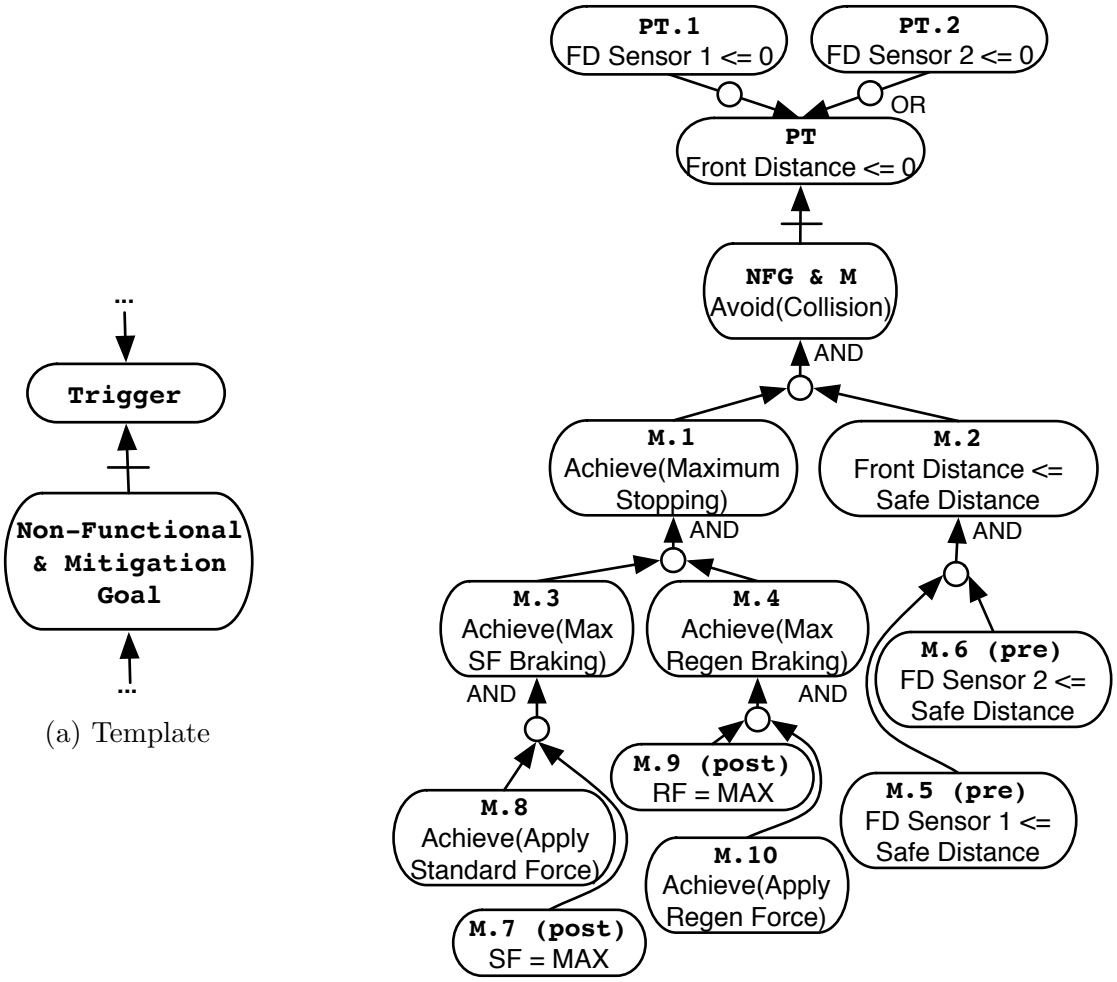
Strong mitigations are represented by non-functional requirement models that provide mitigation requirements that guarantee the non-functional goal will not be violated. For example, in Figure 9.3a the non-functional property `Trigger` measures the violation of the `Non-Functional & Mitigation Goal`. However, unlike weak mitigations, the decomposition of the mitigation is a decomposition of the non-functional goal itself. If the mitigation is satisfied, then the non-functional goal must not be violated.

Strong mitigations are used to model non-functional requirements whose violation can be prevented, but necessitate additional mitigation behavior to ensure the non-functional goal is not violated. For example, in Figure 9.3b, the non-functional safety property trigger (i.e. Goal `ST`) should never indicate that the distance measured by either of the front distance sensors is zero indicating that the non-functional safety goal (i.e., `NGF & M`) is violated. The non-functional safety goal is not violated as the mitigation itself (i.e., `NGF & M`) is decomposed from the non-functional safety goal. In Figure 9.3b, a front collision is avoided by applying the brakes well before the distance sensors measure zero when there is still adequate distance to avoid the collision (i.e., `Safe Distance` in goals `M.2`, `M.5`, and `M.6`).

9.3 Detection via Symbolic Analysis Approach

Soter is a tool for symbolically analyzing the woven goal model comprising the cross-cutting non-functional concerns and the functional requirements to detect interactions or failures. Feature interaction detection is applied to detect aspect interactions and failures in aspect-oriented non-functional requirements models (e.g., Figures 9.1b, 9.2b, and 9.3b) where non-functional requirements have been translated into features in a goal model.

An overview of the *Soter* process is shown in the DFD in Figure 9.4. Step 1 accepts the non-functional models and transforms them into features represented in terms of hierarchical goal models with pre- and post-conditions and requirements at the leaf level that can be



(a) Template

(b) Safety Instantiation

Figure 9.3: Non-Functional Model: Strong Mitigation

woven into a functional goal model (e.g., Figure 7.1). Step 2 weaves the generated non-functional features into an overall woven goal model that includes both the non-functional models and functional goal model. In Step 3, the overall woven goal model woven with the non-functional features is processed to identify feature interactions and failures. Finally, in Step 4, the counterexamples detected are classified according to the type of failure (i.e., non-functional property, mitigation, or both). The classified counterexample causes are provided to the system designer to facilitate the revision of the functional goal model and/or non-functional models. (For brevity, the updated non-functional models and functional goal model

in this section may use goal names to refer to previously defined goals without repeating their full specification.) Next, we describe the details of each of the key steps for *Soter*.

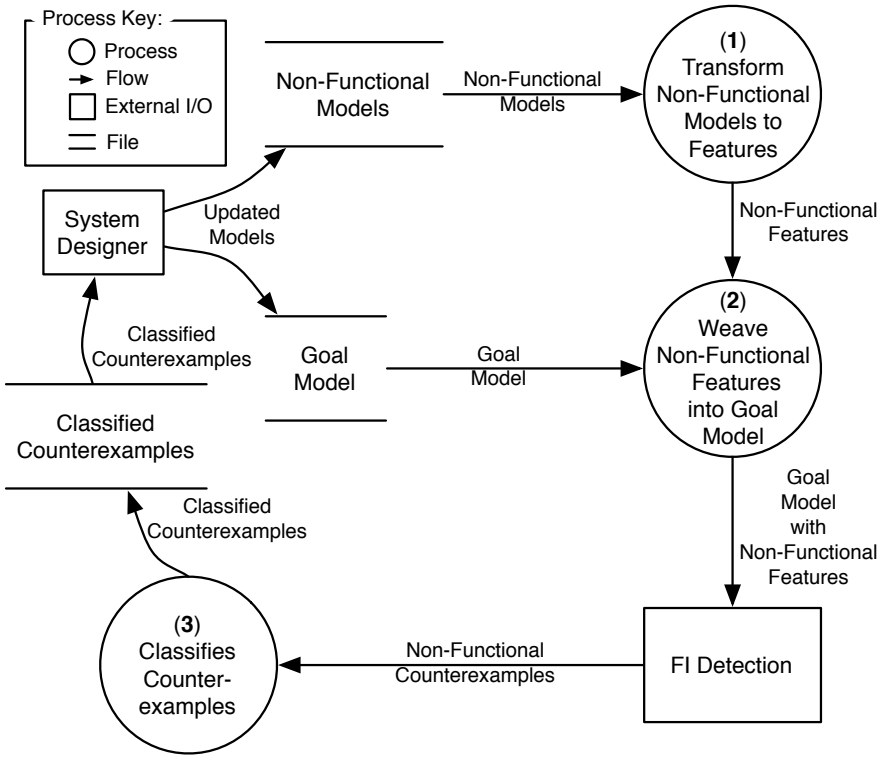


Figure 9.4: *Soter* Data Flow Diagram

Step (1): Transform Non-Functional Models to Features

Step 1 accepts the non-functional models developed by the system designer and translates them to non-functional features in terms of a hierarchical goal model. For each of the three types of non-functional models, a separate transformation process is needed for the respective non-functional feature: non-functional property features, weak mitigation features, and strong mitigation features. All three include a non-functional trigger that specifies a measure that indicates whether the aspect-oriented non-functional goal has been violated.

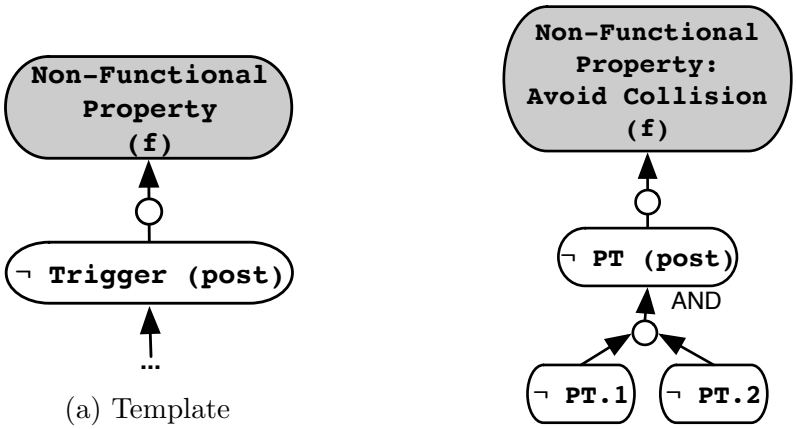
Non-Functional Properties

The complement of the non-functional property trigger specifies the properties that the system as a whole must maintain if it is not to violate the non-functional goal. A non-functional property model comprises this measure (i.e., the non-functional property trigger) and the aspect-oriented non-functional goal. The feature is modeled as the complement of the non-functional property trigger, as shown in Figure 9.5a. The only way that a non-functional property feature can be satisfied is if none of the non-functional property triggers detect a violation of the non-functional goal. Given that there are no pre-conditions for the feature, the feature's applicability is not restricted by its specification. A non-functional property model is intrinsically applicable, or crosscuts, the entire specification in which it is included.

For example, the non-functional safety model in Figure 9.1b is transformed into a feature in Figure 9.5b (as denoted by '(f)'). The shaded goals in Figure 9.5a are portions of the template that are carried forward into the instantiated feature, while the non-shaded goals are replaced by instantiated goals from the non-functional safety model (e.g., Figure 9.1b). Just as in Figure 9.5a, Figure 9.5b includes a top-level feature goal (i.e., goal `Non-Functional Property: Avoid Collision`), as well as the complement of the non-functional safety trigger (i.e., goal ¬`PT` and its decomposed goals) from the non-functional safety model in Figure 9.1b (i.e., `PT` and its decomposed goals). Notice that due to the negation of the non-functional safety property trigger, the decomposition changes from an OR-decomposition to an AND-decomposition in the non-functional safety properties feature goal decomposition. Importantly, neither goal `PT.1` nor goal `PT.2` can be violated if the non-functional safety property trigger, goal `PT`, is to remain unviolated.

Weak Mitigation

A weak mitigation non-functional model provides alternate behavior via the mitigation. Figure 9.6a, presents a template of a weak mitigation feature with two behaviors that define the translation from a safety model. In one branch, goal `Trigger and Mitigation`



(a) Template

(b) Safety Instantiation

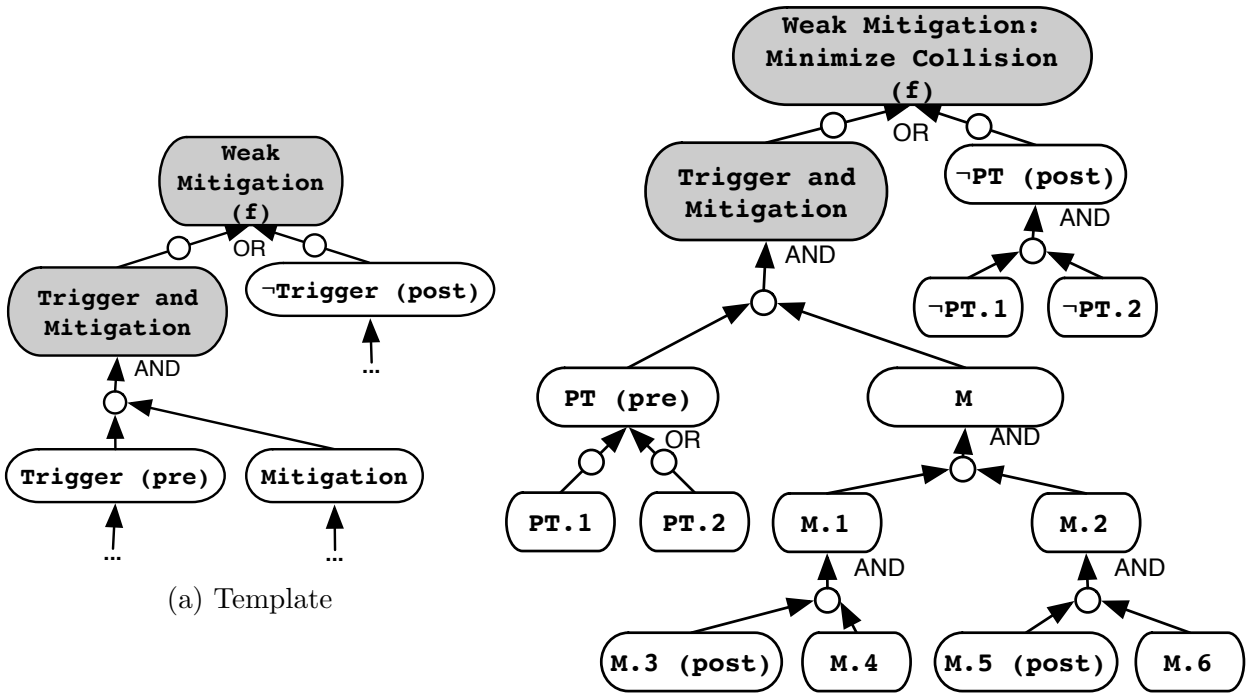
Figure 9.5: Non-Functional Feature: Non-Functional Properties

is AND-decomposed into non-functional goal violations and the corresponding mitigation, both of which must be satisfied (i.e., provide alternate functionality) to satisfy the weak mitigation feature. The second branch is the complement of non-functional properties (i.e., goal ¬`Trigger`) which is satisfied when the non-functional goal is not violated.

For example, the weak mitigation non-functional safety model in Figure 9.2b is translated into a weak mitigation non-functional safety feature in Figure 9.6b. Based on the template in Figure 9.6a, the translated weak mitigation non-functional safety feature in Figure 9.6b includes alternatives for when the non-functional safety goal is violated (i.e., goal `Trigger and Mitigation: Minimize Collision`) and when it is not violated (i.e., ¬`PT`). Functionally, the mitigation (i.e., applying the maximum brake force to both standard friction and regenerative braking) is applied if the non-functional safety goal is violated (i.e., goal `PT` is satisfied due to the distance measured by the front distance sensors). Otherwise, the mitigation is not applied if the safety goal is not violated (i.e., ¬`PT`).

Strong Mitigation

A strong mitigation non-functional model provides a mitigation that, when applied, ensures the non-functional goal is never violated. Figure 9.7a defines the translation template used to generate a strong mitigation non-functional feature. Similar to a weak mitigation non-



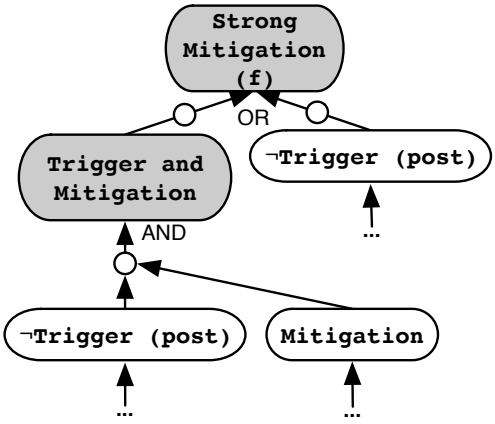
(a) Template

(b) Safety Instantiation

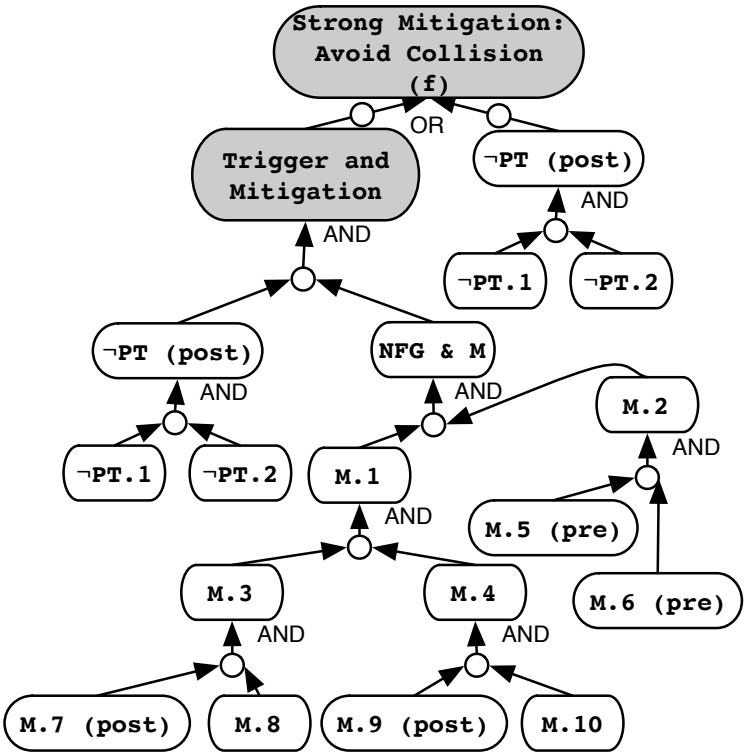
Figure 9.6: Non-Functional Feature: Weak Mitigation

functional feature, a strong mitigation non-functional feature includes two behavior cases. The first case is a mitigation that is applied when the non-functional goal is not violated (i.e., goal `Trigger and Mitigation`). In the second case, no mitigation is applied when the non-functional goal is not violated (i.e., ¬`Trigger`).

For example, the instantiated strong mitigation non-functional safety model in Figure 9.3b is translated into the feature goal model in Figure 9.7b. In both alternatives, the complement of the non-functional safety property trigger is required (i.e., goals ¬`PT`), while in goal `NFG & M`, the mitigation is also applied. Functionally, the non-functional safety feature cannot be satisfied unless the non-functional safety goal is satisfied (i.e., not violated by the complement of the non-functional safety trigger). That is, the distance as measured by the front distance sensors must not be equal to (or less than) 0. In cases where the mitigation may be applied due to a distance measured that is less than or equal to the *'Safe Distance'*, the brakes are applied.



(a) Template



(b) Safety Instantiation

Figure 9.7: Non-Functional Feature: Strong Mitigation

Step (2): Weave Non-Functional Features into Goal Model

The second step of the *Soter* process weaves the non-functional features generated from the translation in Step 1 into the functional goal model provided by the system designer (e.g., Figure 7.1). Unlike source code weaving in aspect-oriented programming, where the aspects are woven into joinpoints along the execution path by the information provided in pointcuts [88], *goal models have no implicit ordering of requirements* based solely on the structure of the model itself. Weaving safety features into a goal model is based on pointcuts described by pre-conditions of the safety feature and expressed as joinpoints of pre-conditions in the non-functional feature that are satisfied by the agents of the system or environment. Since requirements are declarative, they are globally applicable unless filtered by the pre-conditions, unlike source code where a portion of code is applicable only if it is executed. To accomplish the weaving of requirements, each non-functional feature (e.g., Figure 9.5b, 9.6b,

or 9.7b), along with the functional goal model (i.e., Figure 7.1) is AND-decomposed from a new root goal. The translated non-functional features are logically propagated to where the pre-conditions of the functional goals and that of the non-functional features are satisfied (i.e., the pre-condition is satisfied by the environmental and system instantiated values), otherwise they are not applicable.

For example, in Figure 9.8, the non-functional safety feature (`Safety Model (f)`) has been woven into the braking system goal model in Figure 7.1 at the newly introduced root (i.e., 'Woven Goal Model'). Both the non-functional safety feature (i.e., `Safety Model`) and the original functional goal model (i.e., `A`) are AND-decomposed under a single top-level goal.

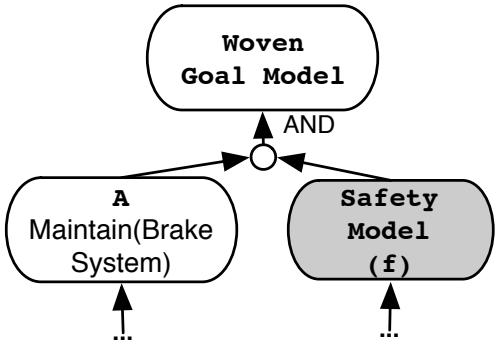


Figure 9.8: Woven Goal Model Example

In order for the `Woven Goal Model` to be satisfied, the original goal model and the non-functional safety feature must be satisfied.

FI Detection

Using the features in the goal model, including the woven safety features representing the safety models, symbolic feature interaction detection is used to detect features that cause interactions or safety failures [40]. A previously developed symbolic feature interaction detection method, *Phorcys* [40], is used to detect the interactions and failures from the woven goal model and returns a set of counterexamples.

Step (3): Classifies Counterexamples

Causes of an interaction or non-functional property violation are individual features where the post-conditions are not satisfied in a given counterexample. *Soter* categorizes these counterexamples based on the location of the unsatisfied post-conditions. If the unsatisfied post-condition exists in a functional feature, then the failure is classified as a feature interaction. However, if the unsatisfied post-condition exists in a safety feature, then *Soter* categorizes the failure based on the type of safety feature violated, as defined below.

Non-Functional Property Features

The only post-condition in a non-functional property feature is the complement of the non-functional property trigger. Given a counterexample that identifies a non-functional property feature as the feature that fails in a feature interaction, *Soter* classifies the failure as a *safety goal violation*.

For example, in the translated non-functional safety property feature in Figure 9.5b, if the counterexample shows that at least one of the front distance sensors (e.g., *'Front Distance Sensor 1'* or *'Front Distance Sensor 2'*) is less than or equal to zero, then the safety goal `SG` in Figure 9.1b has been violated.

Weak Mitigation Features

In weak mitigation features, two sets of post-conditions may be violated and can be traced back to the non-functional model: the complement of the non-functional property trigger and the post-conditions that exist in the weak mitigation. *Soter* classifies failure based on the post-conditions that are violated as follows:

- **The complement of the non-functional property trigger is violated** indicating the mitigation pre-conditions must be violated and the non-functional model is not sufficient to provide alternate functionality in all cases where the non-functional goal is measured as violated by the non-functional property triggers.

- **The complement of the non-functional property trigger *and* the mitigation post-condition is violated** indicating an interaction exists with the mitigation functionality that prevents alternate functionality in all cases where the non-functional goal is measured as violated by the safety triggers.

When the mitigation post-condition is violated, it is assumed that the complement of the non-functional property triggers are also violated. Otherwise the feature could be satisfied by the complement of the non-functional property triggers alone.

Strong Mitigation Features

Strong mitigation features, similar to weak mitigation features, have the same two sets of post-conditions that may be violated and traced back to the original non-functional model (i.e., the complement of the non-functional property trigger and the mitigation post-conditions). However, the complement of the non-functional property triggers are used as post-conditions in both alternatives of the non-functional model (i.e., ¬`Trigger` in Figure 9.7a). *Soter* classifies the failures based on the post-conditions that are violated as follows:

- **The complement of the non-functional property trigger is violated** indicating the mitigation is unable to prevent the violation of the non-functional goal in the non-functional model as measured by the non-functional property triggers.

- **The mitigation post-condition is violated** indicating an interaction exists that prevents the mitigation from satisfying its specification, but has not caused a non-functional goal violation as measured by the non-functional property triggers.

- **The complement of the non-functional property trigger *and* the mitigation post-condition is violated** indicating an interaction exists with the mitigation functionality that prevents the mitigation from ensuring the non-functional goal is not violated as measured by the non-functional property triggers.

System Designer Revisions

The analysis results, including the failure caused by the unwanted interaction and the non-functional property violation is provided to the system designer for manual revision. For each of the possible combinations of non-functional model type and failure or interaction source, a possible resolution is provided in Table 9.1. The resolution is not identified automatically by *Soter*, but rather is identified by the system designer. It may also be necessary to weaken non-functional property triggers or even mitigation model type, if non-functional goals are not practically achievable within the system specification.

Table 9.1: Possible Failure Resolutions

#	**Model Type**	**Failure Source**	**Possible Resolution**
1	Non-Functional Property	Trigger	Weaken non-functional goal or update goal model.
2	Weak Mitigation	Trigger	Weaken pre-condition of mitigation.
3	Weak Mitigation	Trigger, Mitigation	Update mitigation and/or goal model to prevent interaction.
4	Strong Mitigation	Trigger	Weaken mitigation pre-condition.
5	Strong Mitigation	Mitigation	Update mitigation and/or goal model to prevent interaction.
6	Strong Mitigation	Trigger, Mitigation	Update mitigation and/or goal model to prevent interaction.
7	Functional	Feature	Update goal model and/or non-functional mitigation.

Additionally, the type of non-functional model initially used may not be sufficient. For example, if a non-functional property is violated then it may be necessary to provide a weak or strong mitigation. It is suggested to first implement a non-functional model as a non-functional property (i.e., without a mitigation) to determine if the non-functional property is ever violated. If it is, then a strong mitigation is the most appropriate mitigation since a strong mitigation ensures the original non-functional property would never be violated. However, if no mitigation exists that prevents the non-functional violation, then it is im-

portant to minimize the negative impact via a weak mitigation that is applied when the non-functional violation occurs.

Updates to the aspect-oriented non-functional models or functional goal model that are performed by the system designer may be analyzed again by *Soter* to determine whether there exist remaining or newly-introduced interactions or failures.

9.4 Detection of Interactions

This section details the detection of interactions and failures caused by the non-functional safety models developed in Sections 9.2 and 9.3 and their impact on the functional model. This subsection can be viewed as a complete example of *Soter*, as it illustrates the detection and mitigation of interactions caused by the non-functional and functional models. Additionally, we develop non-functional models for an additional non-functional safety goal and a non-functional performance goal to illustrate the generally applicable nature of *Soter* by creating non-functional property models, non-functional weak mitigation models, and non-functional strong mitigation models. Finally, three of the non-functional models (two safety and one performance) are woven into the functional model and any interactions are detected.

9.4.1 Safety Case Study: Collision

This subsection gives the results of applying *Soter* to aspect-oriented safety models related to collisions for the braking system goal model in Figure 7.1. *Soter* is used to detect safety failures, safety model interactions with functional requirements, and interactions between safety models. The safety models used in this example application are defined by the system designer and represent three different non-functional goals intended to avoid or minimize the effect of a collision.

Avoid Collision via Safety Property

Initially, the non-functional safety property model in Figure 9.5b (translated from Figure 9.1b) is woven into the functional goal model in Figure 7.1 in order to detect any feature interactions or failures caused by the safety properties necessary to avoid a frontal collision. An excerpt of the woven goal model, including the functional braking system and the safety properties model is shown in Figure 9.9. This initial woven model is referred to as M.

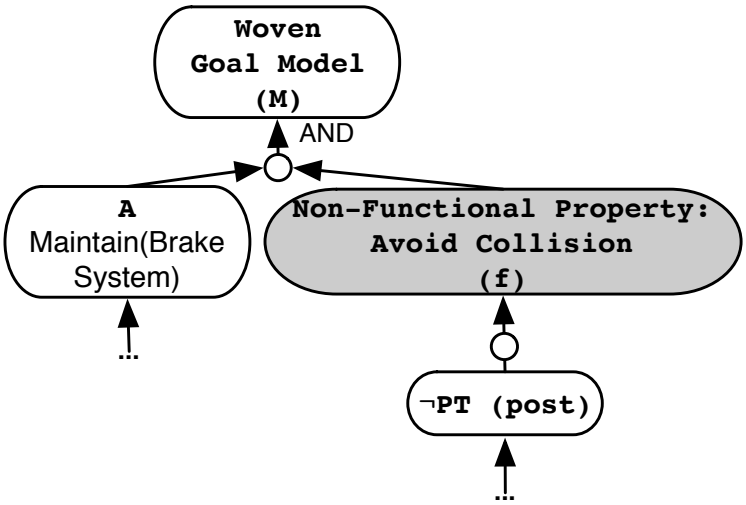


Figure 9.9: Goal Model Woven With Non-Functional Safety Property (M)

Applying *Soter* to detect interactions and failures caused by the safety model results in a counterexample that includes a violation of the safety model without an interaction (i.e., a safety failure). In this counterexample, one of the front distance sensors (*FD Sensor 1*) results in a distance of 0.0. This safety goal violation is caused by there being nothing in the functional goal model (i.e., Figure 7.1) that applies the brakes based on information from the distance sensors.

Avoid Collision via Strong Mitigation

In order to avoid a safety goal violation, a mitigation must be employed to avoid the violation. Accordingly, we replace the non-functional safety property model, which only detects the violation of the non-functional safety properties, with a strong mitigation model

that attempts to avoid the violation entirely. The strong mitigation model in Figure 9.7b (translated from Figure 9.3b) attempts to apply the brakes before an object becomes too close (i.e., causes a front collision) to the front distance sensors. The new woven goal model, including the functional braking system and the strong mitigation model, is shown in Figure 9.10. This updated woven model is referred to as M'.

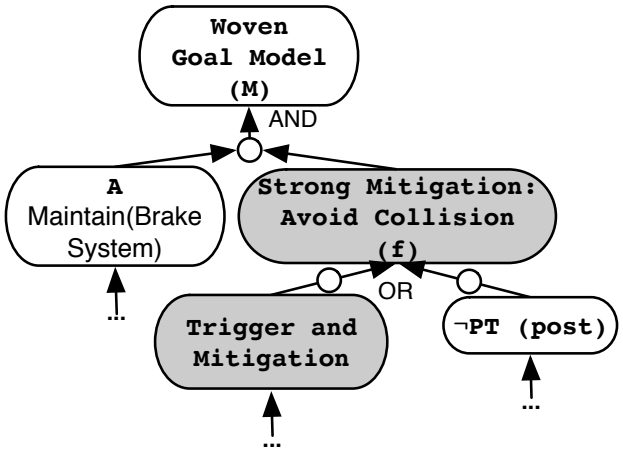


Figure 9.10: Goal Model Woven With Strong Mitigation (M')

Applying *Soter* to detect interactions and failures caused by the safety model results in an interaction detected with features `B` and `E`. Table 9.2 provides the values of the variables in the counterexample, and indicates by which post-conditions a given variable is constrained.

Table 9.2: Counterexample: Variables for Interaction in Goal Model M'

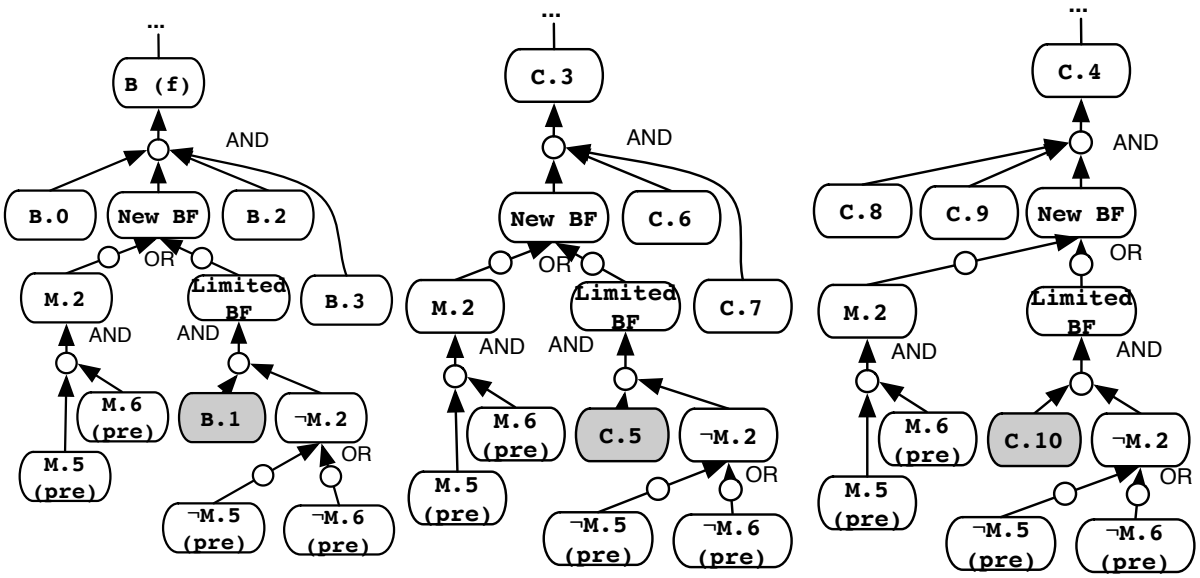
Variable	Value	Constrained In
BF	0.40	
CBF	0.40	`E.3`, `D.11`, `D.13`
DS1	0.0	
DS2	0.0	
RF	0.32	`C.10`, `SG.9`
SF	0.08	`B.1`, `C.5`, `SG.7`
SS	false	`E.6`, `D.6`

While features `B` and `E` are satisfied, the strong mitigation model is violated. Specifically, goals `SG.7` and `SG.9` are violated since they are unable to set regenerative brake force (RF) or standard brake force (SF) to the MAX value (100.0). Further, the safety trigger is satisfied

(i.e., a safety goal violation) since the front distance sensors both read 0.0 (*DS1* and *DS2*). Given the source of the cause and the safety model type, Table 9.1, row 6 suggests an update to either the mitigation or the goal model could be used to remove the interaction.

The purpose of the mitigation in the safety model in Figure 9.3b is to prevent the vehicle from violating the safety goal by applying brake force before the vehicle is involved in a collision. Since the safety goal is violated and we do not want to permit collisions, an update to allow the mitigation is necessary, and the functional goal model must be updated. Functionally, any expectation that changes the value of standard or regenerative braking force (*SF* or *RF*), must only be applied when the mitigation from the strong mitigation model is not applied. The functional goals that change the braking force values are `B.1`, `C.5`, and `C.10`. Each of these goals is replaced with a system designer-defined goal decomposition excerpt to remove the conflict when changing the braking force values. The update for `B.1` is AND-decomposed from goal `B`, taking the position previously occupied by `B.1`, while goal `B.1` moves to the gray goal as shown in Figure 9.11a. Similarly, both expectations `C.5` and `C.10` that are AND decomposed from goals `C.3` and `C.4` are updated in Figures 9.11b and 9.11c, respectively.

The updates to goals `B.1`, `C.5`, and `C.10` using the update excerpts in Figures 9.11a, 9.11b, and 9.11c result in an updated goal model, referred to as M''. However, when *Soter* analyzes goal model M'', the mitigation no longer fails due to an interaction, but the safety trigger can still indicate a safety goal violation. The counterexample variable values for this safety goal violation are shown in Table 9.3. Importantly *DS1* and *DS2*, the distance sensors, both show a distance of zero, indicating a collision despite maximum (*MAX*) braking for both regenerative and standard braking (*RF* and *SF*). Unfortunately, the strong mitigation is not sufficient to *guarantee* the prevention of a collision.



(a) B.1 Excerpt (b) C.5 Excerpt (c) C.10 Excerpt

Figure 9.11: Brake Force Goal Update Excerpts of M''

Table 9.3: Counterexample: Variables for Failure in Goal Model M''

Variable	Value	Constrained In
BF	0.40	
CBF	0.40	E.3, D.11, D.13
DS1	0.0	
DS2	0.0	
RF	*MAX*	C.10, SG.9
SF	*MAX*	B.1, C.5, SG.7
SS	false	E.6, D.6

Minimize Collision via Weak Mitigation

Without significant additional sensors and actuators, it is not possible to prevent a collision simply by braking. For example, a completely stopped vehicle may still be unable to avoid a frontal collision if another vehicle drives into it. In the event of a collision, it is desirable to have a fail-safe, or behavior intended to preserve safe operation. In the case of regenerative brakes, there is the possibility that batteries could be damaged in a collision making it unsafe to recharge them. A weak mitigation model, such as the one in Figure 9.2b could be used as a fail-safe to ensure that when the safety goal is violated, the regenerative

braking force (RF) is set as low as possible to ensure no generation of electricity to recharge the batteries while maximizing the standard braking force (SF) in order to prevent charging damaged batteries. Updating the woven goal model by weaving both the weak and strong mitigations yields the goal model in Figure 9.12, referred to as M'''.

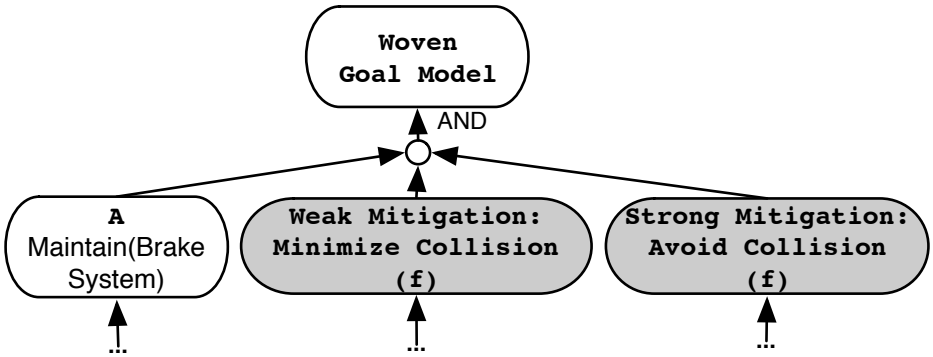


Figure 9.12: Model Woven With Weak & Strong Mitigation (M''')

Applying *Soter* to the goal model M''' where, both the weak and strong mitigation models have been woven indicates that *both* can fail due to an interaction. In the case where the front distance sensors read 0.0, the weak mitigation model attempts to set the regenerative brake force to MIN, while the strong mitigation attempts to set the same regenerative brake force to MAX, resulting in an interaction between safety mitigations. When the strong mitigation is the cause, the values of the counterexample variables are as shown in Table 9.4.

Table 9.4: Counterexample: Variables for Failure in Goal Model M'''

Variable	**Value**	**Constrained In**
BF	0.40	
CBF	0.40	`E.3`, `D.11`, `D.13`
DS1	0.0	
DS2	0.0	
RF	*MIN*	`C.10`, `SG.9`
SF	*MAX*	`B.1`, `C.5`, `SG.7`
SS	false	`E.6`, `D.6`

Since the strong mitigation is unable to prevent the violation of the safety goal, it should have been removed when the weak mitigation was put in place. The model woven with only

the weak mitigation is shown in Figure 9.13, referred to as M^{final}. Applying *Soter* to model M^{final}, which after analysis by *Soter*, has no interactions or safety violations.

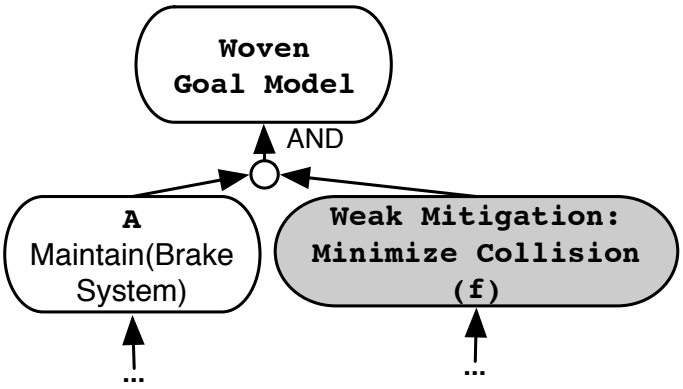


Figure 9.13: Model Woven With Updated Weak Mitigation (M^{final})

Summary

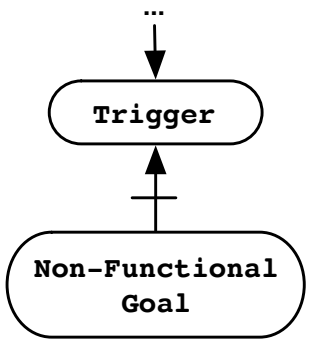
We have demonstrated that *Soter* can detect safety model violations (e.g., M'), feature interactions between safety models and functional requirements (e.g., M''), and feature interactions between multiple safety models (e.g., M''').

9.4.2 Safety Case Study: Battery Charging

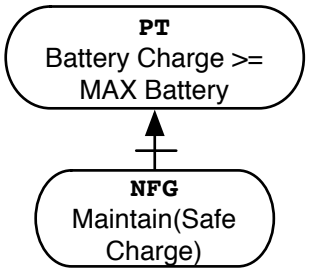
This subsection gives the results of applying *Soter* to aspect-oriented safety models related to battery charging for the braking system goal model in Figure 7.1. *Soter* is used to detect safety failures, safety model interactions with functional requirements, and interactions between safety models. The safety models used in this example application are defined by the system designer. The safety goal modeled in this subsection is intended to ensure that the battery is never overcharged, in order to prevent battery damage that could lead to fire or explosion.

Maintain Safe Charge via Safety Property

Initially, a non-functional safety property is created (i.e., as shown in Figure 9.14b) based on the template of a non-functional property (i.e., as shown in Figure 9.14a). The non-functional safety property assesses one condition: if the battery has been charged to 100% or more (i.e., Battery Charge $\geqslant$ 1.0). If that property is violated, then the safety property is violated and must be mitigated.



(a) Template



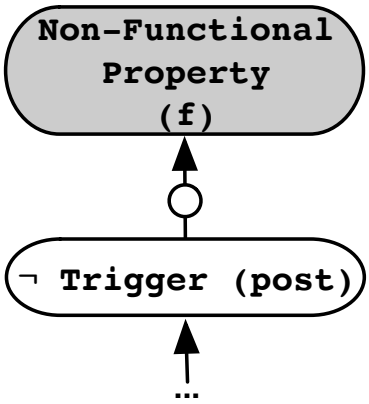
(b) Battery Safety Instantiation

Figure 9.14: Safety Model: Non-Functional Properties

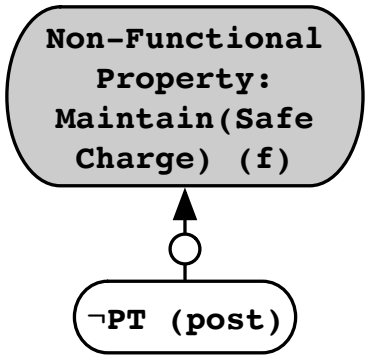
The non-functional safety property model defined in Figure 9.14b is converted by *Soter* into a *feature* by applying the template in Figure 9.15a, resulting in the non-functional safety property feature defined in Figure 9.15b. As expected, the feature can only be satisfied if the non-functional trigger is *not* satisfied, that is, when the battery is *not* charged to or beyond 100%. Due to the relative simplicity of the non-functional safety property, the translation of the non-functional safety model to the feature instantiation in Figure 9.15b only requires the negation of the single non-functional property.

Once *Soter* translates the non-functional safety property model in Figure 9.14b into the feature represented in Figure 9.15b, the feature is woven into the functional goal model by creating a new top-level goal and AND decomposing the new non-functional feature in Figure 9.15b with the functional goal model in Figure 7.1. The complete woven goal model is shown in Figure 9.16.

Soter analyzes the woven goal model in Figure 9.16 to detect any feature interactions



(a) Template



(b) Battery Safety Instantiation

Figure 9.15: Non-Functional Feature: Non-Functional Properties

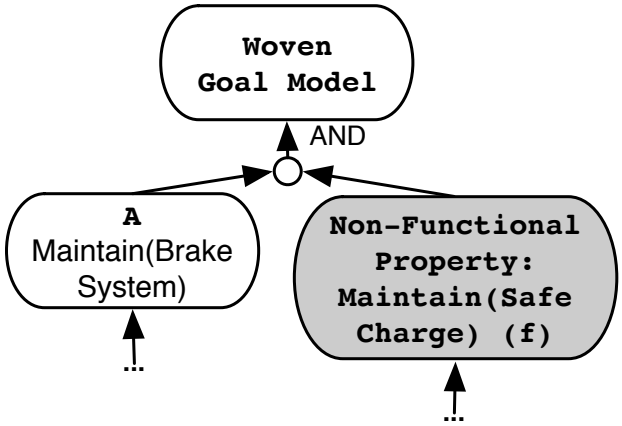


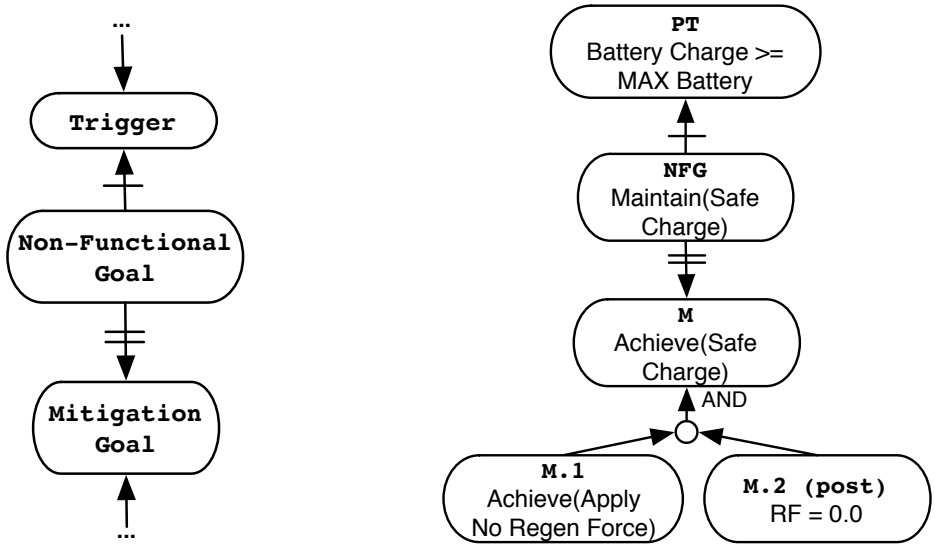
Figure 9.16: Goal Model Woven With Non-Functional Battery Safety Property

or failures of the non-functional feature woven into the functional goal model. Since it is still possible to use the regenerative brakes, which would result in a continued charging of the battery, the non-functional safety property is able to fail. Therefore, a mitigation of some form is required.

Maintain Safe Charge via Weak Mitigation

After the failure of the non-functional safety property, it is necessary to perform some kind of mitigation. In this case, we attempt to create a weak mitigation (as shown in Figure 9.17b) based on the template of a non-functional weak mitigation (i.e., as shown in Figure 9.17a). The effect of a weak mitigation is that in the event of a non-functional prop-

erty violation, in this case the violation of a safety property related to the amount a battery is charged, the mitigation is required in an effort to minimize the negative impact of the non-functional property violation. This weak mitigation works to ensure the regenerative force braking is set to no force (i.e., 0.0) to ensure that the potentially damaged batteries are no longer being charged.

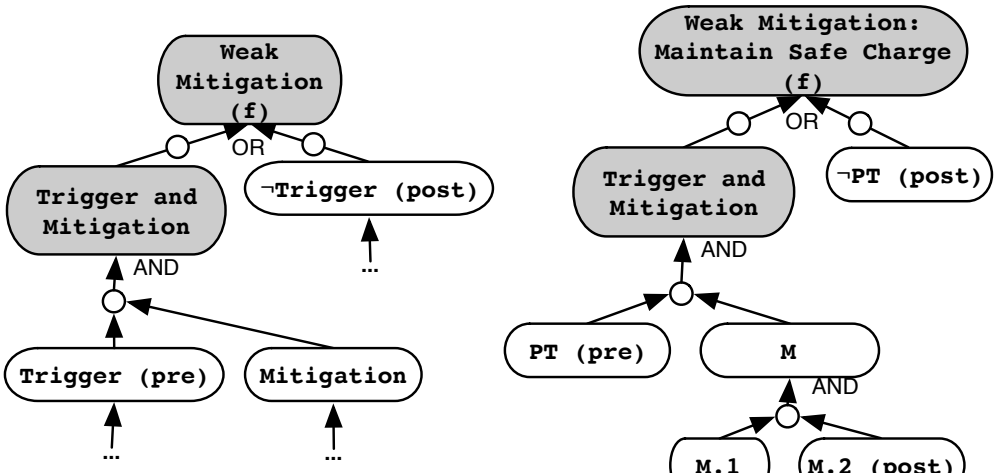


(a) Template (b) Battery Safety Instantiation

Figure 9.17: Non-Functional Model: Weak Mitigation

The non-functional safety weak mitigation model defined in Figure 9.17b is converted by *Soter* into a *feature* by applying the template in Figure 9.18a, resulting in the non-functional safety weak mitigation feature defined in Figure 9.18b. In order for the feature representing the non-functional safety weak mitigation model to be satisfied, one of two conditions must be true: either the non-functional trigger (i.e., ¬PT) is not violated *or* the non-functional trigger is violated (i.e., PT) *and* the mitigation is performed. That is, when the non-functional trigger indicates a violation, the mitigation (i.e., setting the regenerative force to 0.0 to prevent battery charging) is required.

Once *Soter* translates the non-functional weak mitigation model in Figure 9.17b into the feature represented in Figure 9.18b, the feature is woven into the functional goal model by creating a new top-level goal and AND decomposing the new non-functional feature in



(a) Template

(b) Battery Safety Instantiation

Figure 9.18: Non-Functional Feature: Weak Mitigation

Figure 9.18b with the functional goal model in Figure 7.1. The complete woven goal model is shown in Figure 9.19.

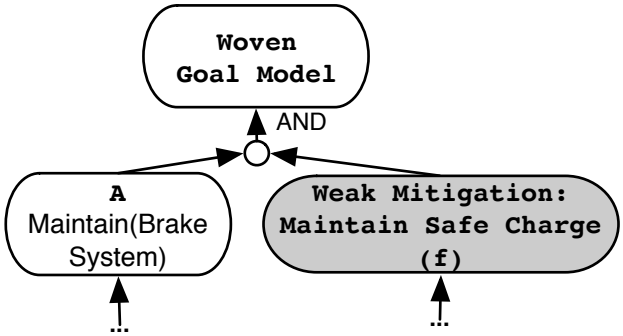


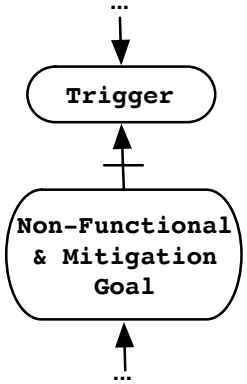
Figure 9.19: Goal Model Woven With Non-Functional Battery Safety Weak Mitigation

Soter analyzes the woven goal model in Figure 9.19 to detect any feature interactions or failures of the non-functional feature woven into the functional goal model. Since it is still possible to use the regenerative brakes, which would result in a continued charging of the battery, the non-functional safety property is able to fail due to an interaction when the non-functional property is violated, thus requiring a mitigation. However, the mitigation (specifically `M.2`) conflicts with goal `C.10` (i.e., '$RF = 0.8 * CBF$') in the functional goal model. However, a pre-condition requiring a non-violated non-functional property (i.e., ¬`PT`)

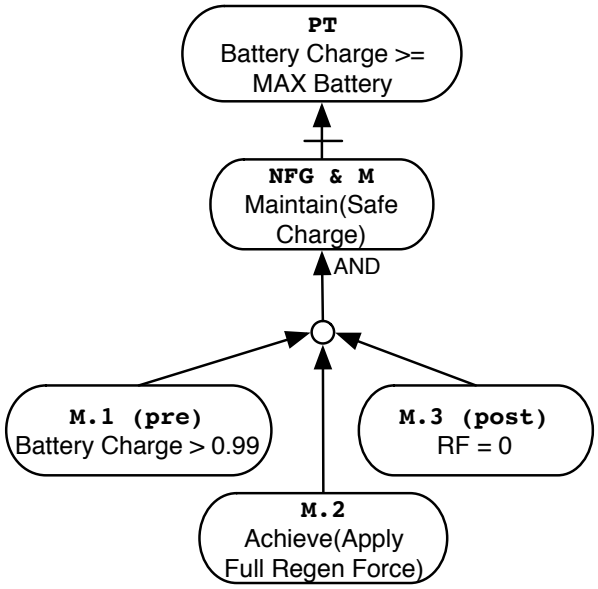
can be added to the children of goal C.4 to ensure the conflict does not occur. That is, goal C.10 would never execute when M.2 was required as a mitigation. The update ensures the battery is no longer charging once it has become dangerous to do so.

Maintain Safe Charge via Strong Mitigation

Based on the success of the weak mitigation in preventing additional charging, it may be possible to mitigate issues with battery charging *before* a safety violation even takes place. In this case, we attempt to create a strong mitigation (as shown in Figure 9.20b) based on the template of a non-functional strong mitigation (i.e., as shown in Figure 9.20a). The effect of a strong mitigation is that the mitigation should prevent the non-functional property violation from ever occurring. In this case, the violation of a safety property is related to the amount a battery is charged, and the mitigation is required to ensure the negative impact of the non-functional property violation never occurs. Rather than wait until the non-functional safety property has been violated, the regenerative braking force is limited *before* the violation occurs.



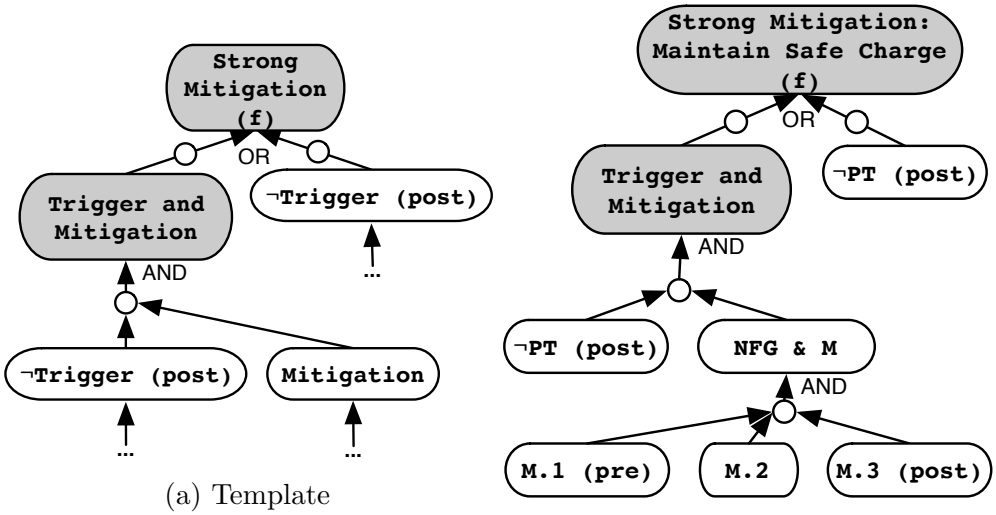
(a) Template



(b) Battery Safety Instantiation

Figure 9.20: Non-Functional Model: Strong Mitigation

The non-functional safety strong mitigation model defined in Figure 9.20b is converted by *Soter* into a *feature* by applying the template in Figure 9.21a, resulting in the non-functional safety strong mitigation feature defined in Figure 9.21b. In order for the feature representing the non-functional safety strong mitigation model to be satisfied, one of two conditions must be true: either the non-functional trigger (i.e., ¬PT) is not violated *or* the non-functional trigger is not violated *and* the mitigation is performed. That is, when strong mitigation is performed, the non-functional trigger should not indicate a violation. Instead, the mitigation (i.e., setting the regenerative force to 0.0 to prevent battery charging) prevents the violation *before* it takes place.



(a) Template

(b) Battery Safety Instantiation

Figure 9.21: Non-Functional Feature: Strong Mitigation

Once *Soter* translates the non-functional strong mitigation model in Figure 9.20b into the feature represented in Figure 9.21b, the feature is woven into the functional goal model by creating a new top-level goal and AND decomposing the new non-functional feature in Figure 9.21b with the functional goal model in Figure 7.1. The complete woven goal model is shown in Figure 9.22.

Soter analyzes the woven goal model in Figure 9.22 to detect any feature interactions or failures of the non-functional feature woven into the functional goal model. Since it is still possible to use the regenerative brakes, which would result in a continued charging of the

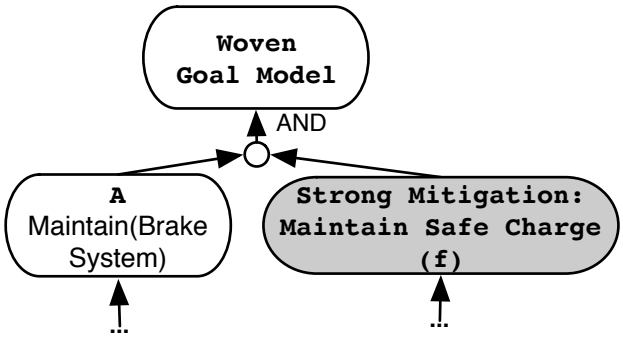


Figure 9.22: Goal Model Woven With Non-Functional Battery Safety Strong Mitigation

battery, the non-functional safety property is able to fail due to an interaction when the non-functional property is violated, requiring a mitigation. However, the mitigation (specifically `M.3`) conflicts with goal `C.10` in the functional goal, just as goal `C.10` conflicted in the weak mitigation. However, just as before, a pre-condition can be added to ensure that when `M.3` is mitigating, `C.10` is not setting the regenerative brake force. Goal `M.3` only sets the regenerative brake force when `M.1` is true, so adding ¬`M.1` as a pre-condition to the children of goal `C.4` ensures the conflict does not occur. That is, goal `C.10` would never execute when `M.2` is required as a mitigation. The update ensures the battery is no longer charged once it has become close to violating the non-functional safety property.

Summary

We have demonstrated that *Soter* can detect safety interactions with functional requirements symbolically by demonstrating each of the three non-functional model types (property, weak mitigation, and strong mitigation). Using the strong mitigation, we can prevent the safety violation from ever occurring, making strong mitigation the clearly appropriate non-functional safety model to select since it can completely prevent the safety violation.

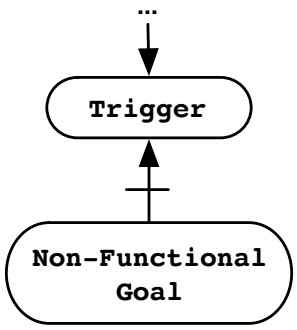
9.4.3 Performance Case Study

This subsection gives the results of applying *Soter* to aspect-oriented performance models for the braking system goal model in Figure 7.1. *Soter* is used to detect performance

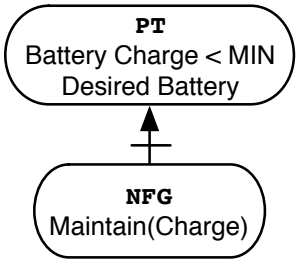
interactions and failures. The performance models used in this example application are defined by the system designer. The performance goals modeled in this subsection are intended to ensure that the battery is always charged to some minimal acceptable level to ensure that the hybrid drive functionality is ensured to operate.

Maintain Charge via Safety Property

Initially, a non-functional performance property is created (as shown in Figure 9.23b) based on the template of a non-functional property (as shown in Figure 9.23a). The non-functional performance property assesses one characteristic: if the battery has been discharged to less than 50% (i.e., Battery Charge < 50). If that property is violated, then the performance property is violated and must be mitigated.



(a) Template

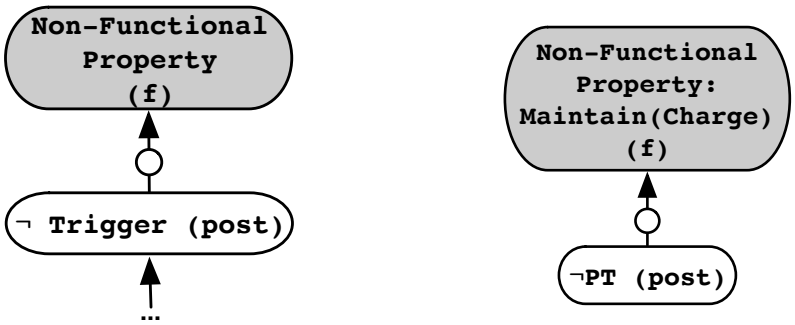


(b) Safety Instantiation

Figure 9.23: Safety Model: Non-Functional Properties

The non-functional performance property model defined in Figure 9.23b is converted by *Soter* into a *feature* by applying the template in Figure 9.24a, resulting in the non-functional performance property feature defined in Figure 9.24b. The feature can only be satisfied if the non-functional trigger is *not* satisfied, that is, when the battery is *not* discharged to less than 50%. Due to the relative simplicity of the non-functional performance property, the translation of the non-functional performance model to the feature instantiation in Figure 9.24b only requires the negation of the single non-functional property.

Once *Soter* translates the non-functional performance property model in Figure 9.23b



(a) Template

(b) Performance Instantiation

Figure 9.24: Non-Functional Feature: Non-Functional Properties

into the feature represented in Figure 9.24b, the feature is woven into the functional goal model by creating a new top-level goal and AND decomposing the new non-functional feature in Figure 9.24b with the functional goal model in Figure 7.1. The complete woven goal model is shown in Figure 9.25.

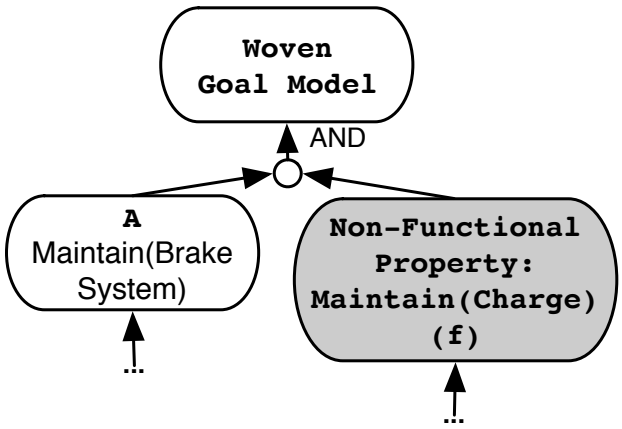
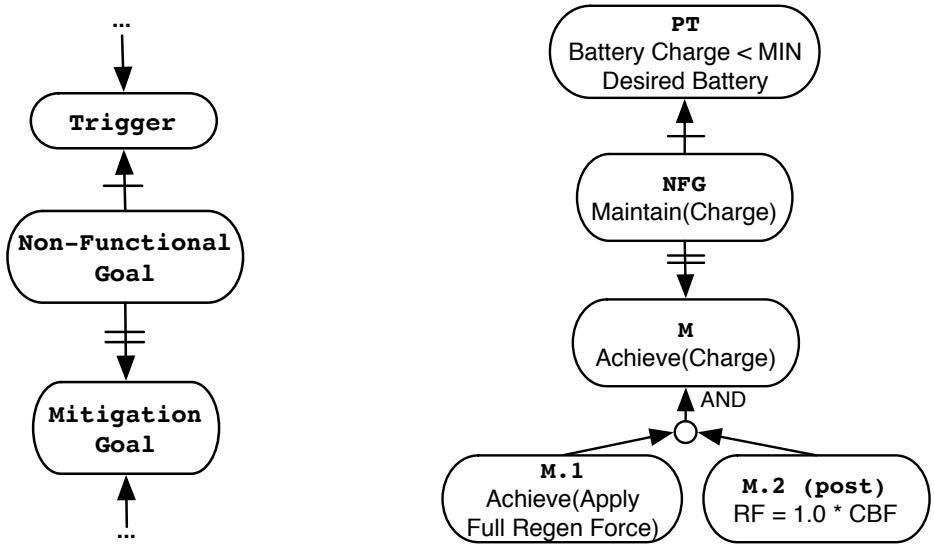


Figure 9.25: Goal Model Woven With Non-Functional Battery Performance Property

Soter analyzes the woven goal model in Figure 9.25 to detect any feature interactions or failures of the non-functional feature woven into the functional goal model. Since it is still possible to discharge the battery when driving, which may not be made up by subsequent braking, the non-functional safety property is able to fail. Therefore, a mitigation of some form is required.

Maintain Charge via Weak Mitigation

After the failure of the non-functional performance property, it is necessary to perform some kind of mitigation. In this case, we attempt to create a weak mitigation (as shown in Figure 9.26b) based on the template of a non-functional weak mitigation (as shown in Figure 9.26a). The effect of a weak mitigation is that in the event of a non-functional property violation, in this case, the violation of a performance property related to the amount a battery is charged, the mitigation is required in an effort to minimize the impact of the non-functional property violation. This weak mitigation works to ensure the regenerative force braking is set to the entire commanded brake force to ensure that the batteries are charged as much as possible.

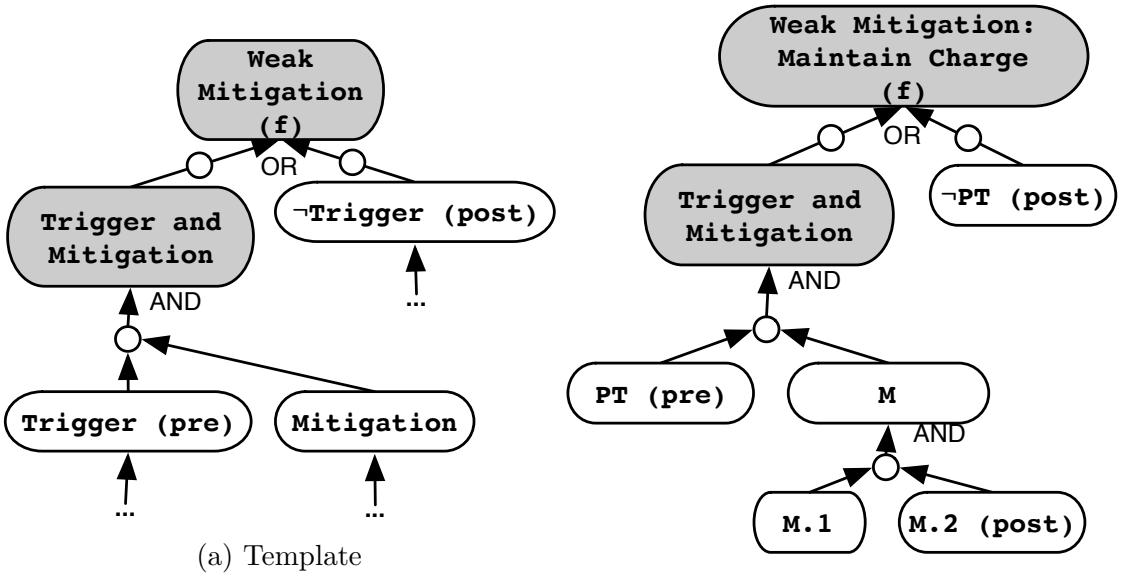


(a) Template (b) Performance Instantiation

Figure 9.26: Non-Functional Model: Weak Mitigation

The non-functional performance weak mitigation model defined in Figure 9.26b is converted by *Soter* into a *feature* by applying the template in Figure 9.27a, resulting in the non-functional performance weak mitigation feature defined in Figure 9.27b. In order for the feature representing the non-functional performance weak mitigation model to be satisfied, one of two properties must be true: either the non-functional trigger (i.e., ¬PT) is not violated *or* the non-functional trigger is violated (i.e., PT) *and* the mitigation is performed.

That is, when the non-functional trigger indicates a violation, the mitigation (i.e., setting the regenerative force to the entire commanded brake force to maximize battery charging) is required.



(a) Template

(b) Performance Instantiation

Figure 9.27: Non-Functional Feature: Weak Mitigation

Once *Soter* translates the non-functional weak mitigation model in Figure 9.26b into the feature represented in Figure 9.27b, the feature is woven into the functional goal model by creating a new top-level goal and AND decomposing the new non-functional feature in Figure 9.27b with the functional goal model in Figure 7.1. The complete woven goal model is shown in Figure 9.28.

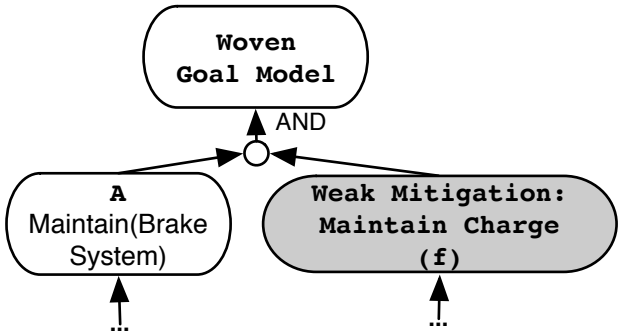


Figure 9.28: Goal Model Woven With Non-Functional Battery Performance Weak Mitigation

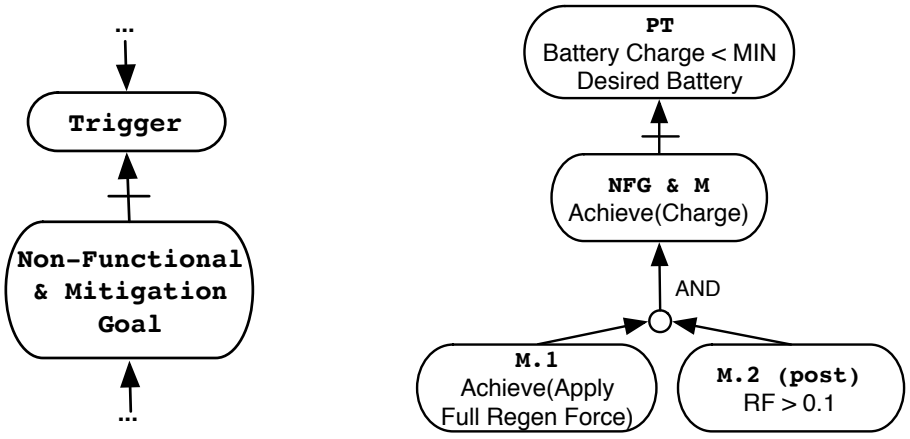
Soter analyzes the woven goal model in Figure 9.28 to detect any feature interactions

or failures of the non-functional feature woven into the functional goal model. Since it is still possible to drive the car without braking, which would result in a continued discharging of the battery, the non-functional performance property is able to fail due to an interaction when the non-functional property is violated, requiring a mitigation. However, the mitigation (specifically `M.2`) conflicts with goal `C.10` (i.e., '$RF = 0.8 * CBF$') in the functional goal. However, a pre-condition requiring a non-violated non-functional property (i.e., $\neg$`PT`) can be added to the children of goal `C` to ensure the conflict does not occur by allowing the mitigation to provide all the commanded brake force when the performance property is violated. That is, goal `C.10` would never execute when `M.2` is required as a mitigation. The update ensures the battery is charged once it has become low enough to be below the desired charge level. While the mitigation cannot prevent further discharging of the battery, it is unlikely that a strong mitigation will be found that guarantees the performance non-functional property will remain unviolated.

Maintain Charge via Strong Mitigation

In this case, we attempt to create a strong mitigation to prevent the non-functional performance property from every being violated. The strong mitigation is shown in Figure 9.29b) and is based on the template of a non-functional strong mitigation (as shown in Figure 9.29a). The effect of a strong mitigation is that the mitigation should prevent the non-functional property violation from ever occurring. In this case, the violation of a performance property is related to the amount a battery is charged, and the mitigation is required to ensure the negative impact of the non-functional property violation never occurs. Rather than wait until the non-functional safety property has been violated, the regenerative braking force is used to charge the battery *before* the violation.

The non-functional performance strong mitigation model defined in Figure 9.29b is converted by *Soter* into a *feature* by applying the template in Figure 9.30a, resulting in the non-functional performance strong mitigation feature defined in Figure 9.30b. In order



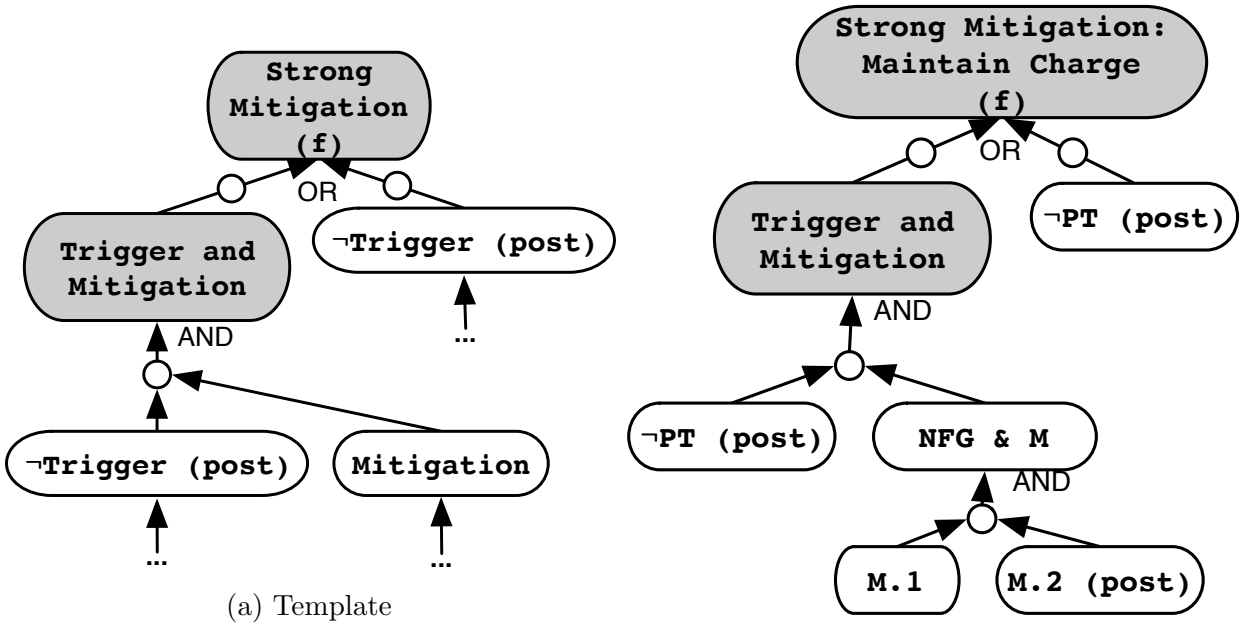
(a) Template (b) Performance Instantiation

Figure 9.29: Non-Functional Model: Strong Mitigation

for the feature representing the non-functional performance strong mitigation model to be satisfied, one of two properties must be true: either the non-functional trigger (i.e., ¬PT) is not violated *or* the non-functional trigger is not violated *and* the mitigation is performed. That is, when strong mitigation is performed, the non-functional trigger should not indicate a violation. Instead, the mitigation (i.e., setting the regenerative force to some amount to attempt to ensure the battery is charged) prevents the violation *before* it takes place.

Once *Soter* translates the non-functional strong mitigation model in Figure 9.29b into the feature represented in Figure 9.30b, the feature is woven into the functional goal model by creating a new top-level goal and AND decomposing the new non-functional feature in Figure 9.30b with the functional goal model in Figure 7.1. The complete woven goal model is shown in Figure 9.31.

Soter analyzes the woven goal model in Figure 9.31 to detect any feature interactions or failures of the non-functional feature woven into the functional goal model. Since it is still possible to use the discharge brakes due to driving, the non-functional performance property is able to fail due to an interaction when the non-functional property is violated, requiring a mitigation. However, the mitigation (specifically `M.2`) conflicts with goal `C.10` in the functional goal, just as goal `C.10` conflicted in the weak mitigation. However, this time it is incorrect to give precedence to the mitigation over the functional braking system. Simply



(a) Template

(b) Performance Instantiation

Figure 9.30: Non-Functional Feature: Strong Mitigation

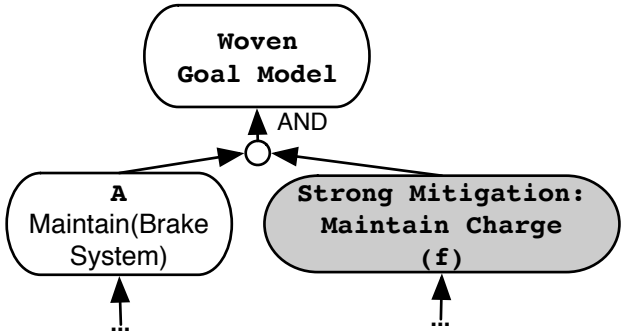


Figure 9.31: Goal Model Woven With Non-Functional Battery Performance Strong Mitigation

ensuring the regenerative brakes are on at all times will help to recharge the battery, but will ultimately also reduce both the battery and gasoline energy stores.

Strong mitigation to ensure that the battery remains sufficiently charged exists, including those that would use the gas engine to charge the battery of a hybrid vehicle. However, using the braking system alone, battery recharging may only be accomplished via braking when the vehicle is moving. Since the vehicle may not brake often enough to ensure the

battery is charged, the best course of action is to mitigate the low battery state when it occurs (i.e., weak mitigation).

Summary

We have shown that *Soter* can detect performance interactions with functional requirements symbolically with respect to each of the three non-functional model types (property, weak mitigation, and strong mitigation). Given the interactions and potential updates to the functional goal model, weak performance mitigation is the appropriate model to use since there is no model that can reasonably prevent the safety violation from ever occurring within the scope of the braking system.

9.4.4 Safety & Performance Case Study

This subsection gives the results of applying *Soter* to aspect-oriented safety models and performance models for the braking system goal model in Figure 7.1. *Soter* is used to detect interactions between safety models, performance models, and the functional goal model using symbolic analysis, a combination of symbolic analysis and evolutionary computation (SA+EC), and finally via generated run-time detection code. The models used in this example application are defined by the system designer.

Interactions Detected via Symbolic Analysis

The braking system goal model in Figure 7.1 has been previously woven with three necessary non-functional models: the weak mitigations for maintaining charge and minimizing collision and the strong mitigation for maintaining a safe charge amount (Figures 9.18b, 9.5b, and 9.30b, respectively). *Soter* weaves these three non-functional models that have been translated to features into the braking system goal model in Figure 7.1 resulting in the woven goal model shown in Figure 9.32.

Applying *Soter* to detect interactions due to the inclusion of non-functional properties

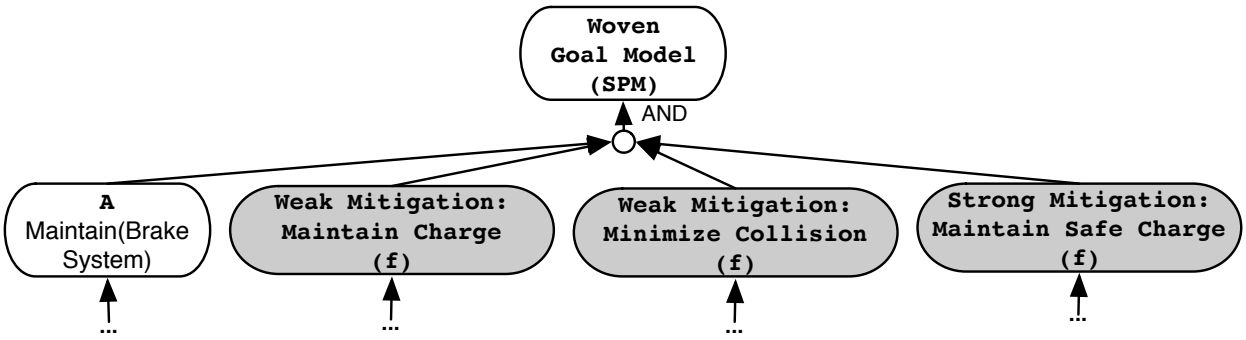


Figure 9.32: Woven Model: Functional With Both Safety and Performance Mitigation

in the overall goal model (i.e., the woven goal model shown in Figure 9.32) results in the following interactions caused by failures in:

- `Weak Mitigation: Maintain Charge`: As illustrated in the previous section describing this mitigation, it is still possible to drive the car without braking, which would result in a continued discharging of the battery. The non-functional performance property is able to fail due to an interaction when the non-functional property is violated, thus requiring a mitigation. However, the mitigation (specifically `M.2`) conflicts with goal `C.10` in the functional goal. However, a pre-condition requiring a non-violated non-functional property (i.e., $\neg$`PT`) can be added to the children of goal `C` to ensure the conflict does not occur by allowing the mitigation to provide all the commanded brake force when the performance property is violated. That is, goal `C.10` would never execute when `M.2` was required as a mitigation. The update ensures the battery is charged once it has become low enough to be below the desired charge level. While the mitigation cannot prevent further discharging of the battery, it is unlikely that a strong mitigation will be found that guarantees the performance non-functional property will remain unviolated.

- `Weak Mitigation: Minimize Collision`: Since this weak mitigation sets both the standard brake force *and* the regenerative brake force, an interaction can exist whenever the braking system has a commanded brake force that is used to set either of the

standard or regenerative brake forces to a value outside of the weak safety mitigation (i.e., goals `B.1`, `C.5`, and `C.10`). To prevent this interaction, the functional goal model must be updated according to the changes specified in the previous section describing this mitigation and detailed in Figures 9.11a, 9.11b, and 9.11c for goals `B.1`, `C.5`, and `C.10`, respectively.

- `Strong Mitigation: Maintain Safe Charge`: As illustrated in the previous section describing this mitigation, it is still possible to use the regenerative brakes, which would result in a continued charging of the battery, the non-functional safety property is able to fail due to an interaction when the non-functional property is violated, requiring a mitigation. However, the mitigation (specifically `M.3`) conflicts with goal `C.10` in the functional goal, just as goal `C.10` conflicted in the weak mitigation. However, just as before, a pre-condition can be added to ensure that when `M.3` is mitigating, `C.10` is not setting the regenerative brake force. Goal `M.3` only sets the regenerative brake force when `M.1` is true, so adding ¬`M.1` as a pre-condition to the children of goal `C.4` ensures the conflict does not occur. That is, goal `C.10` would never execute when `M.2` was required as a mitigation. The update ensures the battery is no longer charged once it has become close to violating the non-functional safety property.

Using symbolic analysis available to *Soter*, each of the non-functional models that have been translated into features and included in the woven goal model (i.e., in Figure 9.32) can fail to be satisfied due to a feature interaction across functional and non-functional specifications. For example, Table 9.5 shows a scenario where the functional model is satisfied and yet every single one of the non-functional features is unsatisfied due to an interaction.

Interactions Detected via SA+EC

Soter-EC combines SA and EC by making use of *Phorcys*-EC to detect failures in all three non-functional features (i.e., features `Weak Mitigation: Maintain Charge`, `Weak`

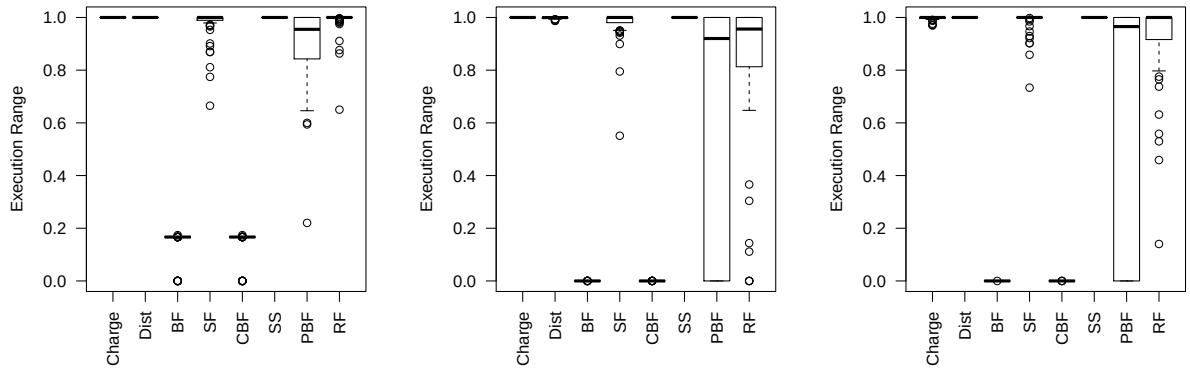
Table 9.5: Example Counterexample for Non-Functional FI Causes

Variable	Value
SF	0.07
CBF	0.35
PBF	0.0
RF	0.28
BF	0.35
SS	*true*
FD Sensor 1	0.0
FD Sensor 2	0.0
Battery Charge	*MAX*

`Mitigation: Minimize Collision`, and `Strong Mitigation: Maintain Safe Charge`) as features that can fail due to FIs, just as they were identified in the SA-only method when woven with the functional goal model individually. In all 50 executions of the genetic algorithm where each feature was analyzed for failure due to a FI, at least 97% of the population succeeded in finding a counterexample (i.e., FI).

Regardless of which feature, or features, failed due to the FI, at least 99.9% of the FIs identified included unique environmental variable values. That is, the FI counterexamples identified were almost always completely unique and every set of counterexamples (i.e., in a population of 200 identified by the SA+EC) included significant diversity as measured by the Manhattan distance of the environmental scenarios. This diversity is illustrated in Figures 9.33a, 9.33b and 9.33c that show the range (distance from minimum to maximum) each environmental variable took across each set of 200 individuals in the SA+EC analysis, where the box plot represents all 50 executions of the SA+EC. It should be noted, that it is expected that some of the ranges are limited since they represent the ranges of environmental variable values necessary for FIs to cause a feature to fail.

In all three cases, we can see that *Charge*, *Dist*, *SF*, *SS*, *PBF*, and *RF* have a large range of values while the remainder (i.e., *BF* and *CBF*) are limited due to the smaller range of those environmental variable values when FIs occur. Variables with a smaller range displayed are more likely to have an impact on the expression of the feature interaction,



(a) `Weak Mitigation: Maintain Charge` (b) `Weak Mitigation: Minimize Collision` (c) `Strong Mitigation: Maintain Safe Charge`

Figure 9.33: Environmental Variables for Non-Functional Failures due to FI

due to the feature interaction only occurring over a small portion of the variable's range. However, variables with a larger range displayed are less likely to have an impact on the expression of the feature interaction, since a feature interaction could be expressed over a large portion of the variable's range. System designers can use this information to assess the types of situations in which an interaction could occur, possibly directing additional testing to scenarios more likely to be adversely impacted by feature interactions.

Interactions Detected at Run Time

Soter-RT generates run-time code using *Thoosa*. Given the SA+EC analysis already applied (using *Soter*-EC), there are more than 9,750 individual and diverse examples for *each* of the three non-functional models (i.e., `Weak Mitigation: Maintain Charge`, `Weak Mitigation: Minimize Collision`, and `Strong Mitigation: Maintain Safe Charge`) that have been converted into features. That is, they are the *causes* of feature interactions, all of which are non-functional features converted from non-functional models. Each of these non-functional features has been previously identified using SA+EC to have multiple counterexamples.

The run-time detection code is verified by applying the set of individuals in the genetic population that identify known failures due to feature interactions, and ensuring that the en-

vironmental and system scenarios described by those individuals result in a detected feature interaction at simulated run time. In the case of the more than 29,000 individual examples identified by the SA+EC detection process, all (100%) are correctly detected and reported by the generated run-time detection code. That is, we use the existing counterexamples identified by SA+EC and initialize the run-time detection code with the values identified to simulate the scenario in order to verify the run-time detection code.

Functional Goal Model Updates and Re-analysis

Finally, the functional goal model originally introduced in Figure 7.1 must be updated to ensure the mitigations put in place by the non-functional features remain in effect. The updated functional goal model is shown in Figure 9.34 that is based on Figure 7.1 with the following changes made to the functional goal model to mitigate the feature interactions:

1. The functional goal model must be updated according to the changes specified in the previous section describing this mitigation and detailed in Figures 9.11a, 9.11b, and 9.11c for goals `B.1`, `C.5`, and `C.10`, respectively.

2. The functional goal model must be updated to constrain the satisfaction of goal `C.4` such that it is only specified if *Battery Charge* $\geqslant 0.50$ by adding such a pre-condition (i.e., `C.New2`) as one of goal `C.4`'s decomposed goals.

3. The functional goal model must be updated to constrain the satisfaction of goal `C` such that it is only specified if *Battery Charge* < 1.0 by adding such a pre-condition (i.e., `C.New1`) as one of goal `C`'s decomposed goals.

Applying *Soter* to the functional goal model updated by these three changes and woven into Figure 9.32 results in no further interactions.

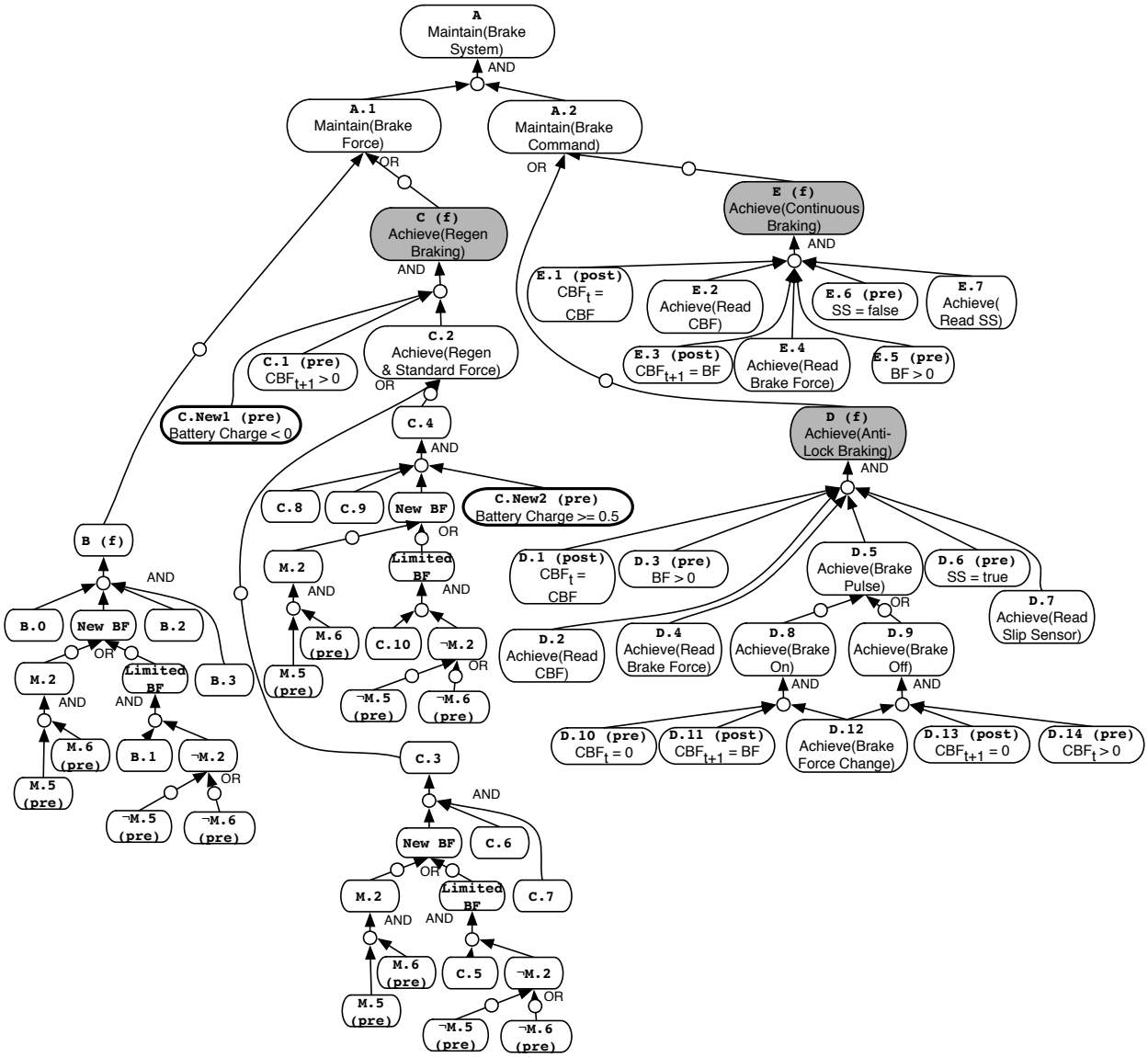


Figure 9.34: Final Braking System Goal Model

Summary

We have demonstrated that *Soter* can detect interactions between safety and performance models in an example of one of two weak mitigations (i.e., safety or performance) that failed due to a conflict with the other weak mitigation. Therefore, *Soter* can detect failures due to interactions when multiple aspect oriented non-functional goal models fail due to an interaction.

9.5 Related Work

Aspect-oriented requirements engineering allows for early separation of concerns, including cross-cutting safety and performance concerns [77]. However, aspect interactions may introduce failures early in the development process and require continued research [74]. This section broadly covers existing AORE work, especially as it relates to explicit modeling of safety and detecting interactions in aspect-oriented requirements.

Non-functional goals and properties are currently represented in GORE by directed edges that connect goals or properties. In KAOS goal modeling, non-functional goals may be included in the requirements decomposition or defined by obstacles and mitigations that are connected to goals or requirements [87]. For *i**, non-functional goals are connected to existing GORE elements based on the impact of the non-functional goal to the connected element [95]. The non-functional models defined using the *Soter* process are defined in relation to a non-functional concern, represented as an aspect-oriented non-functional goal model and is developed independently of the dominant decomposition.

Methods for modeling aspect-oriented requirements exist in multiple modeling paradigms, including UML [8] and GORE [91]. Existing requirements models can even be analyzed to identify candidate aspect-oriented requirements [79]. The *Soter* process is applied to GORE models specifically for analyzing cross-cutting non-functional concerns, including safety and performance.

Aspect interactions occur when separate and conflicting aspects are woven into the same model [71]. While aspect interaction detection in Aspect-Oriented Programming (AOP) can detect behavioral interactions [42, 90, 3], aspect interactions can also occur at the requirements level. *Soter* operates at the requirements level modeled by goals and detects aspect interactions in the corresponding goal models.

In aspect-oriented requirements engineering, aspect interactions can be detected in textual requirements [80], but more often, formalized models are used [71, 72, 73]. Some methods include conflict resolution for interactions [80, 19]. *Soter* differs from existing AORE aspect

interaction detection methods in that it identifies the failure caused by the interaction and is applied specifically to safety and performance non-functional concerns and directly utilizes the similarity between aspect and feature interactions by translating cross-cutting safety goals into formally analyzable features.

9.6 Conclusion

In this chapter, we have presented *Soter*, a design-time and run-time approach for detecting aspect interactions and non-functional model violations in aspect-oriented non-functional models. Unlike previous methods of detecting aspect interactions, *Soter* leverages the similarity between feature interaction and aspect interaction by using feature interaction detection techniques with aspect-oriented non-functional models translated into features.

We demonstrate *Soter* on an industry-based automotive braking system. We show that *Soter* is able to automatically detect safety and performance-based violations and aspect interactions in fractions of a second by translating aspect-oriented non-functional models into features and analyzing those features while maintaining traceability to the original models. Additionally, *Soter*-EC applies SA+EC to identify multiple diverse counterexamples that can be used by the system designer to more fully assess the impact of the interaction. Finally, *Soter*-RT generates run-time detection code and ensures the generated code can detect known counterexamples identified via *Soter*-EC.

Chapter 10

Contributions

This chapter summarizes our contributions with respect to the detection of incomplete requirements decompositions and feature interactions in the fields of software and requirements engineering. Additionally, we also propose future investigations to complement this body of work.

10.1 Summary of Contributions

Given the increasing complexity and intrinsic uncertainty in specifying cyber-physical systems, the design-time assurance of requirements specifications is necessary to reduce and, if possible, eliminate the proliferation of specification errors. Errors in the specification of cyber-physical systems may be introduced in several ways. First, specification errors may be introduced due to an idealized view of the environment where requirements do not account for environmental uncertainty. Second, specification errors may be introduced due to conflicting specifications due to system uncertainty. Due to the growth of system and environmental uncertainty in cyber-physical systems, researchers are actively exploring the analysis and detection of counterexamples in the specification of cyber-physical systems at design-time [4].

We have presented several methods to automatically identify counterexamples to completeness and consistency of requirements by identifying incomplete requirements and feature

interactions. First, we applied *Ares*, *Ares*-EC, and *Lykus* (i.e., a set of incomplete requirements detection tools) to detect incomplete requirements decompositions within a model-based hierarchical requirements decomposition. Second, we applied *Phorcys*, *Phorcys*-EC, and *Thoosa* (i.e., a set of feature interaction detection tools) to detect n-way feature interactions between the functional feature requirements specifications. Finally, we have applied *Soter* to non-functional requirements, including safety and performance non-functional requirements. We have demonstrated these approaches to requirements models related to the adaptive cruise control and braking systems, both of which are examples of onboard automotive system.

The methods and tools presented in this body of work fit within a general framework represented by the DFD in Figure 10.1. First, a problem is converted to propositional logic allowing for the creation of a method to *verify* solutions as correct or incorrect (i.e., detect if an existing solution for the propositional logic satisfies the propositional logic) and a method to *solve* problems by creating solutions (i.e., generating a new solution for the propositional logic that is known to satisfy the propositional logic). Since each of these problems can be used to represent any instance of 3-SAT, an NP-Complete problem [62], verifying solutions (i.e., checking if the solution satisfies the propositional logic) can be done quickly (i.e., in polynomial time) while the worst case for generating a new solution (i.e., generating a new solution that satisfies the propositional logic) is not in polynomial time. Where the faster solution verification method can be used to analyze either at run time or as part of a fitness function within a genetic algorithm, the slower solution generation method can be used to generate single results. These methods (i.e., solution verification and solution generation) can be employed to generate multiple solutions via a combination of evolutionary computation (EC) and symbolic analysis (SA), potentially multiple results as they are encountered individually at run time via verification of potential solutions (i.e., streaming results), or a single result via symbolic analysis of the solution generation method. Subsequently, we

identify the portions of the DFD in relation to the chapters presented in this body of work. The remainder of the contributions and future work are identified within this framework.

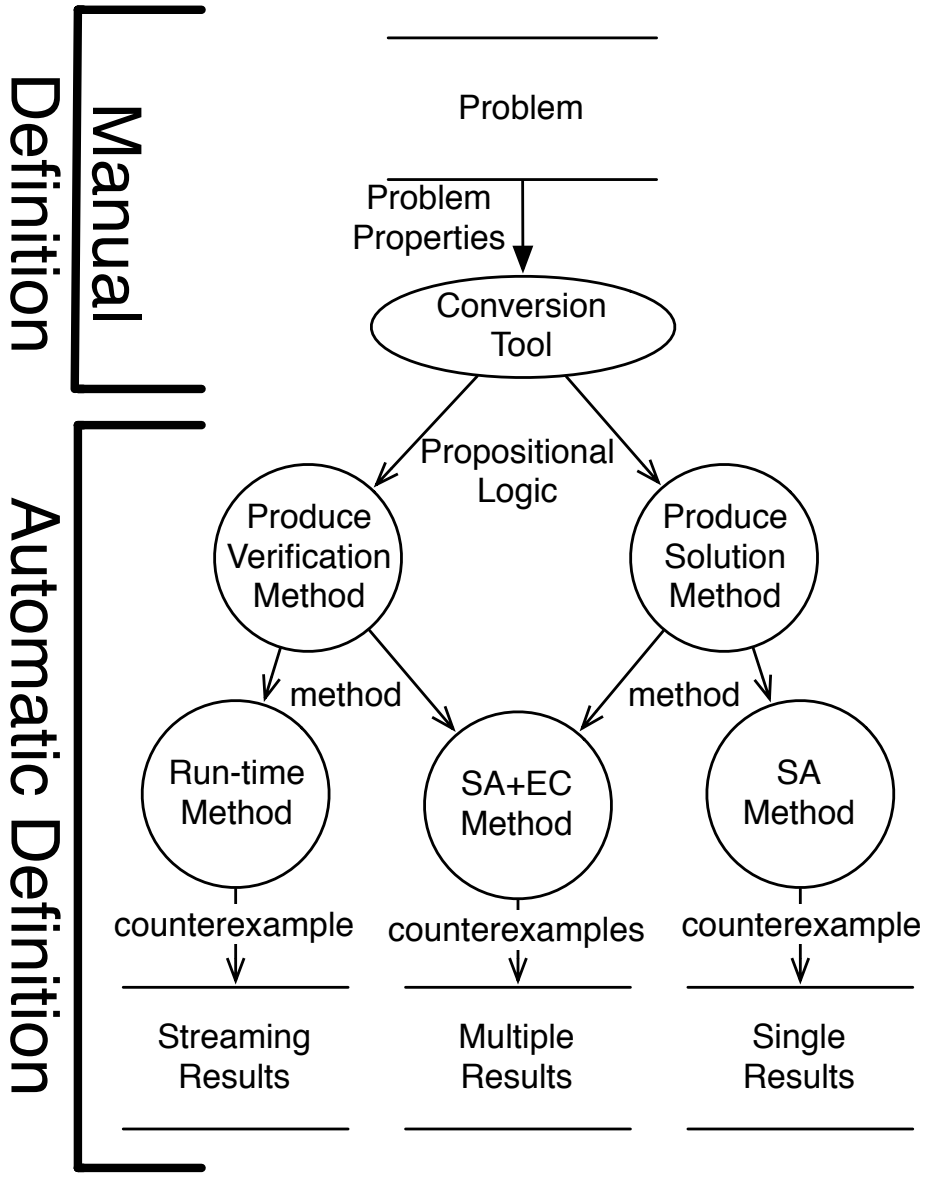


Figure 10.1: General Framework DFD

10.1.1 Requirements Incompleteness

We detect incomplete requirements decomposition in hierarchical requirements models. We illustrate our approach by analyzing a requirements model of an industry-based auto-

motive adaptive cruise control system. We describe the following techniques that we are developing to address this problem:

1. *Symbolic analysis* (*Ares*) to formally define requirements incompleteness and guarantee detection of requirements completeness counterexamples if they exist [37].

2. *Evolutionary computation and symbolic analysis* (*Ares*-EC) to detect a representative range of requirements completeness counterexamples [38].

3. *Run-time analysis* (*Lykus*) to detect incomplete requirements at run time [39].

Figure 10.2 depicts the applicability of the generalized framework (i.e., Figure 10.1) for the three pieces of work in the requirements incompleteness space.

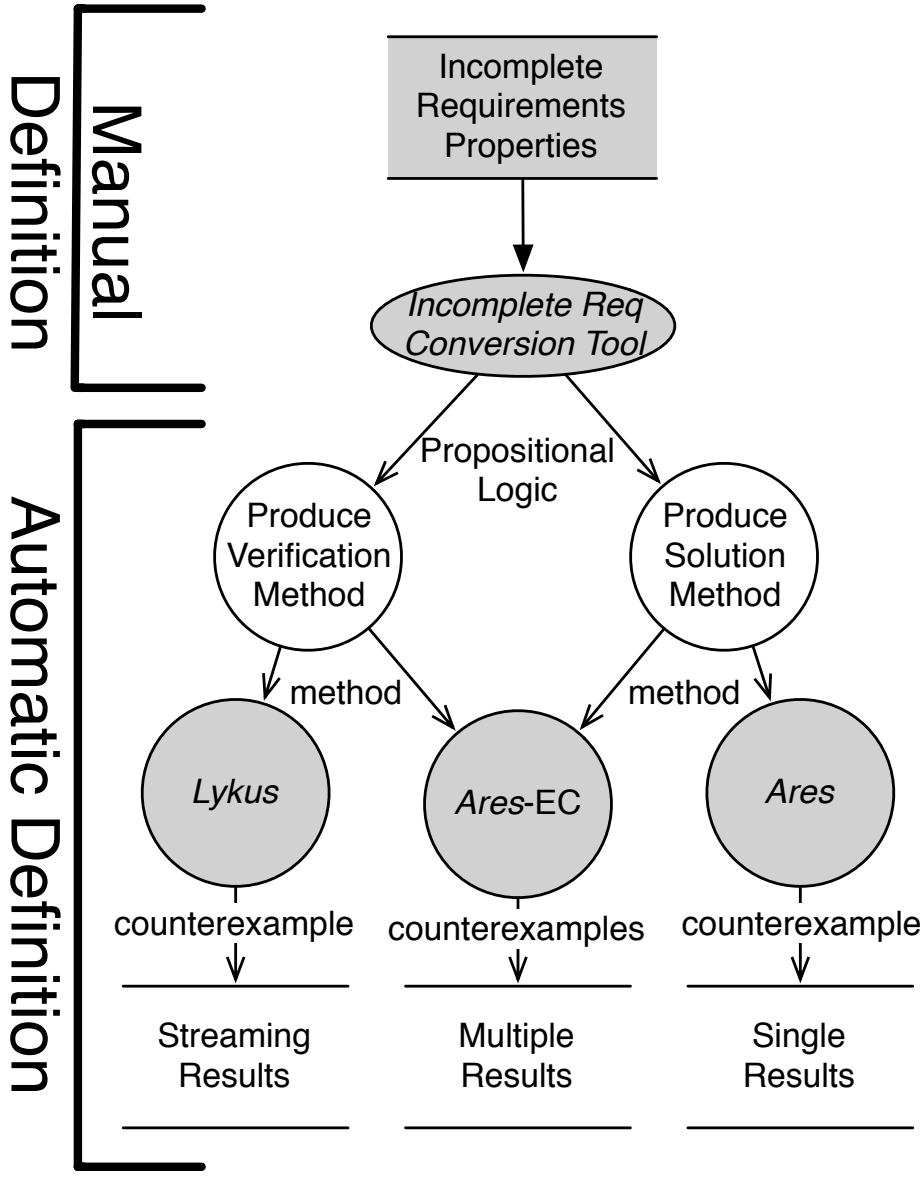


Figure 10.2: Requirements Completeness DFD

10.1.2 Feature Interactions

We detect unwanted n-way feature interactions and determining their causes at the requirements level. Unlike previous n-way feature interaction detection approaches that attempt to enumerate every set of interacting features, our approach analyzes each feature for its ability to cause an interaction with other features, thus reducing designer assessment effort to be linear with respect to the number of features. We illustrate our approach by

applying our approach to an industry-based automotive braking system comprising multiple subsystems. We describe the following techniques that we have developed for this dissertation:

1. *Phorcys: Symbolic analysis* to formally define feature interactions and guarantee detection of interactions if they exist [40].

2. *Phorcys-EC: Evolutionary computation and symbolic analysis* to detect a representative range of feature interactions [40].

3. *Thoosa: Run-time analysis* to detect feature interactions at run time.

Figure 10.3 demonstrates the applicability of the generalized framework (i.e., Figure 10.1) for the three pieces of work in the feature interaction detection space.

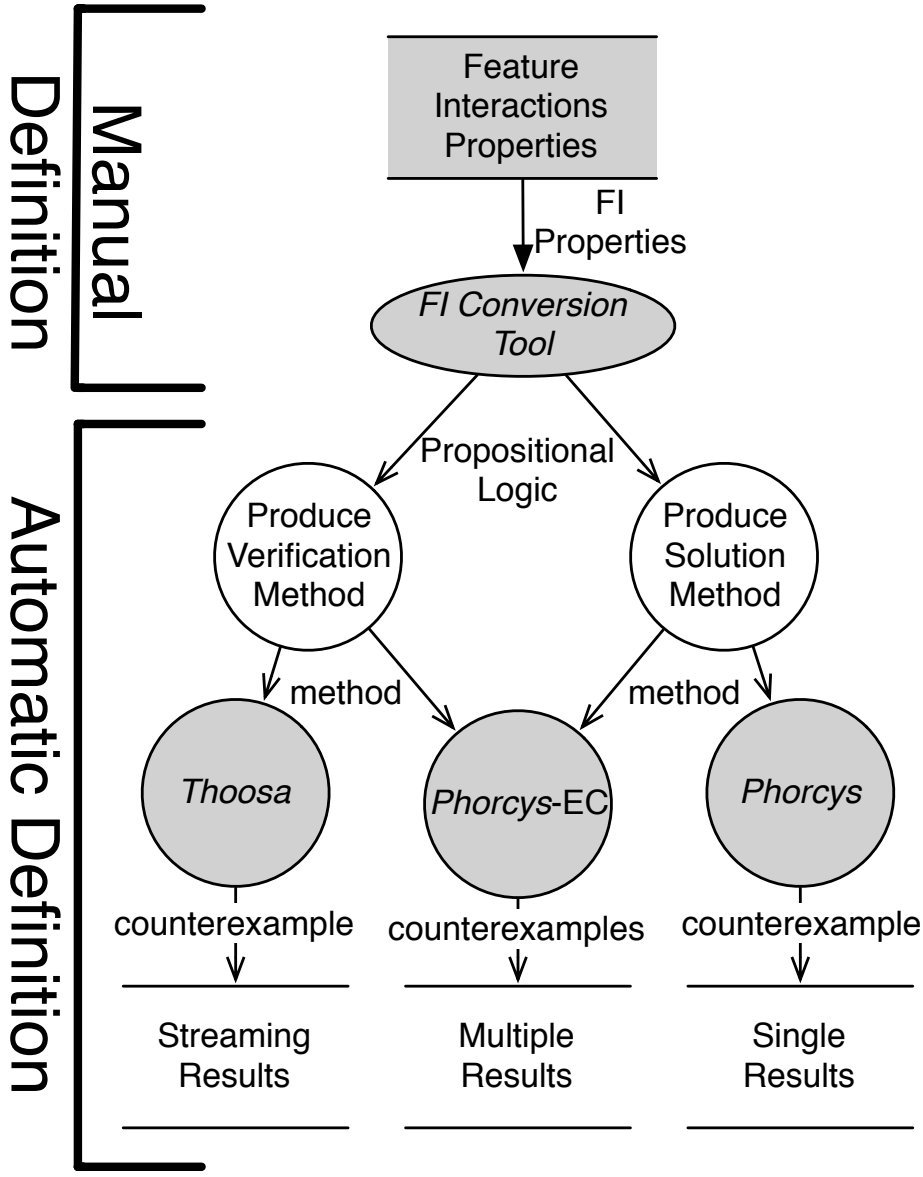


Figure 10.3: Feature Interaction DFD

10.1.3 Non-functional Interactions

We detect feature interactions that include safety and/or performance non-functional requirements and may include, but are not limited to include, functional requirements. We illustrate our approach by applying it to detect unwanted interactions in the safety models and requirements of an industry-based automotive braking system. We apply the detection method to the following types of non-functional requirements:

1. *Safety requirements*, and

2. *Performance requirements*.

Figure 10.4 depicts the applicability of the generalized framework (i.e., Figure 10.1) for the non-functional interaction detection.

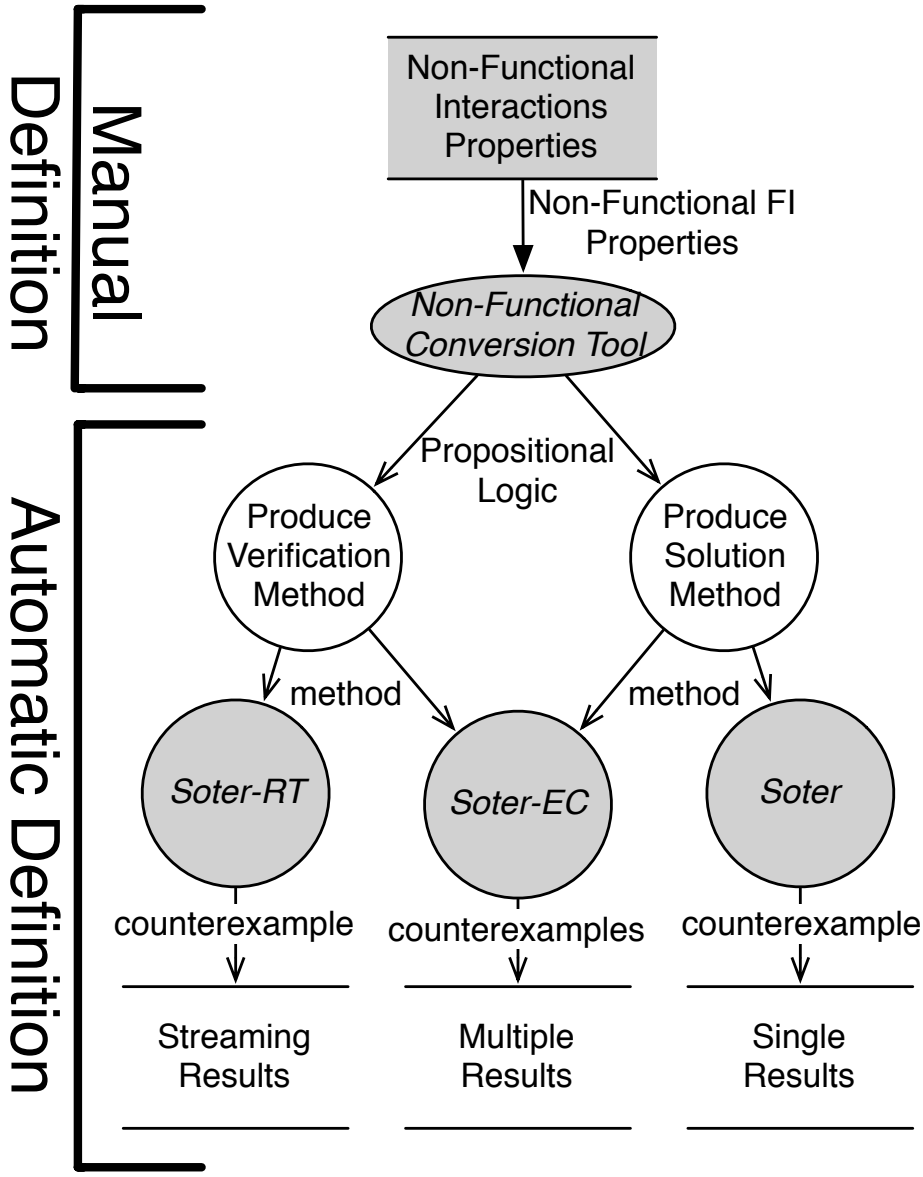


Figure 10.4: Non-Functional Interactions DFD

10.2 Future Investigations

This dissertation has presented a framework for applying symbolic analysis, evolutionary computation, and run-time detection to feature interactions, requirements completeness, and non-functional feature interactions. Given these results, we now describe future work that would complement what has been presented in this dissertation. Each of these is described in turn.

10.2.1 Requirements Incompleteness Caused by FI

While we detect both incomplete requirements decomposition *and* feature interactions within this work, we do not consider the combination of the two problems. However, there are almost certainly situations where a specification cannot act as it is intended due to a feature interaction. Those instances may introduce requirements incompleteness that are not inherent to the local structure of the decomposition, but the cross-cutting structure of conflicting specifications. Whereas our requirements completeness work analyzes each decomposition independently of the remaining specification, an interaction outside of that individual decomposition may introduce an incompleteness not previously detected. We envision analyzing the entire specification for incomplete decompositions *and* identifying if any of the unsatisfied requirements are due to a feature interaction.

10.2.2 Partial Feature Interactions

Currently the feature interaction logic that is generated by *Phorcys* applies logic-based on discrete satisfaction of the features in questions. The satisfaction of those features is derived from the satisfaction of the decomposed goals, requirements, and expectations that make up those features. However, *Phorcys* does not take into account partial satisfaction, or satisficement, of goal model elements that would correspond to a partially-satisfied feature. Naturally, an interaction between two (or more) features is a conflict that allows for a trade-

off between two (or more) features that are partially satisfied. We envision extending the *Phorcys* work to include feature interaction detection logic for partially satisfied features, allowing for multi-objective optimization of the trade-offs between features.

10.2.3 Partial Requirements Incompleteness

Currently the case study for requirements incompleteness includes no RELAXed or AutoRELAXED requirements [49], despite the use of satisficement to measure the proportional satisfaction of each requirement. As such, results are binary for requirements incompleteness, while RELAXed (or otherwise proportionally satisfied) requirements may result in partial requirements incompleteness due to their own intrinsic proportional satisfaction. We envision the application of the *Ares* set of tools to RELAXed goal models to automatically detect partial requirements incompleteness.

10.2.4 Automatic Mitigation Strategies

The methods detailed in this work focus on the *detection* of problematic states. Even the work on non-functional interactions where mitigations are included requires manual definition of the mitigations. However, significant existing work exists in dynamically adaptive software systems that could be leveraged to employ adaptation when such problematic states are encountered. The adaptive mitigation techniques can be particularly applicable to the *Thoosa*, *Lykus*, and *Soter* run-time detection methods detailed within this work.

10.2.5 Non-Functional Security Interactions

The non-functional interaction detection in this work has been shown to be applicable to both safety and performance non-functional goals. However, the detection method is generally applicable to all non-functional requirements. We envision the application of the existing framework to include case studies with non-functional security requirements.

10.2.6 Incomplete Test Sets

Similar to requirements completeness, which depends on the complete decomposition of a parent requirement by its children, individual requirements should be *tested* by a complete set of tests. That is, the functionality defined by the parent should be completely covered by the set of tests that exercise the system. If a specific instance of functionality exists that is not tested, the test set is *incomplete*. We envision using the *Ares* tool set to describe requirements and their tests as parent and child requirements where the child requirements are OR decomposed (i.e., disjunct) from one another.

APPENDICES

Appendix A

Incomplete Requirements Artifacts

This appendix presents the goal model specifications as well as the satisfiability-modulo theory (SMT) solver and code (C++) outputs from the *Ares* and *Lykus* tools for incomplete requirements detection. Incomplete requirements artifacts are included in this appendix since no complete example can be given, due to issues of space, within Chapters 3 and 5.

A.1 Goal Model Specifications

This section of the appendix includes the XML schema for representing goal models in XML, as well as XML representations of the adaptive cruise control system goal model used in Chapters 3, 4, and 5.

A.1.1 Goal Model Schema

Listing A.1: XSD Goal Model XML Schema for Incomplete Requirements

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3        elementFormDefault="qualified"
4        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5     <xs:element name="goalmodel">
6        <xs:complexType>
7           <xs:sequence>
```

```
 8          <xs:element ref="goal"/>
 9          <xs:element ref="environment"/>
10        </xs:sequence>
11      </xs:complexType>
12    </xs:element>
13    <xs:element name="environment">
14      <xs:complexType>
15        <xs:sequence>
16          <xs:element minOccurs="0" maxOccurs="unbounded" ref="env"/>
17        </xs:sequence>
18      </xs:complexType>
19    </xs:element>
20    <xs:element name="env">
21      <xs:complexType>
22        <xs:attribute name="name" use="required" type="xs:NCName"/>
23        <xs:attribute name="relationship" use="required" type="xs:
              string"/>
24      </xs:complexType>
25    </xs:element>
26    <xs:element name="goal">
27      <xs:complexType>
28        <xs:choice>
29          <xs:element maxOccurs="unbounded" ref="goal"/>
30          <xs:element ref="agent"/>
31        </xs:choice>
32        <xs:attribute name="contents" use="required" type="xs:string
              "/>
33        <xs:attribute name="name" use="required" type="xs:NCName"/>
34        <xs:attribute name="relationship" use="required" type="xs:
              string"/>
35        <xs:attribute name="type" use="required" type="xs:NCName"/>
36      </xs:complexType>
37    </xs:element>
38    <xs:element name="agent">
39      <xs:complexType>
40        <xs:attribute name="contents" use="required" type="xs:string
              "/>
41        <xs:attribute name="location" use="required" type="xs:NCName
              "/>
42        <xs:attribute name="name" use="required" type="xs:NCName"/>
43      </xs:complexType>
44    </xs:element>
45  </xs:schema>
```

A.1.2 Incomplete Requirements Goal Model for Adaptive Cruise Control System

Listing A.2: Incomplete ACC Goal Model

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- created with XMLSpear -->
3  <goalmodel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation='goalmodel.xsd'>
4
5      <goal name="A.1" contents="NA" relationship="true" type="OR">
6          <goal name="A.2" contents="NA" relationship="(and2 (EQUALS
               CruiseSwitchSensor 0.0) (EQUALS CruiseActiveSensor 1.0))
               " type="AND">
7              <goal name="A.6" contents="NA" relationship="(EQUALS
                   CruiseSwitchSensor 0.0)" type="PRE">
8              </goal>
9              <goal name="A.7" contents="NA" relationship="(EQUALS
                   CruiseActiveSensor 1.0)" type="REQ">
10             </goal>
11             <goal name="A.8" contents="NA" relationship="(EQUALS
                   CruiseActiveSwitch 0.0)" type="REQ">
12             </goal>
13         </goal>
14         <goal name="A.3" contents="NA" relationship="(and (=
               CruiseSwitchSensor false) (= CruiseActiveSensor false))"
               type="AND">
15             <goal name="A.9" contents="NA" relationship="(EQUALS
                   CruiseSwitchSensor 0.0)" type="PRE">
16             </goal>
17             <goal name="A.10" contents="NA" relationship="(EQUALS
                   CruiseActiveSensor 0.0)" type="PRE">
18             </goal>
19             <goal name="A.15a" contents="NA" relationship="(and2 (
                   EQUALS ThrottlePedalSensor ThrottleActuator) (EQUALS
                    BrakePedalSensor BrakeActuator))" type="AND">
20                 <goal name="A.16a" contents="NA" relationship="(
                       EQUALS ThrottlePedalSensor ThrottleActuator)"
                       type="AND">
21                     <goal name="A.18a" contents="NA" relationship="
                           true" type="REQ">
22                     </goal>
23                     <goal name="A.19a" contents="NA" relationship
                           ="(EQUALS ThrottlePedalSensor
                           ThrottleActuator)" type="PRE">
24                     </goal>
```

```
25                  </goal>
26                  <goal name="A.17a" contents="NA" relationship="(
                        EQUALS BrakePedalSensor BrakeActuator)" type="
                        AND">
27                      <goal name="A.20a" contents="NA" relationship="
                            true" type="REQ">
28                      </goal>
29                      <goal name="A.21a" contents="NA" relationship
                            ="(EQUALS BrakePedalSensor BrakeActuator)"
                            type="PRE">
30                      </goal>
31                  </goal>
32              </goal>
33          </goal>
34          <goal name="A.4" contents="NA" relationship="(and (=
                CruiseSwitchSensor true) (= CruiseActiveSensor false))"
                type="AND">
35              <goal name="A.11" contents="NA" relationship="(EQUALS
                    CruiseSwitchSensor 1.0)" type="PRE">
36              </goal>
37              <goal name="A.12" contents="NA" relationship="(EQUALS
                    CruiseActiveSensor 0.0)" type="PRE">
38              </goal>
39              <goal name="A.15b" contents="NA" relationship="(and2 (
                    EQUALS ThrottlePedalSensor ThrottleActuator) (EQUALS
                     BrakePedalSensor BrakeActuator))" type="AND">
40                  <goal name="A.16b" contents="NA" relationship="(
                        EQUALS ThrottlePedalSensor ThrottleActuator)"
                        type="AND">
41                      <goal name="A.18b" contents="NA" relationship="
                            true" type="REQ">
42                      </goal>
43                      <goal name="A.19b" contents="NA" relationship
                            ="(EQUALS ThrottlePedalSensor
                            ThrottleActuator)" type="PRE">
44                      </goal>
45                  </goal>
46                  <goal name="A.17b" contents="NA" relationship="(
                        EQUALS BrakePedalSensor BrakeActuator)" type="
                        AND">
47                      <goal name="A.20b" contents="NA" relationship="
                            true" type="REQ">
48                      </goal>
49                      <goal name="A.21b" contents="NA" relationship
                            ="(EQUALS BrakePedalSensor BrakeActuator)"
                            type="PRE">
50                      </goal>
```

```
51                  </goal>
52              </goal>
53          </goal>
54          <goal name="A.5" contents="NA" relationship="(and (=
                CruiseSwitchSensor false) (= CruiseActiveSensor true))"
                type="AND">
55              <goal name="A.13" contents="NA" relationship="(EQUALS
                    CruiseSwitchSensor 1.0)" type="PRE">
56              </goal>
57              <goal name="A.14" contents="NA" relationship="(EQUALS
                    CruiseActiveSensor 1.0)" type="PRE">
58              </goal>
59              <goal name="B.1" contents="NA" relationship="(or3 (
                    EQUALS Speed_t Speed_t_plus_1) (GREATER Speed_t
                    Speed_t_plus_1) (LESS Distance SafeDistance))" type
                    ="OR">
60                  <goal name="B.2" contents="NA" relationship="(or2 (
                        and2 (EQUALS Speed_t 0.0) (EQUALS Speed_t_plus_1
                         0.0)) (GREATER Speed_t Speed_t_plus_1))" type="
                        AND">
61                      <goal name="B.3" contents="NA" relationship="(
                            or2 (and2 (EQUALS Speed_t 0.0) (EQUALS
                            Speed_t_plus_1 0.0)) (GREATER Speed_t
                            Speed_t_plus_1))" type="OR">
62                          <goal name="B.4" contents="NA" relationship
                                ="(LESS ThrottleActuator
                                ThrottlePedalSensor)" type="AND">
63                              <goal name="B.5" contents="NA"
                                    relationship="(LESS ThrottleActuator
                                     ThrottlePedalSensor)" type="REQ">
64                              </goal>
65                              <goal name="B.6" contents="NA"
                                    relationship="true" type="REQ">
66                              </goal>
67                              <goal name="B.7" contents="NA"
                                    relationship="(GREATER
                                    ThrottlePedalSensor 0.0)" type="PRE
                                    ">
68                              </goal>
69                          </goal>
70                          <goal name="B.15" contents="NA"
                                relationship="(GREATER BrakeActuator
                                BrakePedalSensor)" type="AND">
71                              <goal name="B.16" contents="NA"
                                    relationship="(GREATER BrakeActuator
                                     BrakePedalSensor)" type="REQ">
72                              </goal>
```

```
73                          <goal name="B.17" contents="NA"
                                relationship="true" type="REQ">
74                          </goal>
75                          <goal name="B.18" contents="NA"
                                relationship="(EQUALS
                                ThrottlePedalSensor 0.0)" type="PRE
                                ">
76                          </goal>
77                          <goal name="B.19" contents="NA"
                                relationship="(LESS BrakePedalSensor
                                 1.0)" type="PRE">
78                          </goal>
79                      </goal>
80                  </goal>
81                  <goal name="B.8" contents="NA" relationship="(
                        or2 (GREATER Speed_t DesiredSpeed) (LESS
                        Distance SafeDistance))" type="OR">
82                      <goal name="B.9" contents="NA" relationship
                            ="(GREATER Speed_t DesiredSpeed)" type="
                            AND">
83                          <goal name="B.11" contents="NA"
                                relationship="(GREATER
                                WheelSpeedSensor DesiredSpeed)" type
                                ="PRE">
84                          </goal>
85                          <goal name="B.12" contents="NA"
                                relationship="(GREATER
                                GPSSpeedSensor DesiredSpeed)" type="
                                PRE">
86                          </goal>
87                      </goal>
88                      <goal name="B.10" contents="NA"
                            relationship="(LESS Distance
                            SafeDistance)" type="AND">
89                          <goal name="B.13" contents="NA"
                                relationship="(LESS DistanceSensor1
                                SafeDistance)" type="PRE">
90                          </goal>
91                          <goal name="B.14" contents="NA"
                                relationship="(LESS DistanceSensor2
                                SafeDistance)" type="PRE">
92                          </goal>
93                      </goal>
94                  </goal>
95              </goal>
96              <goal name="C.1" contents="NA" relationship="(LESS
                    Speed_t Speed_t_plus_1)" type="AND">
```

```
97                  <goal name="C.2" contents="NA" relationship="(
                        and2 (LESS Speed_t DesiredSpeed) (GREATER
                        Distance SafeDistance))" type="AND">
98                      <goal name="C.4" contents="NA" relationship
                            ="(LESS Speed_t DesiredSpeed)" type="AND
                            ">
99                          <goal name="C.6" contents="NA"
                                relationship="(LESS WheelSpeedSensor
                                 DesiredSpeed)" type="PRE">
100                         </goal>
101                         <goal name="C.7" contents="NA"
                                relationship="(LESS GPSSpeedSensor
                                DesiredSpeed)" type="PRE">
102                         </goal>
103                     </goal>
104                     <goal name="C.9" contents="NA" relationship
                            ="(GREATER Distance SafeDistance)" type
                            ="AND">
105                         <goal name="C.10" contents="NA"
                                relationship="(GREATER
                                DistanceSensor1 SafeDistance)" type
                                ="PRE">
106                         </goal>
107                         <goal name="C.11" contents="NA"
                                relationship="(GREATER
                                DistanceSensor2 SafeDistance)" type
                                ="PRE">
108                         </goal>
109                     </goal>
110                 </goal>
111                 <goal name="C.3" contents="NA" relationship="(
                        GREATER ThrottleActuator ThrottlePedalSensor
                        )" type="AND">
112                     <goal name="C.5" contents="NA" relationship
                            ="(GREATER ThrottleActuator
                            ThrottlePedalSensor)" type="REQ">
113                     </goal>
114                     <goal name="C.8" contents="NA" relationship
                            ="(LESS ThrottlePedalSensor 1.0)" type="
                            PRE">
115                     </goal>
116                     <goal name="C.13" contents="NA"
                            relationship="true" type="PRE">
117                     </goal>
118                 </goal>
119                 <goal name="C.12" contents="NA" relationship="(
                        EQUALS BrakeActuator 0.0)" type="REQ">
```

```
120                     </goal>
121                 </goal>
122                 <goal name="D.1" contents="NA" relationship="(
                        EQUALS Speed_t Speed_t_plus_1)" type="AND">
123                     <goal name="D.2" contents="NA" relationship="(
                            and2 (EQUALS Speed_t DesiredSpeed) (GREATER
                            Distance SafeDistance))" type="AND">
124                         <goal name="D.4" contents="NA" relationship
                                ="(EQUALS Speed_t DesiredSpeed)" type="
                                AND">
125                             <goal name="D.6" contents="NA"
                                    relationship="(EQUALS
                                    WheelSpeedSensor DesiredSpeed)" type
                                    ="PRE">
126                             </goal>
127                             <goal name="D.7" contents="NA"
                                    relationship="(EQUALS GPSSpeedSensor
                                     DesiredSpeed)" type="PRE">
128                             </goal>
129                         </goal>
130                         <goal name="D.8" contents="NA" relationship
                                ="(GREATER Distance SafeDistance)" type
                                ="AND">
131                             <goal name="D.9" contents="NA"
                                    relationship="(GREATER
                                    DistanceSensor1 SafeDistance)" type
                                    ="PRE">
132                             </goal>
133                             <goal name="D.10" contents="NA"
                                    relationship="(GREATER
                                    DistanceSensor2 SafeDistance)" type
                                    ="PRE">
134                             </goal>
135                         </goal>
136                     </goal>
137                     <goal name="D.5" contents="NA" relationship="
                            true" type="REQ">
138                     </goal>
139                     <goal name="D.3" contents="NA" relationship="(
                        EQUALS ThrottleActuator ThrottlePedalSensor)
                        " type="REQ">
140                     </goal>
141                 </goal>
142             </goal>
143         </goal>
144     </goal>
145
```

```
146     <environment>
147         <env name="speed_t_a" relationship="(or (= Speed_t
              WheelSpeedSensor) (= Speed_t GPSSpeedSensor))"/>
148         <env name="speed_t_b" relationship="(= Speed_t (max2 0.0 (-
              ThrottlePedalSensor BrakePedalSensor)))"/>
149         <env name="speed_t_plus_one" relationship="(= Speed_t_plus_1
               (max2 0.0 (- ThrottleActuator BrakeActuator)))"/>
150         <env name="distance" relationship="(or (= Distance
              DistanceSensor1) (= Distance DistanceSensor2))"/>
151         </environment>
152
153 </goalmodel>
```

A.1.3 Complete Requirements Goal Model for Adaptive Cruise Control System

Listing A.3: Complete ACC Goal Model

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <!-- created with XMLSpear -->
 3 <goalmodel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation='goalmodel.xsd'>
 4
 5     <goal name="A.1" contents="NA" relationship="true" type="OR">
 6         <goal name="A.2" contents="NA" relationship="(and2 (EQUALS
              CruiseSwitchSensor 0.0) (EQUALS CruiseActiveSensor 1.0))
              " type="AND">
 7             <goal name="A.6" contents="NA" relationship="(EQUALS
                  CruiseSwitchSensor 0.0)" type="PRE">
 8             </goal>
 9             <goal name="A.7" contents="NA" relationship="(EQUALS
                  CruiseActiveSensor 1.0)" type="REQ">
10             </goal>
11             <goal name="A.8" contents="NA" relationship="(EQUALS
                  CruiseActiveSwitch 0.0)" type="REQ">
12             </goal>
13         </goal>
14         <goal name="A.3" contents="NA" relationship="(and (=
              CruiseSwitchSensor false) (= CruiseActiveSensor false))"
               type="AND">
15             <goal name="A.9" contents="NA" relationship="(EQUALS
                  CruiseSwitchSensor 0.0)" type="PRE">
16             </goal>
17             <goal name="A.10" contents="NA" relationship="(EQUALS
                  CruiseActiveSensor 0.0)" type="PRE">
```

```
18              </goal>
19              <goal name="A.15a" contents="NA" relationship="(and2 (
                    EQUALS ThrottlePedalSensor ThrottleActuator) (EQUALS
                     BrakePedalSensor BrakeActuator))" type="AND">
20                  <goal name="A.16a" contents="NA" relationship="(
                        EQUALS ThrottlePedalSensor ThrottleActuator)"
                        type="AND">
21                      <goal name="A.18a" contents="NA" relationship="
                            true" type="REQ">
22                      </goal>
23                      <goal name="A.19a" contents="NA" relationship
                            ="(EQUALS ThrottlePedalSensor
                            ThrottleActuator)" type="PRE">
24                      </goal>
25                  </goal>
26                  <goal name="A.17a" contents="NA" relationship="(
                        EQUALS BrakePedalSensor BrakeActuator)" type="
                        AND">
27                      <goal name="A.20a" contents="NA" relationship="
                            true" type="REQ">
28                      </goal>
29                      <goal name="A.21a" contents="NA" relationship
                            ="(EQUALS BrakePedalSensor BrakeActuator)"
                            type="PRE">
30                      </goal>
31                  </goal>
32              </goal>
33          </goal>
34          <goal name="A.4" contents="NA" relationship="(and (=
                CruiseSwitchSensor true) (= CruiseActiveSensor false))"
                type="AND">
35              <goal name="A.11" contents="NA" relationship="(EQUALS
                    CruiseSwitchSensor 1.0)" type="PRE">
36              </goal>
37              <goal name="A.12" contents="NA" relationship="(EQUALS
                    CruiseActiveSensor 0.0)" type="PRE">
38              </goal>
39              <goal name="A.15b" contents="NA" relationship="(and2 (
                    EQUALS ThrottlePedalSensor ThrottleActuator) (EQUALS
                     BrakePedalSensor BrakeActuator))" type="AND">
40                  <goal name="A.16b" contents="NA" relationship="(
                        EQUALS ThrottlePedalSensor ThrottleActuator)"
                        type="AND">
41                      <goal name="A.18b" contents="NA" relationship="
                            true" type="REQ">
42                      </goal>
43                      <goal name="A.19b" contents="NA" relationship
```

```
                    ="(EQUALS ThrottlePedalSensor
                    ThrottleActuator)" type="PRE">
44                  </goal>
45                </goal>
46                <goal name="A.17b" contents="NA" relationship="(
                    EQUALS BrakePedalSensor BrakeActuator)" type="
                    AND">
47                  <goal name="A.20b" contents="NA" relationship="
                      true" type="REQ">
48                  </goal>
49                  <goal name="A.21b" contents="NA" relationship
                      ="(EQUALS BrakePedalSensor BrakeActuator)"
                      type="PRE">
50                  </goal>
51                </goal>
52              </goal>
53            </goal>
54            <goal name="A.5" contents="NA" relationship="(and (=
                CruiseSwitchSensor false) (= CruiseActiveSensor true))"
                type="AND">
55              <goal name="A.13" contents="NA" relationship="(EQUALS
                  CruiseSwitchSensor 1.0)" type="PRE">
56              </goal>
57              <goal name="A.14" contents="NA" relationship="(EQUALS
                  CruiseActiveSensor 1.0)" type="PRE">
58              </goal>
59              <goal name="B.1" contents="NA" relationship="(or3 (
                  EQUALS Speed_t Speed_t_plus_1) (GREATER Speed_t
                  Speed_t_plus_1) (LESS Distance SafeDistance))" type
                  ="OR">
60                <goal name="B.2" contents="NA" relationship="(or2 (
                    and2 (EQUALS Speed_t 0.0) (EQUALS Speed_t_plus_1
                     0.0)) (GREATER Speed_t Speed_t_plus_1))" type="
                    AND">
61                  <goal name="B.3" contents="NA" relationship="(
                      or2 (and2 (EQUALS Speed_t 0.0) (EQUALS
                      Speed_t_plus_1 0.0)) (GREATER Speed_t
                      Speed_t_plus_1))" type="AND">
62                    <goal name="B.4" contents="NA" relationship
                        ="(LESS ThrottleActuator
                        ThrottlePedalSensor)" type="AND">
63                      <goal name="B.5" contents="NA"
                          relationship="(LESS ThrottleActuator
                           ThrottlePedalSensor)" type="REQ">
64                      </goal>
65                      <goal name="B.6" contents="NA"
                          relationship="true" type="REQ">
```

```
66                          </goal>
67                          <goal name="B.7" contents="NA"
                                relationship="(GREATER
                                ThrottlePedalSensor 0.0)" type="PRE
                                ">
68                          </goal>
69                      </goal>
70                      <goal name="B.15" contents="NA"
                            relationship="(GREATER BrakeActuator
                            BrakePedalSensor)" type="AND">
71                          <goal name="B.16" contents="NA"
                                relationship="(GREATER BrakeActuator
                                 BrakePedalSensor)" type="REQ">
72                          </goal>
73                          <goal name="B.17" contents="NA"
                                relationship="true" type="REQ">
74                          </goal>
75                          <goal name="B.18" contents="NA"
                                relationship="(EQUALS
                                ThrottlePedalSensor 0.0)" type="PRE
                                ">
76                          </goal>
77                          <goal name="B.19" contents="NA"
                                relationship="(LESS BrakePedalSensor
                                 1.0)" type="PRE">
78                          </goal>
79                      </goal>
80                  </goal>
81                  <goal name="B.8" contents="NA" relationship="(
                        or2 (GREATER Speed_t DesiredSpeed) (LESS
                        Distance SafeDistance))" type="OR">
82                      <goal name="B.9" contents="NA" relationship
                            ="(GREATER Speed_t DesiredSpeed)" type="
                            AND">
83                          <goal name="B.11" contents="NA"
                                relationship="(GREATER
                                WheelSpeedSensor DesiredSpeed)" type
                                ="PRE">
84                          </goal>
85                          <goal name="B.12" contents="NA"
                                relationship="(GREATER
                                GPSSpeedSensor DesiredSpeed)" type="
                                PRE">
86                          </goal>
87                      </goal>
88                      <goal name="B.10" contents="NA"
```

```
                relationship="(LESS Distance
                SafeDistance)" type="AND">
89                <goal name="B.13" contents="NA"
                      relationship="(LESS DistanceSensor1
                      SafeDistance)" type="PRE">
90                </goal>
91                <goal name="B.14" contents="NA"
                      relationship="(LESS DistanceSensor2
                      SafeDistance)" type="PRE">
92                </goal>
93            </goal>
94        </goal>
95    </goal>
96    <goal name="C.1" contents="NA" relationship="(LESS
          Speed_t Speed_t_plus_1)" type="AND">
97        <goal name="C.2" contents="NA" relationship="(
              and2 (LESS Speed_t DesiredSpeed) (GREATER
              Distance SafeDistance))" type="AND">
98            <goal name="C.4" contents="NA" relationship
                  ="(LESS Speed_t DesiredSpeed)" type="AND
                  ">
99                <goal name="C.6" contents="NA"
                      relationship="(LESS WheelSpeedSensor
                       DesiredSpeed)" type="PRE">
100               </goal>
101               <goal name="C.7" contents="NA"
                      relationship="(LESS GPSSpeedSensor
                      DesiredSpeed)" type="PRE">
102               </goal>
103           </goal>
104           <goal name="C.9" contents="NA" relationship
                  ="(GREATER Distance SafeDistance)" type
                  ="AND">
105               <goal name="C.10" contents="NA"
                      relationship="(GREATER
                      DistanceSensor1 SafeDistance)" type
                      ="PRE">
106               </goal>
107               <goal name="C.11" contents="NA"
                      relationship="(GREATER
                      DistanceSensor2 SafeDistance)" type
                      ="PRE">
108               </goal>
109           </goal>
110       </goal>
111       <goal name="C.3" contents="NA" relationship="(
```

```
                    GREATER ThrottleActuator ThrottlePedalSensor
                    )" type="AND">
112                  <goal name="C.5" contents="NA" relationship
                        ="(GREATER ThrottleActuator
                        ThrottlePedalSensor)" type="REQ">
113                  </goal>
114                  <goal name="C.8" contents="NA" relationship
                        ="(LESS ThrottlePedalSensor 1.0)" type="
                        PRE">
115                  </goal>
116                  <goal name="C.13" contents="NA"
                        relationship="true" type="PRE">
117                  </goal>
118               </goal>
119               <goal name="C.12" contents="NA" relationship="(
                     EQUALS BrakeActuator 0.0)" type="REQ">
120               </goal>
121            </goal>
122            <goal name="D.1" contents="NA" relationship="(
                  EQUALS Speed_t Speed_t_plus_1)" type="AND">
123               <goal name="D.2" contents="NA" relationship="(
                     and2 (EQUALS Speed_t DesiredSpeed) (GREATER
                     Distance SafeDistance))" type="AND">
124                  <goal name="D.4" contents="NA" relationship
                        ="(EQUALS Speed_t DesiredSpeed)" type="
                        AND">
125                     <goal name="D.6" contents="NA"
                           relationship="(EQUALS
                           WheelSpeedSensor DesiredSpeed)" type
                           ="PRE">
126                     </goal>
127                     <goal name="D.7" contents="NA"
                           relationship="(EQUALS GPSSpeedSensor
                            DesiredSpeed)" type="PRE">
128                     </goal>
129                  </goal>
130                  <goal name="D.8" contents="NA" relationship
                        ="(GREATER Distance SafeDistance)" type
                        ="AND">
131                     <goal name="D.9" contents="NA"
                           relationship="(GREATER
                           DistanceSensor1 SafeDistance)" type
                           ="PRE">
132                     </goal>
133                     <goal name="D.10" contents="NA"
                           relationship="(GREATER
```

```
                                        DistanceSensor2 SafeDistance)" type
                                        ="PRE">
134                                 </goal>
135                              </goal>
136                          </goal>
137                          <goal name="D.5" contents="NA" relationship="
                                true" type="REQ">
138                          </goal>
139                          <goal name="D.3" contents="NA" relationship="(
                                EQUALS ThrottleActuator ThrottlePedalSensor)
                                " type="REQ">
140                          </goal>
141                          <goal name="D.New1" contents="NA" relationship
                                ="(EQUALS BrakeActuator BrakePedalSensor)"
                                type="PRE">
142                          </goal>
143                          <goal name="D.New2" contents="NA" relationship
                                ="true" type="REQ">
144                          </goal>
145                      </goal>
146                  </goal>
147              </goal>
148          </goal>
149
150          <environment>
151              <env name="speed_t_a" relationship="(or (= Speed_t
                    WheelSpeedSensor) (= Speed_t GPSSpeedSensor))"/>
152              <env name="speed_t_b" relationship="(= Speed_t (max2 0.0 (-
                     ThrottlePedalSensor BrakePedalSensor)))"/>
153              <env name="speed_t_plus_one" relationship="(=
                    Speed_t_plus_1 (max2 0.0 (- ThrottleActuator
                    BrakeActuator)))"/>
154              <env name="distance" relationship="(or (= Distance
                    DistanceSensor1) (= Distance DistanceSensor2))"/>
155              </environment>
156
157  </goalmodel>
```

A.2 Design-Time Detection Constraints (SMT)

This section of the appendix includes the SMT constraint output for detecting incomplete requirements decompositions in Chapter 3. Constraint sets are listed for an incomplete requirements model and a complete requirements model.

A.2.1 Detection Constraints for Incomplete Adaptive Cruise Control System

Listing A.4: Incomplete Adaptive Cruise Control System Constraints

```
(define-fun min2 ((x1 Real) (x2 Real)) Real
        (/ (- (+ x1 x2) (abs (- x1 x2))) 2))

(define-fun and2 ((x1 Real) (x2 Real)) Real
        (min2 x1 x2))

(define-fun and3 ((x1 Real) (x2 Real) (x3 Real)) Real
        (and2 (and2 x1 x2) x3))

(define-fun and4 ((x1 Real) (x2 Real) (x3 Real) (x4 Real)) Real
        (and2 (and3 x1 x2 x3) x4))

(define-fun and5 ((x1 Real) (x2 Real) (x3 Real) (x4 Real) (x5 Real)
    ) Real
        (and2 (and4 x1 x2 x3 x4) x5))

(define-fun max2 ((x1 Real) (x2 Real)) Real
   (/ (+ x1 x2 (abs (- x1 x2))) 2))

(define-fun or2 ((x1 Real) (x2 Real)) Real
        (max2 x1 x2))

(define-fun or3 ((x1 Real) (x2 Real) (x3 Real)) Real
        (or2 x1 (or2 x2 x3)))

(define-fun or4 ((x1 Real) (x2 Real) (x3 Real) (x4 Real)) Real
        (or2 x1 (or3 x2 x3 x4)))

(define-fun not1 ((x1 Real)) Real
        (- 1.0 x1))

(define-fun EQUALS ((a Real) (b Real)) Real
        (ite (= a b) 1.0 0.0))

(define-fun GREATER ((a Real) (b Real)) Real
        (ite (> a b) 1.0 0.0))

(define-fun LESS ((a Real) (b Real)) Real
        (ite (< a b) 1.0 0.0))

(declare-const Speed_t Real)
```

```
41 (assert (and (<= Speed_t 1.0) (>= Speed_t 0.0)))
42
43 (declare-const Speed_t_plus_1 Real)
44 (assert (and (<= Speed_t_plus_1 1.0) (>= Speed_t_plus_1 0.0)))
45
46 (declare-const Distance Real)
47 (assert (and (<= Distance 1.0) (>= Distance 0.0)))
48
49 (declare-const SafeDistance Real)
50 (assert (and (<= SafeDistance 1.0) (>= SafeDistance 0.0)))
51
52 (declare-const DesiredSpeed Real)
53 (assert (and (<= DesiredSpeed 1.0) (>= DesiredSpeed 0.0)))
54
55 (declare-const ThrottleActuator Real)
56 (assert (and (<= ThrottleActuator 1.0) (>= ThrottleActuator 0.0)))
57
58 (declare-const ThrottlePedalSensor Real)
59 (assert (and (<= ThrottlePedalSensor 1.0) (>= ThrottlePedalSensor
      0.0)))
60
61 (declare-const WheelSpeedSensor Real)
62 (assert (and (<= WheelSpeedSensor 1.0) (>= WheelSpeedSensor 0.0)))
63
64 (declare-const GPSSpeedSensor Real)
65 (assert (and (<= GPSSpeedSensor 1.0) (>= GPSSpeedSensor 0.0)))
66
67 (declare-const DistanceSensor1 Real)
68 (assert (and (<= DistanceSensor1 1.0) (>= DistanceSensor1 0.0)))
69
70 (declare-const DistanceSensor2 Real)
71 (assert (and (<= DistanceSensor2 1.0) (>= DistanceSensor2 0.0)))
72
73 (declare-const BrakeActuator Real)
74 (assert (and (<= BrakeActuator 1.0) (>= BrakeActuator 0.0)))
75
76 (declare-const BrakePedalSensor Real)
77 (assert (and (<= BrakePedalSensor 1.0) (>= BrakePedalSensor 0.0)))
78
79 (declare-const CruiseSwitchSensor Real)
80 (assert (and (<= CruiseSwitchSensor 1.0) (>= CruiseSwitchSensor
      0.0)))
81
82 (declare-const CruiseActiveSensor Real)
83 (assert (and (<= CruiseActiveSensor 1.0) (>= CruiseActiveSensor
      0.0)))
84
```

```
85 (declare-const CruiseActiveSwitch Real)
86 (assert (and (<= CruiseActiveSwitch 1.0) (>= CruiseActiveSwitch
      0.0)))
87
88 (assert (or (= Speed_t WheelSpeedSensor) (= Speed_t GPSSpeedSensor)
      ))
89 (assert (or (= Distance DistanceSensor1) (= Distance
      DistanceSensor2)))
90 (assert (= Speed_t_plus_1 (max2 0.0 (- ThrottleActuator
      BrakeActuator))))
91 (assert (= Speed_t (max2 0.0 (- ThrottlePedalSensor
      BrakePedalSensor))))
92
93 (define-fun A_1 () Real
94         1.0)
95
96 (define-fun A_2 () Real
97         (and2 (EQUALS CruiseSwitchSensor 0.0) (EQUALS
            CruiseActiveSensor 1.0)))
98
99 (define-fun A_3 () Real
100        (and2 (EQUALS CruiseSwitchSensor 0.0) (EQUALS
            CruiseActiveSensor 0.0)))
101
102 (define-fun A_4 () Real
103        (and2 (EQUALS CruiseSwitchSensor 1.0) (EQUALS
            CruiseActiveSensor 0.0)))
104
105 (define-fun A_5 () Real
106        (and2 (EQUALS CruiseSwitchSensor 1.0) (EQUALS
            CruiseActiveSensor 1.0)))
107
108 (define-fun A_6 () Real
109        (EQUALS CruiseSwitchSensor 0.0))
110
111 (define-fun A_7 () Real
112        (EQUALS CruiseActiveSensor 1.0))
113
114 (define-fun A_8 () Real
115        (EQUALS CruiseActiveSwitch 0.0))
116
117 (define-fun A_9 () Real
118        (EQUALS CruiseSwitchSensor 0.0))
119
120 (define-fun A_10 () Real
121        (EQUALS CruiseActiveSensor 0.0))
122
```

```
123 (define-fun A_11 () Real
124         (EQUALS CruiseSwitchSensor 1.0))
125
126 (define-fun A_12 () Real
127         (EQUALS CruiseActiveSensor 0.0))
128
129 (define-fun A_13 () Real
130         (EQUALS CruiseSwitchSensor 1.0))
131
132 (define-fun A_14 () Real
133         (EQUALS CruiseActiveSensor 1.0))
134
135 (define-fun A_15 () Real
136         (and2 (EQUALS ThrottlePedalSensor ThrottleActuator) (EQUALS
                BrakePedalSensor BrakeActuator)))
137
138 (define-fun A_16 () Real
139         (EQUALS ThrottlePedalSensor ThrottleActuator))
140
141 (define-fun A_17 () Real
142         (EQUALS BrakePedalSensor BrakeActuator))
143
144 (define-fun A_18 () Real
145         1.0)
146
147 (define-fun A_19 () Real
148         (EQUALS ThrottlePedalSensor ThrottleActuator))
149
150 (define-fun A_20 () Real
151         1.0)
152
153 (define-fun A_21 () Real
154         (EQUALS BrakePedalSensor BrakeActuator))
155
156 (define-fun B_1 () Real
157         (or3 (EQUALS Speed_t Speed_t_plus_1) (GREATER Speed_t
               Speed_t_plus_1) (LESS Distance SafeDistance)))
158
159 (define-fun B_2 () Real
160         (or2 (and2 (EQUALS Speed_t 0.0) (EQUALS Speed_t_plus_1 0.0)
               ) (GREATER Speed_t Speed_t_plus_1)))
161
162 (define-fun B_3 () Real
163         (or2 (and2 (EQUALS Speed_t 0.0) (EQUALS Speed_t_plus_1 0.0)
               ) (GREATER Speed_t Speed_t_plus_1)))
164
165 (define-fun B_4 () Real
```

```
166         (LESS ThrottleActuator ThrottlePedalSensor))
167
168 (define-fun B_5 () Real
169         (LESS ThrottleActuator ThrottlePedalSensor))
170
171 (define-fun B_6 () Real
172         1.0)
173
174 (define-fun B_7 () Real
175         (GREATER ThrottlePedalSensor 0.0))
176
177 (define-fun B_8 () Real
178         (or2 (GREATER Speed_t DesiredSpeed) (LESS Distance
              SafeDistance)))
179
180 (define-fun B_9 () Real
181         (GREATER Speed_t DesiredSpeed))
182
183 (define-fun B_10 () Real
184         (LESS Distance SafeDistance))
185
186 (define-fun B_11 () Real
187         (GREATER WheelSpeedSensor DesiredSpeed))
188
189 (define-fun B_12 () Real
190         (GREATER GPSSpeedSensor DesiredSpeed))
191
192 (define-fun B_13 () Real
193         (LESS DistanceSensor1 SafeDistance))
194
195 (define-fun B_14 () Real
196         (LESS DistanceSensor2 SafeDistance))
197
198 (define-fun B_15 () Real
199         (GREATER BrakeActuator BrakePedalSensor))
200
201 (define-fun B_16 () Real
202         (GREATER BrakeActuator BrakePedalSensor))
203
204 (define-fun B_17 () Real
205         1.0)
206
207 (define-fun B_18 () Real
208         (EQUALS ThrottlePedalSensor 0.0))
209
210 (define-fun B_19 () Real
211         (LESS BrakePedalSensor 1.0))
```

```
212
213 (define-fun C_1 () Real
214         (LESS Speed_t Speed_t_plus_1))
215
216 (define-fun C_2 () Real
217         (and2 (LESS Speed_t DesiredSpeed) (GREATER Distance
              SafeDistance)))
218
219 (define-fun C_3 () Real
220         (GREATER ThrottleActuator ThrottlePedalSensor))
221
222 (define-fun C_4 () Real
223         (LESS Speed_t DesiredSpeed))
224
225 (define-fun C_5 () Real
226         (GREATER ThrottleActuator ThrottlePedalSensor))
227
228 (define-fun C_6 () Real
229         (LESS WheelSpeedSensor DesiredSpeed))
230
231 (define-fun C_7 () Real
232         (LESS GPSSpeedSensor DesiredSpeed))
233
234 (define-fun C_8 () Real
235         (LESS ThrottlePedalSensor 1.0))
236
237 (define-fun C_9 () Real
238         (GREATER Distance SafeDistance))
239
240 (define-fun C_10 () Real
241         (GREATER DistanceSensor1 SafeDistance))
242
243 (define-fun C_11 () Real
244         (GREATER DistanceSensor2 SafeDistance))
245
246 (define-fun C_12 () Real
247         (EQUALS BrakeActuator 0.0))
248
249 (define-fun C_13 () Real
250         1.0)
251
252 (define-fun D_1 () Real
253         (EQUALS Speed_t Speed_t_plus_1))
254
255 (define-fun D_2 () Real
256         (and2 (EQUALS Speed_t DesiredSpeed) (GREATER Distance
              SafeDistance)))
```

```
257
258 (define-fun D_3 () Real
259         (EQUALS ThrottleActuator ThrottlePedalSensor))
260
261 (define-fun D_4 () Real
262         (EQUALS Speed_t DesiredSpeed))
263
264 (define-fun D_5 () Real
265         1.0)
266
267 (define-fun D_6 () Real
268         (EQUALS WheelSpeedSensor DesiredSpeed))
269
270 (define-fun D_7 () Real
271         (EQUALS GPSSpeedSensor DesiredSpeed))
272
273 (define-fun D_8 () Real
274         (GREATER Distance SafeDistance))
275
276 (define-fun D_9 () Real
277         (GREATER DistanceSensor1 SafeDistance))
278
279 (define-fun D_10 () Real
280         (GREATER DistanceSensor2 SafeDistance))
281
282 (define-fun A_1_decomp () Real
283         (or4 A_2 A_3 A_4 A_5))
284
285 (define-fun A_2_decomp () Real
286         (and3 A_6 A_7 A_8))
287
288 (define-fun A_3_decomp () Real
289         (and3 A_9 A_10 A_15))
290
291 (define-fun A_4_decomp () Real
292         (and3 A_15 A_11 A_12))
293
294 (define-fun A_5_decomp () Real
295         (and3 A_13 A_14 B_1))
296
297 (define-fun A_15_decomp () Real
298         (and2 A_16 A_17))
299
300 (define-fun A_16_decomp () Real
301         (and2 A_18 A_19))
302
303 (define-fun A_17_decomp () Real
```

```
304         (and2 A_20 A_21))
305
306 (define-fun B_1_decomp () Real
307         (and3 C_1 B_2 D_1))
308
309 (define-fun B_2_decomp () Real
310         (and2 B_8 B_3))
311
312 (define-fun B_3_decomp () Real
313         (or2 B_4 B_15))
314
315 (define-fun B_4_decomp () Real
316         (and3 B_5 B_6 B_7))
317
318 (define-fun B_8_decomp () Real
319         (or2 B_9 B_10))
320
321 (define-fun B_9_decomp () Real
322         (and2 B_11 B_12))
323
324 (define-fun B_10_decomp () Real
325         (and2 B_13 B_14))
326
327 (define-fun B_15_decomp () Real
328         (and4 B_16 B_17 B_18 B_19))
329
330 (define-fun C_1_decomp () Real
331         (and3 C_2 C_3 C_12))
332
333 (define-fun C_2_decomp () Real
334         (and2 C_4 C_9))
335
336 (define-fun C_3_decomp () Real
337         (and3 C_5 C_13 C_8))
338
339 (define-fun C_4_decomp () Real
340         (and2 C_6 C_7))
341
342 (define-fun C_9_decomp () Real
343         (and2 C_10 C_11))
344
345 (define-fun D_1_decomp () Real
346         (and3 D_2 D_5 D_3))
347
348 (define-fun D_2_decomp () Real
349         (and2 D_4 D_8))
350
```

```
(define-fun D_4_decomp () Real
        (and2 D_6 D_7))

(define-fun D_8_decomp () Real
        (and2 D_9 D_10))

(push)
(maximize (- A_1_decomp A_1))
(check-sat)
(pop)

(push)
(maximize (- A_2_decomp A_2))
(check-sat)
(pop)

(push)
(maximize (- A_3_decomp A_3))
(check-sat)
(pop)

(push)
(maximize (- A_4_decomp A_4))
(check-sat)
(pop)

(push)
(maximize (- A_5_decomp A_5))
(check-sat)
(pop)

(push)
(maximize (- A_15_decomp A_15))
(check-sat)
(pop)

(push)
(maximize (- A_16_decomp A_16))
(check-sat)
(pop)

(push)
(maximize (- A_17_decomp A_17))
(check-sat)
(pop)

(push)
```

```
(maximize (- B_1_decomp B_1))
(check-sat)
(pop)

(push)
(maximize (- B_2_decomp B_2))
(check-sat)
(pop)

(push)
(maximize (- B_3_decomp B_3))
(check-sat)
(pop)

(push)
(maximize (- B_4_decomp B_4))
(check-sat)
(pop)

(push)
(maximize (- B_8_decomp B_8))
(check-sat)
(pop)

(push)
(maximize (- B_9_decomp B_9))
(check-sat)
(pop)

(push)
(maximize (- B_10_decomp B_10))
(check-sat)
(pop)

(push)
(maximize (- B_15_decomp B_15))
(check-sat)
(pop)

(push)
(maximize (- C_1_decomp C_1))
(check-sat)
(pop)

(push)
(maximize (- C_2_decomp C_2))
(check-sat)
```

```
445 (pop)
446
447 (push)
448 (maximize (- C_3_decomp C_3))
449 (check-sat)
450 (pop)
451
452 (push)
453 (maximize (- C_4_decomp C_4))
454 (check-sat)
455 (pop)
456
457 (push)
458 (maximize (- C_9_decomp C_9))
459 (check-sat)
460 (pop)
461
462 (push)
463 (maximize (- D_1_decomp D_1))
464 (check-sat)
465 (pop)
466
467 (push)
468 (maximize (- D_2_decomp D_2))
469 (check-sat)
470 (pop)
471
472 (push)
473 (maximize (- D_4_decomp D_4))
474 (check-sat)
475 (pop)
476
477 (push)
478 (maximize (- D_8_decomp D_8))
479 (check-sat)
480 (pop)
```

A.2.2 Detection Constraints for Complete Adaptive Cruise Control System

Listing A.5: Complete Adaptive Cruise Control System Constraints

```
1 (define-fun min2 ((x1 Real) (x2 Real)) Real
2         (/ (- (+ x1 x2) (abs (- x1 x2))) 2))
3
```

```
4  (define-fun and2 ((x1 Real) (x2 Real)) Real
5          (min2 x1 x2))
6
7  (define-fun and3 ((x1 Real) (x2 Real) (x3 Real)) Real
8          (and2 (and2 x1 x2) x3))
9
10 (define-fun and4 ((x1 Real) (x2 Real) (x3 Real) (x4 Real)) Real
11         (and2 (and3 x1 x2 x3) x4))
12
13 (define-fun and5 ((x1 Real) (x2 Real) (x3 Real) (x4 Real) (x5 Real)
       ) Real
14         (and2 (and4 x1 x2 x3 x4) x5))
15
16 (define-fun max2 ((x1 Real) (x2 Real)) Real
17    (/ (+ x1 x2 (abs (- x1 x2))) 2))
18
19 (define-fun or2 ((x1 Real) (x2 Real)) Real
20         (max2 x1 x2))
21
22 (define-fun or3 ((x1 Real) (x2 Real) (x3 Real)) Real
23         (or2 x1 (or2 x2 x3)))
24
25 (define-fun or4 ((x1 Real) (x2 Real) (x3 Real) (x4 Real)) Real
26         (or2 x1 (or3 x2 x3 x4)))
27
28 (define-fun not1 ((x1 Real)) Real
29         (- 1.0 x1))
30
31 (define-fun EQUALS ((a Real) (b Real)) Real
32         (ite (= a b) 1.0 0.0))
33
34 (define-fun GREATER ((a Real) (b Real)) Real
35         (ite (> a b) 1.0 0.0))
36
37 (define-fun LESS ((a Real) (b Real)) Real
38         (ite (< a b) 1.0 0.0))
39
40 (declare-const Speed_t Real)
41 (assert (and (<= Speed_t 1.0) (>= Speed_t 0.0)))
42
43 (declare-const Speed_t_plus_1 Real)
44 (assert (and (<= Speed_t_plus_1 1.0) (>= Speed_t_plus_1 0.0)))
45
46 (declare-const Distance Real)
47 (assert (and (<= Distance 1.0) (>= Distance 0.0)))
48
49 (declare-const SafeDistance Real)
```

```
50 (assert (and (<= SafeDistance 1.0) (>= SafeDistance 0.0)))
51
52 (declare-const DesiredSpeed Real)
53 (assert (and (<= DesiredSpeed 1.0) (>= DesiredSpeed 0.0)))
54
55 (declare-const ThrottleActuator Real)
56 (assert (and (<= ThrottleActuator 1.0) (>= ThrottleActuator 0.0)))
57
58 (declare-const ThrottlePedalSensor Real)
59 (assert (and (<= ThrottlePedalSensor 1.0) (>= ThrottlePedalSensor
      0.0)))
60
61 (declare-const WheelSpeedSensor Real)
62 (assert (and (<= WheelSpeedSensor 1.0) (>= WheelSpeedSensor 0.0)))
63
64 (declare-const GPSSpeedSensor Real)
65 (assert (and (<= GPSSpeedSensor 1.0) (>= GPSSpeedSensor 0.0)))
66
67 (declare-const DistanceSensor1 Real)
68 (assert (and (<= DistanceSensor1 1.0) (>= DistanceSensor1 0.0)))
69
70 (declare-const DistanceSensor2 Real)
71 (assert (and (<= DistanceSensor2 1.0) (>= DistanceSensor2 0.0)))
72
73 (declare-const BrakeActuator Real)
74 (assert (and (<= BrakeActuator 1.0) (>= BrakeActuator 0.0)))
75
76 (declare-const BrakePedalSensor Real)
77 (assert (and (<= BrakePedalSensor 1.0) (>= BrakePedalSensor 0.0)))
78
79 (declare-const CruiseSwitchSensor Real)
80 (assert (and (<= CruiseSwitchSensor 1.0) (>= CruiseSwitchSensor
      0.0)))
81
82 (declare-const CruiseActiveSensor Real)
83 (assert (and (<= CruiseActiveSensor 1.0) (>= CruiseActiveSensor
      0.0)))
84
85 (declare-const CruiseActiveSwitch Real)
86 (assert (and (<= CruiseActiveSwitch 1.0) (>= CruiseActiveSwitch
      0.0)))
87
88 (assert (or (= Speed_t WheelSpeedSensor) (= Speed_t GPSSpeedSensor)
      ))
89 (assert (or (= Distance DistanceSensor1) (= Distance
      DistanceSensor2)))
```

```
90 (assert (= Speed_t_plus_1 (max2 0.0 (- ThrottleActuator
      BrakeActuator))))
91 (assert (= Speed_t (max2 0.0 (- ThrottlePedalSensor
      BrakePedalSensor))))
92
93 (define-fun A_1 () Real
94         1.0)
95
96 (define-fun A_2 () Real
97         (and2 (EQUALS CruiseSwitchSensor 0.0) (EQUALS
            CruiseActiveSensor 1.0)))
98
99 (define-fun A_3 () Real
100        (and2 (EQUALS CruiseSwitchSensor 0.0) (EQUALS
            CruiseActiveSensor 0.0)))
101
102 (define-fun A_4 () Real
103        (and2 (EQUALS CruiseSwitchSensor 1.0) (EQUALS
            CruiseActiveSensor 0.0)))
104
105 (define-fun A_5 () Real
106        (and2 (EQUALS CruiseSwitchSensor 1.0) (EQUALS
            CruiseActiveSensor 1.0)))
107
108 (define-fun A_6 () Real
109        (EQUALS CruiseSwitchSensor 0.0))
110
111 (define-fun A_7 () Real
112        (EQUALS CruiseActiveSensor 1.0))
113
114 (define-fun A_8 () Real
115        (EQUALS CruiseActiveSwitch 0.0))
116
117 (define-fun A_9 () Real
118        (EQUALS CruiseSwitchSensor 0.0))
119
120 (define-fun A_10 () Real
121        (EQUALS CruiseActiveSensor 0.0))
122
123 (define-fun A_11 () Real
124        (EQUALS CruiseSwitchSensor 1.0))
125
126 (define-fun A_12 () Real
127        (EQUALS CruiseActiveSensor 0.0))
128
129 (define-fun A_13 () Real
130        (EQUALS CruiseSwitchSensor 1.0))
```

```
131
132 (define-fun A_14 () Real
133         (EQUALS CruiseActiveSensor 1.0))
134
135 (define-fun A_15 () Real
136         (and2 (EQUALS ThrottlePedalSensor ThrottleActuator) (EQUALS
                BrakePedalSensor BrakeActuator)))
137
138 (define-fun A_16 () Real
139         (EQUALS ThrottlePedalSensor ThrottleActuator))
140
141 (define-fun A_17 () Real
142         (EQUALS BrakePedalSensor BrakeActuator))
143
144 (define-fun A_18 () Real
145         1.0)
146
147 (define-fun A_19 () Real
148         (EQUALS ThrottlePedalSensor ThrottleActuator))
149
150 (define-fun A_20 () Real
151         1.0)
152
153 (define-fun A_21 () Real
154         (EQUALS BrakePedalSensor BrakeActuator))
155
156 (define-fun B_1 () Real
157         (or3 (EQUALS Speed_t Speed_t_plus_1) (GREATER Speed_t
               Speed_t_plus_1) (LESS Distance SafeDistance)))
158
159 (define-fun B_2 () Real
160         (or2 (and2 (EQUALS Speed_t 0.0) (EQUALS Speed_t_plus_1 0.0)
               ) (GREATER Speed_t Speed_t_plus_1)))
161
162 (define-fun B_3 () Real
163         (or2 (and2 (EQUALS Speed_t 0.0) (EQUALS Speed_t_plus_1 0.0)
               ) (GREATER Speed_t Speed_t_plus_1)))
164
165 (define-fun B_4 () Real
166         (LESS ThrottleActuator ThrottlePedalSensor))
167
168 (define-fun B_5 () Real
169         (LESS ThrottleActuator ThrottlePedalSensor))
170
171 (define-fun B_6 () Real
172         1.0)
173
```

```
174 (define-fun B_7 () Real
175         (GREATER ThrottlePedalSensor 0.0))
176
177 (define-fun B_8 () Real
178         (or2 (GREATER Speed_t DesiredSpeed) (LESS Distance
               SafeDistance)))
179
180 (define-fun B_9 () Real
181         (GREATER Speed_t DesiredSpeed))
182
183 (define-fun B_10 () Real
184         (LESS Distance SafeDistance))
185
186 (define-fun B_11 () Real
187         (GREATER WheelSpeedSensor DesiredSpeed))
188
189 (define-fun B_12 () Real
190         (GREATER GPSSpeedSensor DesiredSpeed))
191
192 (define-fun B_13 () Real
193         (LESS DistanceSensor1 SafeDistance))
194
195 (define-fun B_14 () Real
196         (LESS DistanceSensor2 SafeDistance))
197
198 (define-fun B_15 () Real
199         (GREATER BrakeActuator BrakePedalSensor))
200
201 (define-fun B_16 () Real
202         (GREATER BrakeActuator BrakePedalSensor))
203
204 (define-fun B_17 () Real
205         1.0)
206
207 (define-fun B_18 () Real
208         (EQUALS ThrottlePedalSensor 0.0))
209
210 (define-fun B_19 () Real
211         (LESS BrakePedalSensor 1.0))
212
213 (define-fun C_1 () Real
214         (LESS Speed_t Speed_t_plus_1))
215
216 (define-fun C_2 () Real
217         (and2 (LESS Speed_t DesiredSpeed) (GREATER Distance
               SafeDistance)))
218
```

```
219 (define-fun C_3 () Real
220         (GREATER ThrottleActuator ThrottlePedalSensor))
221
222 (define-fun C_4 () Real
223         (LESS Speed_t DesiredSpeed))
224
225 (define-fun C_5 () Real
226         (GREATER ThrottleActuator ThrottlePedalSensor))
227
228 (define-fun C_6 () Real
229         (LESS WheelSpeedSensor DesiredSpeed))
230
231 (define-fun C_7 () Real
232         (LESS GPSSpeedSensor DesiredSpeed))
233
234 (define-fun C_8 () Real
235         (LESS ThrottlePedalSensor 1.0))
236
237 (define-fun C_9 () Real
238         (GREATER Distance SafeDistance))
239
240 (define-fun C_10 () Real
241         (GREATER DistanceSensor1 SafeDistance))
242
243 (define-fun C_11 () Real
244         (GREATER DistanceSensor2 SafeDistance))
245
246 (define-fun C_12 () Real
247         (EQUALS BrakeActuator 0.0))
248
249 (define-fun C_13 () Real
250         1.0)
251
252 (define-fun D_New1 () Real
253         (EQUALS BrakeActuator BrakePedalSensor))
254
255 (define-fun D_New2 () Real
256         1.0)
257
258 (define-fun D_1 () Real
259         (EQUALS Speed_t Speed_t_plus_1))
260
261 (define-fun D_2 () Real
262         (and2 (EQUALS Speed_t DesiredSpeed) (GREATER Distance
                SafeDistance)))
263
264 (define-fun D_3 () Real
```

```
265         (EQUALS ThrottleActuator ThrottlePedalSensor))
266
267 (define-fun D_4 () Real
268         (EQUALS Speed_t DesiredSpeed))
269
270 (define-fun D_5 () Real
271         1.0)
272
273 (define-fun D_6 () Real
274         (EQUALS WheelSpeedSensor DesiredSpeed))
275
276 (define-fun D_7 () Real
277         (EQUALS GPSSpeedSensor DesiredSpeed))
278
279 (define-fun D_8 () Real
280         (GREATER Distance SafeDistance))
281
282 (define-fun D_9 () Real
283         (GREATER DistanceSensor1 SafeDistance))
284
285 (define-fun D_10 () Real
286         (GREATER DistanceSensor2 SafeDistance))
287
288 (define-fun A_1_decomp () Real
289         (or4 A_2 A_3 A_4 A_5))
290
291 (define-fun A_2_decomp () Real
292         (and3 A_6 A_7 A_8))
293
294 (define-fun A_3_decomp () Real
295         (and3 A_9 A_10 A_15))
296
297 (define-fun A_4_decomp () Real
298         (and3 A_15 A_11 A_12))
299
300 (define-fun A_5_decomp () Real
301         (and3 A_13 A_14 B_1))
302
303 (define-fun A_15_decomp () Real
304         (and2 A_16 A_17))
305
306 (define-fun A_16_decomp () Real
307         (and2 A_18 A_19))
308
309 (define-fun A_17_decomp () Real
310         (and2 A_20 A_21))
311
```

```
312 (define-fun B_1_decomp () Real
313         (and3 C_1 B_2 D_1))
314
315 (define-fun B_2_decomp () Real
316         (and2 B_8 B_3))
317
318 (define-fun B_3_decomp () Real
319         (and2 B_4 B_15))
320
321 (define-fun B_4_decomp () Real
322         (and3 B_5 B_6 B_7))
323
324 (define-fun B_8_decomp () Real
325         (or2 B_9 B_10))
326
327 (define-fun B_9_decomp () Real
328         (and2 B_11 B_12))
329
330 (define-fun B_10_decomp () Real
331         (and2 B_13 B_14))
332
333 (define-fun B_15_decomp () Real
334         (and4 B_16 B_17 B_18 B_19))
335
336 (define-fun C_1_decomp () Real
337         (and3 C_2 C_3 C_12))
338
339 (define-fun C_2_decomp () Real
340         (and2 C_4 C_9))
341
342 (define-fun C_3_decomp () Real
343         (and3 C_5 C_13 C_8))
344
345 (define-fun C_4_decomp () Real
346         (and2 C_6 C_7))
347
348 (define-fun C_9_decomp () Real
349         (and2 C_10 C_11))
350
351 (define-fun D_1_decomp () Real
352         (and5 D_2 D_5 D_3 D_New1 D_New2))
353
354 (define-fun D_2_decomp () Real
355         (and2 D_4 D_8))
356
357 (define-fun D_4_decomp () Real
358         (and2 D_6 D_7))
```

```
359
360 (define-fun D_8_decomp () Real
361         (and2 D_9 D_10))
362
363 (push)
364 (maximize (- A_1_decomp A_1))
365 (check-sat)
366 (pop)
367
368 (push)
369 (maximize (- A_2_decomp A_2))
370 (check-sat)
371 (pop)
372
373 (push)
374 (maximize (- A_3_decomp A_3))
375 (check-sat)
376 (pop)
377
378 (push)
379 (maximize (- A_4_decomp A_4))
380 (check-sat)
381 (pop)
382
383 (push)
384 (maximize (- A_5_decomp A_5))
385 (check-sat)
386 (pop)
387
388 (push)
389 (maximize (- A_15_decomp A_15))
390 (check-sat)
391 (pop)
392
393 (push)
394 (maximize (- A_16_decomp A_16))
395 (check-sat)
396 (pop)
397
398 (push)
399 (maximize (- A_17_decomp A_17))
400 (check-sat)
401 (pop)
402
403 (push)
404 (maximize (- B_1_decomp B_1))
405 (check-sat)
```

```
406 (pop)
407
408 (push)
409 (maximize (- B_2_decomp B_2))
410 (check-sat)
411 (pop)
412
413 (push)
414 (maximize (- B_3_decomp B_3))
415 (check-sat)
416 (pop)
417
418 (push)
419 (maximize (- B_4_decomp B_4))
420 (check-sat)
421 (pop)
422
423 (push)
424 (maximize (- B_8_decomp B_8))
425 (check-sat)
426 (pop)
427
428 (push)
429 (maximize (- B_9_decomp B_9))
430 (check-sat)
431 (pop)
432
433 (push)
434 (maximize (- B_10_decomp B_10))
435 (check-sat)
436 (pop)
437
438 (push)
439 (maximize (- B_15_decomp B_15))
440 (check-sat)
441 (pop)
442
443 (push)
444 (maximize (- C_1_decomp C_1))
445 (check-sat)
446 (pop)
447
448 (push)
449 (maximize (- C_2_decomp C_2))
450 (check-sat)
451 (pop)
452
```

```
453 (push)
454 (maximize (- C_3_decomp C_3))
455 (check-sat)
456 (pop)
457
458 (push)
459 (maximize (- C_4_decomp C_4))
460 (check-sat)
461 (pop)
462
463 (push)
464 (maximize (- C_9_decomp C_9))
465 (check-sat)
466 (pop)
467
468 (push)
469 (maximize (- D_1_decomp D_1))
470 (check-sat)
471 (pop)
472
473 (push)
474 (maximize (- D_2_decomp D_2))
475 (check-sat)
476 (pop)
477
478 (push)
479 (maximize (- D_4_decomp D_4))
480 (check-sat)
481 (pop)
482
483 (push)
484 (maximize (- D_8_decomp D_8))
485 (check-sat)
486 (pop)
```

A.3 Run-Time Detection Code (C++)

This section of the appendix includes the C++ code output for detecting incomplete and inconsistent requirements at run time in Chapter 5.

Listing A.6: Incomplete and Inconsistent Requirements Detection

```
1 #include <cmath>
2 #include <string>
3
```

```
4  void record_counterexample(std::string message) {
5      ;
6  }
7
8  double min2 (double x1, double x2) {
9      return (((x1 + x2) - (std::abs(x1 - x2))) / 2.0);
10 }
11
12 double and2(double x1, double x2) {
13     return (min2(x1, x2));
14 }
15
16 double and3(double x1, double x2, double x3) {
17     return (and2(and2(x1, x2), x3));
18 }
19
20 double and4(double x1, double x2, double x3, double x4) {
21     return (and2(and3(x1, x2, x3), x4));
22 }
23
24 double and5(double x1, double x2, double x3, double x4, double x5)
       {
25     return (and2(and4(x1, x2, x3, x4), x5));
26 }
27
28 double max2(double x1, double x2) {
29     return ((x1 + x2 + (std::abs(x1 - x2))) / 2);
30 }
31
32 double or2(double x1, double x2) {
33     return (max2(x1, x2));
34 }
35
36 double or3(double x1, double x2, double x3) {
37     return (or2(x1, or2(x2, x3)));
38 }
39
40 double or4(double x1, double x2, double x3, double x4) {
41     return (or2(x1, or3(x2, x3, x4)));
42 }
43
44 double not1(double x1) {
45     return (1.0 - x1);
46 }
47
48 double EQUALS(double a, double b) {
49     return (std::abs(a - b) < 0.0001) ? 1.0 : 0.0;
```

```
50 }
51
52 double GREATER(double a, double b) {
53     return (a > b) ? 1.0 : 0.0;
54 }
55
56 double LESS(double a, double b) {
57     return (a < b) ? 1.0 : 0.0;
58 }
59
60 double Speed_t;
61 double Speed_t_plus_1;
62 double Distance;
63 double SafeDistance;
64 double DesiredSpeed;
65 double ThrottleActuator;
66 double ThrottlePedalSensor;
67 double WheelSpeedSensor;
68 double GPSSpeedSensor;
69 double DistanceSensor1;
70 double DistanceSensor2;
71 double BrakeActuator;
72 double BrakePedalSensor;
73 double CruiseSwitchSensor;
74 double CruiseActiveSensor;
75 double CruiseActiveSwitch;
76
77 double A_1 (){
78     return 1.0;
79 }
80
81 double A_2 (){
82     return (and2(EQUALS(CruiseSwitchSensor, 0.0), (EQUALS(
        CruiseActiveSensor, 1.0))));
83 }
84
85 double A_3 (){
86     return (and2(EQUALS(CruiseSwitchSensor, 0.0), (EQUALS(
        CruiseActiveSensor, 0.0))));
87 }
88
89 double A_4 (){
90     return (and2(EQUALS(CruiseSwitchSensor, 1.0), (EQUALS(
        CruiseActiveSensor, 0.0))));
91 }
92
93 double A_5 (){
```

```
94      return (and2(EQUALS(CruiseSwitchSensor, 1.0), (EQUALS(
            CruiseActiveSensor, 1.0))));
95  }
96
97  double A_6 (){
98      return (EQUALS(CruiseSwitchSensor, 0.0));
99  }
100
101 double A_7 (){
102     return (EQUALS(CruiseActiveSensor, 1.0));
103 }
104
105 double A_8 (){
106     return (EQUALS(CruiseActiveSwitch, 0.0));
107 }
108
109 double A_9 (){
110     return (EQUALS(CruiseSwitchSensor, 0.0));
111 }
112
113 double A_10 (){
114     return (EQUALS(CruiseActiveSensor, 0.0));
115 }
116
117 double A_11 (){
118     return (EQUALS(CruiseSwitchSensor, 1.0));
119 }
120
121 double A_12 (){
122     return (EQUALS(CruiseActiveSensor, 0.0));
123 }
124
125 double A_13 (){
126     return (EQUALS(CruiseSwitchSensor, 1.0));
127 }
128
129 double A_14 (){
130     return (EQUALS(CruiseActiveSensor, 1.0));
131 }
132
133 double A_15 (){
134     return (and2(EQUALS(ThrottlePedalSensor, ThrottleActuator), (
            EQUALS(BrakePedalSensor, BrakeActuator))));
135 }
136
137 double A_16 (){
138     return (EQUALS(ThrottlePedalSensor, ThrottleActuator));
```

```
139 }
140
141 double A_17 (){
142     return (EQUALS(BrakePedalSensor, BrakeActuator));
143 }
144
145 double A_18 (){
146     return 1.0;
147 }
148
149 double A_19 (){
150     return (EQUALS(ThrottlePedalSensor, ThrottleActuator));
151 }
152
153 double A_20 (){
154     return 1.0;
155 }
156
157 double A_21 (){
158     return (EQUALS(BrakePedalSensor, BrakeActuator));
159 }
160
161 double B_1 (){
162     return (or3(EQUALS(Speed_t, Speed_t_plus_1), (GREATER(Speed_t,
        Speed_t_plus_1)), (LESS(Distance, SafeDistance))));
163 }
164
165 double B_2 (){
166     return (or2(and2(EQUALS(Speed_t, 0.0), (EQUALS(Speed_t_plus_1,
        0.0))), (GREATER(Speed_t, Speed_t_plus_1))));
167 }
168
169 double B_3 (){
170     return (or2(and2(EQUALS(Speed_t, 0.0), (EQUALS(Speed_t_plus_1,
        0.0))), (GREATER(Speed_t, Speed_t_plus_1))));
171 }
172
173 double B_4 (){
174     return (LESS(ThrottleActuator,ThrottlePedalSensor));
175 }
176
177 double B_5 (){
178     return (LESS(ThrottleActuator, ThrottlePedalSensor));
179 }
180
181 double B_6 (){
182     return 1.0;
```

```
183 }
184
185 double B_7 (){
186     return (GREATER(ThrottlePedalSensor,0.0));
187 }
188
189 double B_8 (){
190     return (or2(GREATER(Speed_t, DesiredSpeed), (LESS(Distance,
          SafeDistance))));
191 }
192
193 double B_9 (){
194     return (GREATER(Speed_t, DesiredSpeed));
195 }
196
197 double B_10 (){
198     return (LESS(Distance, SafeDistance));
199 }
200
201 double B_11 (){
202     return (GREATER(WheelSpeedSensor, DesiredSpeed));
203 }
204
205 double B_12 (){
206     return (GREATER(GPSSpeedSensor, DesiredSpeed));
207 }
208
209 double B_13 (){
210     return (LESS(DistanceSensor1, SafeDistance));
211 }
212
213 double B_14 (){
214     return (LESS(DistanceSensor2, SafeDistance));
215 }
216
217 double B_15 (){
218     return (GREATER(BrakeActuator, BrakePedalSensor));
219 }
220
221 double B_16 (){
222     return (GREATER(BrakeActuator, BrakePedalSensor));
223 }
224
225 double B_17 (){
226     return 1.0;
227 }
228
```

```
229 double B_18 (){
230     return (EQUALS(ThrottlePedalSensor, 0.0));
231 }
232
233 double B_19 (){
234     return (LESS(BrakePedalSensor, 1.0));
235 }
236
237 double C_1 (){
238     return (LESS(Speed_t, Speed_t_plus_1));
239 }
240
241 double C_2 (){
242     return (and2(LESS(Speed_t, DesiredSpeed), (GREATER(Distance,
            SafeDistance))));
243 }
244
245 double C_3 (){
246     return (GREATER(ThrottleActuator, ThrottlePedalSensor));
247 }
248
249 double C_4 (){
250     return (LESS(Speed_t, DesiredSpeed));
251 }
252
253 double C_5 (){
254     return (GREATER(ThrottleActuator, ThrottlePedalSensor));
255 }
256
257 double C_6 (){
258     return (LESS(WheelSpeedSensor, DesiredSpeed));
259 }
260
261 double C_7 (){
262     return (LESS(GPSSpeedSensor, DesiredSpeed));
263 }
264
265 double C_8 (){
266     return (LESS(ThrottlePedalSensor, 1.0));
267 }
268
269 double C_9 (){
270     return (GREATER(Distance, SafeDistance));
271 }
272
273 double C_10 (){
274     return (GREATER(DistanceSensor1, SafeDistance));
```

```
275 }
276
277 double C_11 (){
278     return (GREATER(DistanceSensor2, SafeDistance));
279 }
280
281 double C_12 (){
282     return (EQUALS(BrakeActuator, 0.0));
283 }
284
285 double C_13 (){
286     return 1.0;
287 }
288
289 double D_New1 (){
290     return (EQUALS(BrakeActuator, BrakePedalSensor));
291 }
292
293 double D_New2 (){
294     return 1.0;
295 }
296
297 double D_1 (){
298     return (EQUALS(Speed_t, Speed_t_plus_1));
299 }
300
301 double D_2 (){
302     return (and2(EQUALS(Speed_t, DesiredSpeed), (GREATER(Distance,
            SafeDistance))));
303 }
304
305 double D_3 (){
306     return (EQUALS(ThrottleActuator, ThrottlePedalSensor));
307 }
308
309 double D_4 (){
310     return (EQUALS(Speed_t, DesiredSpeed));
311 }
312
313 double D_5 (){
314     return 1.0;
315 }
316
317 double D_6 (){
318     return (EQUALS(WheelSpeedSensor, DesiredSpeed));
319 }
320
```

```
321 double D_7 (){
322     return (EQUALS(GPSSpeedSensor, DesiredSpeed));
323 }
324
325 double D_8 (){
326     return (GREATER(Distance, SafeDistance));
327 }
328
329 double D_9 (){
330     return (GREATER(DistanceSensor1, SafeDistance));
331 }
332
333 double D_10 (){
334     return (GREATER(DistanceSensor2, SafeDistance));
335 }
336
337 double A_1_decomp (){
338     return (or4(A_2(), A_3(), A_4(), A_5()));
339 }
340
341 double A_2_decomp (){
342     return (and3(A_6(), A_7(), A_8()));
343 }
344
345 double A_3_decomp (){
346     return (and3(A_9(), A_10(), A_15()));
347 }
348
349 double A_4_decomp (){
350     return (and3(A_15(), A_11(), A_12()));
351 }
352
353 double A_5_decomp (){
354     return (and3(A_13(), A_14(), B_1()));
355 }
356
357 double A_15_decomp (){
358     return (and2(A_16(), A_17()));
359 }
360
361 double A_16_decomp (){
362     return (and2(A_18(), A_19()));
363 }
364
365 double A_17_decomp (){
366     return (and2(A_20(), A_21()));
367 }
```

```
368
369 double B_1_decomp (){
370     return (and3(C_1(), B_2(), D_1()));
371 }
372
373 double B_2_decomp (){
374     return (and2(B_8(), B_3()));
375 }
376
377 double B_3_decomp (){
378     return (and2(B_4(), B_15()));
379 }
380
381 double B_4_decomp (){
382     return (and3(B_5(), B_6(), B_7()));
383 }
384
385 double B_8_decomp (){
386     return (or2(B_9(), B_10()));
387 }
388
389 double B_9_decomp (){
390     return (and2(B_11(), B_12()));
391 }
392
393 double B_10_decomp (){
394     return (and2(B_13(), B_14()));
395 }
396
397 double B_15_decomp (){
398     return (and4(B_16(), B_17(), B_18(), B_19()));
399 }
400
401 double C_1_decomp (){
402     return (and3(C_2(), C_3(), C_12()));
403 }
404
405 double C_2_decomp (){
406     return (and2(C_4(), C_9()));
407 }
408
409 double C_3_decomp (){
410     return (and3(C_5(), C_13(), C_8()));
411 }
412
413 double C_4_decomp (){
414     return (and2(C_6(), C_7()));
```

```
415 }
416
417 double C_9_decomp (){
418     return (and2(C_10(), C_11()));
419 }
420
421 double D_1_decomp (){
422     return (and5(D_2(), D_5(), D_3(), D_New1(), D_New2()));
423 }
424
425 double D_2_decomp (){
426     return (and2(D_4(), D_8()));
427 }
428
429 double D_4_decomp (){
430     return (and2(D_6(), D_7()));
431 }
432
433 double D_8_decomp (){
434     return (and2(D_9(), D_10()));
435 }
436
437 bool updated_sat_of_A_1() {
438     // Assess Parent Satisfaction
439     bool parent_sat = A_1();
440     // Assess Aggregate Child Satisfaction
441     bool child_sat = A_1_decomp();
442
443     if(parent_sat == child_sat)
444         return parent_sat;
445     else {
446         // Save variables and requirement
447         record_counterexample("A_1");
448
449         // Unsat if incomplete / incorrect
450         return false;
451     }
452 }
453
454 bool updated_sat_of_A_2() {
455     // Assess Parent Satisfaction
456     bool parent_sat = A_2();
457     // Assess Aggregate Child Satisfaction
458     bool child_sat = A_2_decomp();
459
460     if(parent_sat == child_sat)
461         return parent_sat;
```

```
462     else {
463         // Save variables and requirement
464         record_counterexample("A_2");
465 
466         // Unsat if incomplete / incorrect
467         return false;
468     }
469 }
470 
471 bool updated_sat_of_A_3() {
472     // Assess Parent Satisfaction
473     bool parent_sat = A_3();
474     // Assess Aggregate Child Satisfaction
475     bool child_sat = A_3_decomp();
476 
477     if(parent_sat == child_sat)
478         return parent_sat;
479     else {
480         // Save variables and requirement
481         record_counterexample("A_3");
482 
483         // Unsat if incomplete / incorrect
484         return false;
485     }
486 }
487 
488 bool updated_sat_of_A_4() {
489     // Assess Parent Satisfaction
490     bool parent_sat = A_4();
491     // Assess Aggregate Child Satisfaction
492     bool child_sat = A_4_decomp();
493 
494     if(parent_sat == child_sat)
495         return parent_sat;
496     else {
497         // Save variables and requirement
498         record_counterexample("A_4");
499 
500         // Unsat if incomplete / incorrect
501         return false;
502     }
503 }
504 
505 bool updated_sat_of_A_5() {
506     // Assess Parent Satisfaction
507     bool parent_sat = A_5();
508     // Assess Aggregate Child Satisfaction
```

```
509     bool child_sat = A_5_decomp();
510
511     if(parent_sat == child_sat)
512         return parent_sat;
513     else {
514         // Save variables and requirement
515         record_counterexample("A_5");
516
517         // Unsat if incomplete / incorrect
518         return false;
519     }
520 }
521
522 bool updated_sat_of_A_15() {
523     // Assess Parent Satisfaction
524     bool parent_sat = A_15();
525     // Assess Aggregate Child Satisfaction
526     bool child_sat = A_15_decomp();
527
528     if(parent_sat == child_sat)
529         return parent_sat;
530     else {
531         // Save variables and requirement
532         record_counterexample("A_15");
533
534         // Unsat if incomplete / incorrect
535         return false;
536     }
537 }
538
539 bool updated_sat_of_A_16() {
540     // Assess Parent Satisfaction
541     bool parent_sat = A_16();
542     // Assess Aggregate Child Satisfaction
543     bool child_sat = A_16_decomp();
544
545     if(parent_sat == child_sat)
546         return parent_sat;
547     else {
548         // Save variables and requirement
549         record_counterexample("A_16");
550
551         // Unsat if incomplete / incorrect
552         return false;
553     }
554 }
555
```

```
556 bool updated_sat_of_A_17() {
557     // Assess Parent Satisfaction
558     bool parent_sat = A_17();
559     // Assess Aggregate Child Satisfaction
560     bool child_sat = A_17_decomp();
561 
562     if(parent_sat == child_sat)
563         return parent_sat;
564     else {
565         // Save variables and requirement
566         record_counterexample("A_17");
567 
568         // Unsat if incomplete / incorrect
569         return false;
570     }
571 }
572 
573 bool updated_sat_of_B_1() {
574     // Assess Parent Satisfaction
575     bool parent_sat = B_1();
576     // Assess Aggregate Child Satisfaction
577     bool child_sat = B_1_decomp();
578 
579     if(parent_sat == child_sat)
580         return parent_sat;
581     else {
582         // Save variables and requirement
583         record_counterexample("B_1");
584 
585         // Unsat if incomplete / incorrect
586         return false;
587     }
588 }
589 
590 bool updated_sat_of_B_2() {
591     // Assess Parent Satisfaction
592     bool parent_sat = B_2();
593     // Assess Aggregate Child Satisfaction
594     bool child_sat = B_2_decomp();
595 
596     if(parent_sat == child_sat)
597         return parent_sat;
598     else {
599         // Save variables and requirement
600         record_counterexample("B_2");
601 
602         // Unsat if incomplete / incorrect
```

```
603         return false;
604     }
605 }
606
607 bool updated_sat_of_B_3() {
608     // Assess Parent Satisfaction
609     bool parent_sat = B_3();
610     // Assess Aggregate Child Satisfaction
611     bool child_sat = B_3_decomp();
612
613     if(parent_sat == child_sat)
614         return parent_sat;
615     else {
616         // Save variables and requirement
617         record_counterexample("B_3");
618
619         // Unsat if incomplete / incorrect
620         return false;
621     }
622 }
623
624 bool updated_sat_of_B_4() {
625     // Assess Parent Satisfaction
626     bool parent_sat = B_4();
627     // Assess Aggregate Child Satisfaction
628     bool child_sat = B_4_decomp();
629
630     if(parent_sat == child_sat)
631         return parent_sat;
632     else {
633         // Save variables and requirement
634         record_counterexample("B_4");
635
636         // Unsat if incomplete / incorrect
637         return false;
638     }
639 }
640
641 bool updated_sat_of_B_8() {
642     // Assess Parent Satisfaction
643     bool parent_sat = B_8();
644     // Assess Aggregate Child Satisfaction
645     bool child_sat = B_8_decomp();
646
647     if(parent_sat == child_sat)
648         return parent_sat;
649     else {
```

```
650         // Save variables and requirement
651         record_counterexample("B_8");
652 
653         // Unsat if incomplete / incorrect
654         return false;
655     }
656 }
657 
658 bool updated_sat_of_B_9() {
659     // Assess Parent Satisfaction
660     bool parent_sat = B_9();
661     // Assess Aggregate Child Satisfaction
662     bool child_sat = B_9_decomp();
663 
664     if(parent_sat == child_sat)
665         return parent_sat;
666     else {
667         // Save variables and requirement
668         record_counterexample("B_9");
669 
670         // Unsat if incomplete / incorrect
671         return false;
672     }
673 }
674 
675 bool updated_sat_of_B_10() {
676     // Assess Parent Satisfaction
677     bool parent_sat = B_10();
678     // Assess Aggregate Child Satisfaction
679     bool child_sat = B_10_decomp();
680 
681     if(parent_sat == child_sat)
682         return parent_sat;
683     else {
684         // Save variables and requirement
685         record_counterexample("B_10");
686 
687         // Unsat if incomplete / incorrect
688         return false;
689     }
690 }
691 
692 bool updated_sat_of_B_15() {
693     // Assess Parent Satisfaction
694     bool parent_sat = B_15();
695     // Assess Aggregate Child Satisfaction
696     bool child_sat = B_15_decomp();
```

```
697
698     if(parent_sat == child_sat)
699         return parent_sat;
700     else {
701         // Save variables and requirement
702         record_counterexample("B_15");
703
704         // Unsat if incomplete / incorrect
705         return false;
706     }
707 }
708
709 bool updated_sat_of_C_1() {
710     // Assess Parent Satisfaction
711     bool parent_sat = C_1();
712     // Assess Aggregate Child Satisfaction
713     bool child_sat = C_1_decomp();
714
715     if(parent_sat == child_sat)
716         return parent_sat;
717     else {
718         // Save variables and requirement
719         record_counterexample("C_1");
720
721         // Unsat if incomplete / incorrect
722         return false;
723     }
724 }
725
726 bool updated_sat_of_C_2() {
727     // Assess Parent Satisfaction
728     bool parent_sat = C_2();
729     // Assess Aggregate Child Satisfaction
730     bool child_sat = C_2_decomp();
731
732     if(parent_sat == child_sat)
733         return parent_sat;
734     else {
735         // Save variables and requirement
736         record_counterexample("C_2");
737
738         // Unsat if incomplete / incorrect
739         return false;
740     }
741 }
742
743 bool updated_sat_of_C_3() {
```

```
744     // Assess Parent Satisfaction
745     bool parent_sat = C_3();
746     // Assess Aggregate Child Satisfaction
747     bool child_sat = C_3_decomp();
748 
749     if(parent_sat == child_sat)
750         return parent_sat;
751     else {
752         // Save variables and requirement
753         record_counterexample("C_3");
754 
755         // Unsat if incomplete / incorrect
756         return false;
757     }
758 }
759 
760 bool updated_sat_of_C_4() {
761     // Assess Parent Satisfaction
762     bool parent_sat = C_4();
763     // Assess Aggregate Child Satisfaction
764     bool child_sat = C_4_decomp();
765 
766     if(parent_sat == child_sat)
767         return parent_sat;
768     else {
769         // Save variables and requirement
770         record_counterexample("C_4");
771 
772         // Unsat if incomplete / incorrect
773         return false;
774     }
775 }
776 
777 bool updated_sat_of_C_9() {
778     // Assess Parent Satisfaction
779     bool parent_sat = C_9();
780     // Assess Aggregate Child Satisfaction
781     bool child_sat = C_9_decomp();
782 
783     if(parent_sat == child_sat)
784         return parent_sat;
785     else {
786         // Save variables and requirement
787         record_counterexample("C_9");
788 
789         // Unsat if incomplete / incorrect
790         return false;
```

```
791     }
792 }
793
794 bool updated_sat_of_D_1() {
795     // Assess Parent Satisfaction
796     bool parent_sat = D_1();
797     // Assess Aggregate Child Satisfaction
798     bool child_sat = D_1_decomp();
799
800     if(parent_sat == child_sat)
801         return parent_sat;
802     else {
803         // Save variables and requirement
804         record_counterexample("D_1");
805
806         // Unsat if incomplete / incorrect
807         return false;
808     }
809 }
810
811 bool updated_sat_of_D_2() {
812     // Assess Parent Satisfaction
813     bool parent_sat = D_2();
814     // Assess Aggregate Child Satisfaction
815     bool child_sat = D_2_decomp();
816
817     if(parent_sat == child_sat)
818         return parent_sat;
819     else {
820         // Save variables and requirement
821         record_counterexample("D_2");
822
823         // Unsat if incomplete / incorrect
824         return false;
825     }
826 }
827
828 bool updated_sat_of_D_4() {
829     // Assess Parent Satisfaction
830     bool parent_sat = D_4();
831     // Assess Aggregate Child Satisfaction
832     bool child_sat = D_4_decomp();
833
834     if(parent_sat == child_sat)
835         return parent_sat;
836     else {
837         // Save variables and requirement
```

```
838             record_counterexample("D_4");
839
840             // Unsat if incomplete / incorrect
841             return false;
842         }
843     }
844
845     bool updated_sat_of_D_8() {
846         // Assess Parent Satisfaction
847         bool parent_sat = D_8();
848         // Assess Aggregate Child Satisfaction
849         bool child_sat = D_8_decomp();
850
851         if(parent_sat == child_sat)
852             return parent_sat;
853         else {
854             // Save variables and requirement
855             record_counterexample("D_8");
856
857             // Unsat if incomplete / incorrect
858             return false;
859         }
860     }
```

Appendix B

Feature Interaction Artifacts

This appendix presents the goal model specifications as well as the satisfiability-modulo theory (SMT) solver and code (C++) outputs from the *Phorcys* and *Thoosa* tools for feature interaction detection. Feature interaction artifacts are included in this appendix since no complete example can be given, due to issues of space, within Chapters 6 and 8.

B.1 Goal Model Specifications

This section of the appendix includes the XML schema for representing goal models in XML, as well as XML representations of the braking system goal model used in Chapters 6, 7, and 8.

B.1.1 Goal Model Schema

Listing B.1: XSD Goal Model XML Schema for Feature Interactions

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3      elementFormDefault="qualified"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5    <xs:element name="goalmodel">
6      <xs:complexType>
7        <xs:sequence>
8          <xs:element ref="goal"/>
9          <xs:element ref="environment"/>
10       </xs:sequence>
```

```
11      </xs:complexType>
12    </xs:element>
13    <xs:element name="environment">
14      <xs:complexType>
15        <xs:sequence>
16          <xs:element minOccurs="0" maxOccurs="unbounded" ref="env"/>
17        </xs:sequence>
18      </xs:complexType>
19    </xs:element>
20    <xs:element name="env">
21      <xs:complexType>
22        <xs:attribute name="name" use="required" type="xs:NCName"/>
23        <xs:attribute name="relationship" use="required" type="xs:
              string"/>
24      </xs:complexType>
25    </xs:element>
26    <xs:element name="goal">
27      <xs:complexType>
28        <xs:choice>
29          <xs:element maxOccurs="unbounded" ref="goal"/>
30          <xs:element ref="agent"/>
31        </xs:choice>
32        <xs:attribute name="contents" use="required" type="xs:string
              "/>
33        <xs:attribute name="name" use="required" type="xs:NCName"/>
34        <xs:attribute name="relationship" use="required" type="xs:
              string"/>
35        <xs:attribute name="type" use="required" type="xs:NCName"/>
36      </xs:complexType>
37    </xs:element>
38    <xs:element name="agent">
39      <xs:complexType>
40        <xs:attribute name="contents" use="required" type="xs:string
              "/>
41        <xs:attribute name="location" use="required" type="xs:NCName
              "/>
42        <xs:attribute name="name" use="required" type="xs:NCName"/>
43      </xs:complexType>
44    </xs:element>
45  </xs:schema>
```

B.1.2 Goal Model for Braking System in Chapter 6

Listing B.2: Braking System Goal Model in Chapter 6

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- created with XMLSpear -->
```

```
3 <goalmodel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation='goalmodel.xsd'>
4
5     <goal name="A" contents="Maintain(Brake System)" relationship
        ="" type="AND">
6       <goal name="A_1" contents="Maintain(Brake Force)"
          relationship="" type="OR">
7         <goal name="B" contents="Achieve(Standard Force Braking
            )" relationship="" type="AND">
8           <goal name="B_1" contents="(= SF CBF_t_plus_one)"
              relationship="" type="POST">
9             <agent name="B_1_Agent" contents="Hydraulic Brake
                Sensor" location="environment"/>
10          </goal>
11          <goal name="B_2" contents="Achieve(Apply Standard
              Force)" relationship="" type="REQ">
12            <agent name="B_2_Agent" contents="Hydraulic Brake
                Actuator" location="system"/>
13          </goal>
14          <goal name="B_3" contents="(&gt; CBF_t_plus_one 0.0)"
              relationship="" type="PRE">
15            <agent name="B_3_Agent" contents="CBF Value"
                location="environment"/>
16          </goal>
17        </goal>
18        <goal name="C" contents="Achieve(Regen Braking)"
            relationship="" type="AND">
19          <goal name="C_1" contents="(&gt; CBF_t_plus_one 0.0)"
              relationship="" type="PRE">
20            <agent name="C_1_Agent" contents="CBF Value"
                location="environment"/>
21          </goal>
22          <goal name="C_2" contents="Achieve(Regen and Standard
              Force)" relationship="" type="OR">
23            <goal name="C_3" contents="Achieve(Standard Force
                Braking)" relationship="" type="AND">
24              <goal name="C_5" contents="(= SF (*
                  CBF_t_plus_one 0.2))" relationship="" type="
                  POST">
25                <agent name="C_5_Agent" contents="Hydraulic
                    Brake Sensor" location="environment"/>
26              </goal>
27              <goal name="C_6" contents="Achieve(Apply
                  Standard Force)" relationship="" type="REQ">
28                <agent name="C_6_Agent" contents="Hydraulic
                    Brake Actuator" location="system"/>
29              </goal>
```

```
30                    <goal name="C_7" contents="(&gt; CBF_t_plus_one
                          20.0)" relationship="" type="PRE">
31                      <agent name="C_7_Agent" contents="CBF Value"
                            location="environment"/>
32                     </goal>
33                  </goal>
34                  <goal name="C_4" contents="Achieve(Regen Force)"
                        relationship="" type="AND">
35                    <goal name="C_8" contents="(&lt; CBF_t_plus_one
                          50.0)" relationship="" type="PRE">
36                      <agent name="C_7_Agent" contents="CBF Value"
                            location="environment"/>
37                     </goal>
38                    <goal name="C_9" contents="Achieve(Apply Regen
                          Force)" relationship="" type="REQ">
39                      <agent name="C_6_Agent" contents="Regeneration
                            Brake Actuator" location="system"/>
40                     </goal>
41                    <goal name="C_10" contents="(= RF (*
                          CBF_t_plus_one 0.8))" relationship="" type="
                          POST">
42                      <agent name="C_5_Agent" contents="Regeneration
                            Brake Sensor" location="environment"/>
43                     </goal>
44                  </goal>
45               </goal>
46            </goal>
47         </goal>
48      <goal name="A_2" contents="Maintain(Brake Command)"
           relationship="" type="OR">
49         <goal name="D" contents="Achieve(Brake-by-Wire)"
              relationship="" type="AND">
50            <goal name="D_2" contents="Achieve(Read CBF)"
                 relationship="" type="REQ">
51               <agent name="D_2_Agent" contents="Memory (CBF)"
                    location="system"/>
52             </goal>
53            <goal name="D_3" contents="(&gt; BF 0.0)" relationship
                 ="" type="PRE">
54               <agent name="D_3_Agent" contents="BF Value" location
                    ="environment"/>
55             </goal>
56            <goal name="D_4" contents="Achieve(Read Brake Force)"
                 relationship="" type="REQ">
57               <agent name="D_4_Agent" contents="Brake Pedal Sensor
                     (BF)" location="system"/>
58             </goal>
```

```
59          <goal name="D_5" contents="Achieve(Brake Pulse)"
                relationship="" type="OR">
60             <goal name="D_8" contents="Achieve(Brake On)"
                   relationship="" type="AND">
61                <goal name="D_10" contents="(= CBF_t 0.0)"
                      relationship="" type="PRE">
62                   <agent name="D_10_Agent" contents="CBF Value"
                         location="environment"/>
63                </goal>
64                <goal name="D_11" contents="(= CBF_t_plus_one BF)
                      " relationship="" type="POST">
65                   <agent name="D_11_Agent" contents="Brake Pedal
                         Sensor (BF)" location="environment"/>
66                </goal>
67                <goal name="D_12a" contents="Achieve(Brake Force
                      Change)" relationship="" type="REQ">
68                   <agent name="D_12a_Agent" contents="Memory (
                         CBF_t_plus_one)" location="system"/>
69                </goal>
70             </goal>
71             <goal name="D_9" contents="Achieve(Brake Off)"
                   relationship="" type="AND">
72                <goal name="D_12b" contents="Achieve(Brake Force
                      Change)" relationship="" type="REQ">
73                   <agent name="D_12b_Agent" contents="Memory (
                         CBF_t_plus_one)" location="system"/>
74                </goal>
75                <goal name="D_13" contents="(= CBF_t_plus_one
                      0.0)" relationship="" type="POST">
76                   <agent name="D_13_Agent" contents="
                         CBF_t_plus_one Value" location="environment
                         "/>
77                </goal>
78                <goal name="D_14" contents="(&amp;gt; CBF_t 0.0)"
                      relationship="" type="PRE">
79                   <agent name="D_14_Agent" contents="CBF Value"
                         location="environment"/>
80                </goal>
81             </goal>
82          </goal>
83          <goal name="D_6" contents="(= SS 1.0)" relationship=""
                type="PRE">
84             <agent name="D_6_Agent" contents="SS Value" location
                   ="environment"/>
85          </goal>
86          <goal name="D_7" contents="Achieve(Read SS)"
                relationship="" type="REQ">
```

```
87                  <agent name="D_7_Agent" contents="Slip Sensor"
                        location="system"/>
88                </goal>
89              </goal>
90              <goal name="E" contents="Achieve(Anti-Lock Braking)"
                   relationship="" type="AND">
91                <goal name="E_2" contents="Achieve(Read CBF)"
                      relationship="" type="REQ">
92                  <agent name="E_2_Agent" contents="Memory (CBF)"
                        location="system"/>
93                </goal>
94                <goal name="E_3" contents="(= CBF_t_plus_one BF)"
                      relationship="" type="POST">
95                  <agent name="E_3_Agent" contents="BF Value" location
                        ="environment"/>
96                </goal>
97                <goal name="E_4" contents="Achieve(Read Brake Force)"
                      relationship="" type="REQ">
98                  <agent name="E_4_Agent" contents="Brake Pedal Sensor
                         (BF)" location="system"/>
99                </goal>
100               <goal name="E_5" contents="(&gt; BF 0.0)" relationship
                      ="" type="PRE">
101                 <agent name="E_5_Agent" contents="BF Value" location
                        ="environment"/>
102               </goal>
103               <goal name="E_6" contents="(&lt; SS 1.0)" relationship
                      ="" type="PRE">
104                 <agent name="E_6_Agent" contents="SS Value" location
                        ="environment"/>
105               </goal>
106               <goal name="E_7" contents="Achieve(Read SS)"
                      relationship="" type="REQ">
107                 <agent name="E_7_Agent" contents="Slip Sensor"
                        location="system"/>
108               </goal>
109             </goal>
110           </goal>
111       </goal>
112
113     <environment/>
114 </goalmodel>
```

B.1.3 Goal Model for Braking System in Chapter 7 and 8

Listing B.3: Braking System Goal Model in Chapter 7 and 8

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- created with XMLSpear -->
3  <goalmodel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation='goalmodel.xsd'>
4
5      <goal name="A" contents="Maintain(Brake System)" relationship
           ="" type="AND">
6          <goal name="A_1" contents="Maintain(Brake Force)"
               relationship="" type="OR">
7              <goal name="B" contents="Achieve(Standard Force Braking
                   )" relationship="" type="AND">
8                  <goal name="B_1" contents="(= SF CBF_t_plus_one)"
                       relationship="" type="POST">
9                      <agent name="B_1_Agent" contents="Hydraulic
                           Brake Sensor" location="environment"/>
10                 </goal>
11                 <goal name="B_2" contents="Achieve(Apply Standard
                       Force)" relationship="" type="REQ">
12                     <agent name="B_2_Agent" contents="Hydraulic
                           Brake Actuator" location="system"/>
13                 </goal>
14                 <goal name="B_3" contents="(&gt; CBF_t_plus_one
                       0.0)" relationship="" type="PRE">
15                     <agent name="B_3_Agent" contents="CBF Value"
                           location="environment"/>
16                 </goal>
17             </goal>
18             <goal name="C" contents="Achieve(Regen Braking)"
                   relationship="" type="AND">
19                 <goal name="C_1" contents="(&gt; CBF_t_plus_one
                       0.0)" relationship="" type="PRE">
20                     <agent name="C_1_Agent" contents="CBF Value"
                           location="environment"/>
21                 </goal>
22                 <goal name="C_2" contents="Achieve(Regen and
                       Standard Force)" relationship="" type="OR">
23                     <goal name="C_3" contents="Achieve(Standard
                           Force Braking)" relationship="" type="AND">
24                         <goal name="C_5" contents="(= SF (*
                               CBF_t_plus_one 0.2))" relationship=""
                               type="POST">
25                             <agent name="C_5_Agent" contents="
                                   Hydraulic Brake Sensor" location="
                                   environment"/>
26                         </goal>
27                         <goal name="C_6" contents="Achieve(Apply
```

```
                                Standard Force)" relationship="" type="
                                REQ">
28                                <agent name="C_6_Agent" contents="
                                    Hydraulic Brake Actuator" location="
                                    system"/>
29                              </goal>
30                              <goal name="C_7" contents="(&gt;
                                    CBF_t_plus_one 0.2)" relationship=""
                                    type="PRE">
31                                  <agent name="C_7_Agent" contents="CBF
                                        Value" location="environment"/>
32                              </goal>
33
34                          </goal>
35                          <goal name="C_4" contents="Achieve(Regen Force)
                                " relationship="" type="AND">
36                              <goal name="C_8" contents="(&lt;
                                    CBF_t_plus_one 0.5)" relationship=""
                                    type="PRE">
37                                  <agent name="C_7_Agent" contents="CBF
                                        Value" location="environment"/>
38                              </goal>
39                              <goal name="C_9" contents="Achieve(Apply
                                    Regen Force)" relationship="" type="REQ
                                    ">
40                                  <agent name="C_6_Agent" contents="
                                        Regeneration Brake Actuator"
                                        location="system"/>
41                              </goal>
42                              <goal name="C_10" contents="(= RF (*
                                    CBF_t_plus_one 0.8))" relationship=""
                                    type="POST">
43                                  <agent name="C_5_Agent" contents="
                                        Regeneration Brake Sensor" location
                                        ="environment"/>
44                              </goal>
45                          </goal>
46                      </goal>
47                  </goal>
48              </goal>
49          <goal name="A_2" contents="Maintain(Brake Command)"
                relationship="" type="OR">
50              <goal name="D" contents="Achieve(Brake-by-Wire)"
                    relationship="" type="AND">
51                  <goal name="D_2" contents="Achieve(Read CBF)"
                        relationship="" type="REQ">
```

```
52                  <agent name="D_2_Agent" contents="Memory (CBF)"
                        location="system"/>
53              </goal>
54              <goal name="D_3" contents="(&gt; BF 0.0)"
                    relationship="" type="PRE">
55                  <agent name="D_3_Agent" contents="BF Value"
                        location="environment"/>
56              </goal>
57              <goal name="D_4" contents="Achieve(Read Brake Force
                    )" relationship="" type="REQ">
58                  <agent name="D_4_Agent" contents="Brake Pedal
                        Sensor (BF)" location="system"/>
59              </goal>
60              <goal name="D_5" contents="Achieve(Brake Pulse)"
                    relationship="" type="OR">
61                  <goal name="D_8" contents="Achieve(Brake On)"
                        relationship="" type="AND">
62                      <goal name="D_10" contents="(= CBF_t 0.0)"
                            relationship="" type="PRE">
63                          <agent name="D_10_Agent" contents="CBF
                                Value" location="environment"/>
64                      </goal>
65                      <goal name="D_11" contents="(=
                            CBF_t_plus_one BF)" relationship="" type
                            ="POST">
66                          <agent name="D_11_Agent" contents="
                                Brake Pedal Sensor (BF)" location="
                                environment"/>
67                      </goal>
68                      <goal name="D_12a" contents="Achieve(Brake
                            Force Change)" relationship="" type="REQ
                            ">
69                          <agent name="D_12a_Agent" contents="
                                Memory (CBF_t_plus_one)" location="
                                system"/>
70                      </goal>
71                  </goal>
72                  <goal name="D_9" contents="Achieve(Brake Off)"
                        relationship="" type="AND">
73                      <goal name="D_12b" contents="Achieve(Brake
                            Force Change)" relationship="" type="REQ
                            ">
74                          <agent name="D_12b_Agent" contents="
                                Memory (CBF_t_plus_one)" location="
                                system"/>
75                      </goal>
76                      <goal name="D_13" contents="(=
```

```
                    CBF_t_plus_one 0.0)" relationship=""
                    type="POST">
77                   <agent name="D_13_Agent" contents="
                        CBF_t_plus_one Value" location="
                        environment"/>
78                  </goal>
79                  <goal name="D_14" contents="(&gt; CBF_t
                        0.0)" relationship="" type="PRE">
80                   <agent name="D_14_Agent" contents="CBF
                        Value" location="environment"/>
81                  </goal>
82              </goal>
83          </goal>
84          <goal name="D_6" contents="(= SS 1.0)" relationship
                ="" type="PRE">
85              <agent name="D_6_Agent" contents="SS Value"
                    location="environment"/>
86          </goal>
87          <goal name="D_7" contents="Achieve(Read SS)"
                relationship="" type="REQ">
88              <agent name="D_7_Agent" contents="Slip Sensor"
                    location="system"/>
89          </goal>
90      </goal>
91      <goal name="E" contents="Achieve(Anti-Lock Braking)"
            relationship="" type="AND">
92          <goal name="E_2" contents="Achieve(Read CBF)"
                relationship="" type="REQ">
93              <agent name="E_2_Agent" contents="Memory (CBF)"
                     location="system"/>
94          </goal>
95          <goal name="E_3" contents="(= CBF_t_plus_one BF)"
                relationship="" type="POST">
96              <agent name="E_3_Agent" contents="BF Value"
                    location="environment"/>
97          </goal>
98          <goal name="E_4" contents="Achieve(Read Brake Force
                )" relationship="" type="REQ">
99              <agent name="E_4_Agent" contents="Brake Pedal
                    Sensor (BF)" location="system"/>
100         </goal>
101         <goal name="E_5" contents="(&gt; BF 0.0)"
                relationship="" type="PRE">
102             <agent name="E_5_Agent" contents="BF Value"
                    location="environment"/>
103         </goal>
```

```
104                 <goal name="E_6" contents="(&lt; SS 1.0)"
                        relationship="" type="PRE">
105                     <agent name="E_6_Agent" contents="SS Value"
                            location="environment"/>
106                 </goal>
107                 <goal name="E_7" contents="Achieve(Read SS)"
                        relationship="" type="REQ">
108                     <agent name="E_7_Agent" contents="Slip Sensor"
                            location="system"/>
109                 </goal>
110             </goal>
111         </goal>
112     </goal>
113
114     <environment/>
115 </goalmodel>
```

B.2 Design-Time Detection Constraints (SMT)

This section of the appendix includes the SMT constraint output for detecting feature interactions in Chapter 6. Constraint sets are listed for each of the four features in the braking system goal model.

B.2.1 Detection Constraints for Failure of Feature B

Listing B.4: Feature Interaction for Failure of Feature B

```
1  (set-option :print-success true)
2  (define-fun min2 ((x1 Real) (x2 Real)) Real (/ (- (+ x1 x2) (abs (-
       x1 x2))) 2))
3  (define-fun max2 ((x1 Real) (x2 Real)) Real (/ (+ x1 x2 (abs (- x1
     x2))) 2))
4  (declare-const BF Real)
5  (declare-const CBF_t_plus_one_D_11 Bool)
6  (declare-const RF_C_10 Bool)
7  (declare-const CBF_t_plus_one_D_13 Bool)
8  (declare-const CBF_t_D_14 Bool)
9  (declare-const SF Real)
10 (declare-const CBF_t_plus_one Real)
11 (declare-const BF_D_3 Bool)
12 (declare-const CBF_t_plus_one_C_8 Bool)
13 (declare-const CBF_t_plus_one_C_7 Bool)
14 (declare-const SS Real)
15 (declare-const SF_B_1 Bool)
```

```
16 (declare-const CBF_t_plus_one_E_3 Bool)
17 (declare-const SF_C_5 Bool)
18 (declare-const CBF_t_plus_one_B_3 Bool)
19 (declare-const BF_E_5 Bool)
20 (declare-const CBF_t_plus_one_C_1 Bool)
21 (declare-const CBF_t_D_10 Bool)
22 (declare-const CBF_t Real)
23 (declare-const RF Real)
24 (declare-const C_s Bool)
25 (declare-const B_s Bool)
26 (declare-const E_s Bool)
27 (declare-const D_s Bool)
28 (declare-const SS_D_6 Bool)
29 (declare-const SS_E_6 Bool)
30 (define-fun V_SS () Bool (or (or false SS_D_6) SS_E_6))
31 (define-fun V_SF () Bool (or (or false SF_B_1) SF_C_5))
32 (define-fun V_CBF_t () Bool (or (or false CBF_t_D_10) CBF_t_D_14))
33 (define-fun V_RF () Bool (or false RF_C_10))
34 (define-fun V_BF () Bool (or (or false BF_D_3) BF_E_5))
35 (define-fun V_CBF_t_plus_one () Bool (or (or (or (or (or (or
      false CBF_t_plus_one_B_3) CBF_t_plus_one_C_1) CBF_t_plus_one_C_7
      ) CBF_t_plus_one_C_8) CBF_t_plus_one_D_11) CBF_t_plus_one_D_13)
      CBF_t_plus_one_E_3))
36 (define-fun B_1 () Bool (and (and (= SF CBF_t_plus_one) V_SF)
      V_CBF_t_plus_one))
37 (define-fun B_2 () Bool true)
38 (define-fun B_3 () Bool (and (> CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
39 (define-fun B () Bool (and (and (and true B_1) B_2) B_3))
40 (define-fun C_1 () Bool (and (> CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
41 (define-fun C_5 () Bool (and (and (= SF (* CBF_t_plus_one 0.2))
      V_SF) V_CBF_t_plus_one))
42 (define-fun C_6 () Bool true)
43 (define-fun C_7 () Bool (and (> CBF_t_plus_one 20.0)
      V_CBF_t_plus_one))
44 (define-fun C_3 () Bool (and (and (and true C_5) C_6) C_7))
45 (define-fun C_8 () Bool (and (< CBF_t_plus_one 50.0)
      V_CBF_t_plus_one))
46 (define-fun C_9 () Bool true)
47 (define-fun C_10 () Bool (and (and (= RF (* CBF_t_plus_one 0.8))
      V_RF) V_CBF_t_plus_one))
48 (define-fun C_4 () Bool (and (and (and true C_8) C_9) C_10))
49 (define-fun C_2 () Bool (or (or false C_3) C_4))
50 (define-fun C () Bool (and (and true C_1) C_2))
51 (define-fun A_1 () Bool (or (or false B) C))
52 (define-fun D_2 () Bool true)
```

```
53 (define-fun D_3 () Bool (and (> BF 0.0) V_BF))
54 (define-fun D_4 () Bool true)
55 (define-fun D_10 () Bool (and (= CBF_t 0.0) V_CBF_t))
56 (define-fun D_11 () Bool (and (and (= CBF_t_plus_one BF) V_BF)
     V_CBF_t_plus_one))
57 (define-fun D_12a () Bool true)
58 (define-fun D_8 () Bool (and (and (and true D_10) D_11) D_12a))
59 (define-fun D_12b () Bool true)
60 (define-fun D_13 () Bool (and (= CBF_t_plus_one 0.0)
     V_CBF_t_plus_one))
61 (define-fun D_14 () Bool (and (> CBF_t 0.0) V_CBF_t))
62 (define-fun D_9 () Bool (and (and (and true D_12b) D_13) D_14))
63 (define-fun D_5 () Bool (or (or false D_8) D_9))
64 (define-fun D_6 () Bool (and (= SS 1.0) V_SS))
65 (define-fun D_7 () Bool true)
66 (define-fun D () Bool (and (and (and (and (and (and true D_2) D_3)
     D_4) D_5) D_6) D_7))
67 (define-fun E_2 () Bool true)
68 (define-fun E_3 () Bool (and (and (= CBF_t_plus_one BF) V_BF)
     V_CBF_t_plus_one))
69 (define-fun E_4 () Bool true)
70 (define-fun E_5 () Bool (and (> BF 0.0) V_BF))
71 (define-fun E_6 () Bool (and (< SS 1.0) V_SS))
72 (define-fun E_7 () Bool true)
73 (define-fun E () Bool (and (and (and (and (and (and true E_2) E_3)
     E_4) E_5) E_6) E_7))
74 (define-fun A_2 () Bool (or (or false D) E))
75 (define-fun A () Bool (and (and true A_1) A_2))
76 (define-fun B_P () Bool (and true B_3))
77 (define-fun C_3_P () Bool (and true C_7))
78 (define-fun C_4_P () Bool (and true C_8))
79 (define-fun C_2_P () Bool (or (or false C_3_P) C_4_P))
80 (define-fun C_P () Bool (and (and true C_1) C_2_P))
81 (define-fun A_1_P () Bool (or (or false B_P) C_P))
82 (define-fun D_8_P () Bool (and true D_10))
83 (define-fun D_9_P () Bool (and true D_14))
84 (define-fun D_5_P () Bool (or (or false D_8_P) D_9_P))
85 (define-fun D_P () Bool (and (and (and true D_3) D_5_P) D_6))
86 (define-fun E_P () Bool (and (and true E_5) E_6))
87 (define-fun A_2_P () Bool (or (or false D_P) E_P))
88 (define-fun A_P () Bool (and (and true A_1_P) A_2_P))
89 (define-fun B_1_N () Bool (and (and (not (= SF CBF_t_plus_one))
     V_SF) V_CBF_t_plus_one))
90 (define-fun C_5_N () Bool (and (and (not (= SF (* CBF_t_plus_one
     0.2))) V_SF) V_CBF_t_plus_one))
91 (define-fun C_10_N () Bool (and (and (not (= RF (* CBF_t_plus_one
     0.8))) V_RF) V_CBF_t_plus_one))
```

```
92 (define-fun D_11_N () Bool (and (and (not (= CBF_t_plus_one BF))
     V_BF) V_CBF_t_plus_one))
93 (define-fun D_13_N () Bool (and (not (= CBF_t_plus_one 0.0))
     V_CBF_t_plus_one))
94 (define-fun E_3_N () Bool (and (and (not (= CBF_t_plus_one BF))
     V_BF) V_CBF_t_plus_one))
95 (define-fun B_PP () Bool (and (and true B_3) (or false B_1_N)))
96 (define-fun C_3_PP () Bool (and (and true C_7) (or false C_5_N)))
97 (define-fun C_4_PP () Bool (and (and true C_8) (or false C_10_N)))
98 (define-fun C_2_PP () Bool (or (or false C_3_PP) C_4_PP))
99 (define-fun C_PP () Bool (and (and true C_1) C_2_PP))
100 (define-fun A_1_PP () Bool (or (or false B_PP) C_PP))
101 (define-fun D_8_PP () Bool (and (and true D_10) (or false D_11_N)))
102 (define-fun D_9_PP () Bool (and (and true D_14) (or false D_13_N)))
103 (define-fun D_5_PP () Bool (or (or false D_8_PP) D_9_PP))
104 (define-fun D_PP () Bool (and (and (and true D_3) D_5_PP) D_6))
105 (define-fun E_PP () Bool (and (and (and true E_5) E_6) (or false
     E_3_N)))
106 (define-fun A_2_PP () Bool (or (or false D_PP) E_PP))
107 (define-fun A_PP () Bool (and (and true A_1_PP) A_2_PP))
108 (define-fun A_Contributes () Bool A)
109 (define-fun A_1_Contributes () Bool (and A_1 A_Contributes))
110 (define-fun B_Contributes () Bool B)
111 (define-fun B_1_Contributes () Bool (and B_1 B_Contributes))
112 (define-fun B_2_Contributes () Bool (and B_2 B_Contributes))
113 (define-fun B_3_Contributes () Bool (and B_3 B_Contributes))
114 (define-fun C_Contributes () Bool C)
115 (define-fun C_1_Contributes () Bool (and C_1 C_Contributes))
116 (define-fun C_2_Contributes () Bool (and C_2 C_Contributes))
117 (define-fun C_3_Contributes () Bool (and C_3 C_2_Contributes))
118 (define-fun C_5_Contributes () Bool (and C_5 C_3_Contributes))
119 (define-fun C_6_Contributes () Bool (and C_6 C_3_Contributes))
120 (define-fun C_7_Contributes () Bool (and C_7 C_3_Contributes))
121 (define-fun C_4_Contributes () Bool (and C_4 C_2_Contributes))
122 (define-fun C_8_Contributes () Bool (and C_8 C_4_Contributes))
123 (define-fun C_9_Contributes () Bool (and C_9 C_4_Contributes))
124 (define-fun C_10_Contributes () Bool (and C_10 C_4_Contributes))
125 (define-fun A_2_Contributes () Bool (and A_2 A_Contributes))
126 (define-fun D_Contributes () Bool D)
127 (define-fun D_2_Contributes () Bool (and D_2 D_Contributes))
128 (define-fun D_3_Contributes () Bool (and D_3 D_Contributes))
129 (define-fun D_4_Contributes () Bool (and D_4 D_Contributes))
130 (define-fun D_5_Contributes () Bool (and D_5 D_Contributes))
131 (define-fun D_8_Contributes () Bool (and D_8 D_5_Contributes))
132 (define-fun D_10_Contributes () Bool (and D_10 D_8_Contributes))
133 (define-fun D_11_Contributes () Bool (and D_11 D_8_Contributes))
134 (define-fun D_12a_Contributes () Bool (and D_12a D_8_Contributes))
```

```
135 (define-fun D_9_Contributes () Bool (and D_9 D_5_Contributes))
136 (define-fun D_12b_Contributes () Bool (and D_12b D_9_Contributes))
137 (define-fun D_13_Contributes () Bool (and D_13 D_9_Contributes))
138 (define-fun D_14_Contributes () Bool (and D_14 D_9_Contributes))
139 (define-fun D_6_Contributes () Bool (and D_6 D_Contributes))
140 (define-fun D_7_Contributes () Bool (and D_7 D_Contributes))
141 (define-fun E_Contributes () Bool E)
142 (define-fun E_2_Contributes () Bool (and E_2 E_Contributes))
143 (define-fun E_3_Contributes () Bool (and E_3 E_Contributes))
144 (define-fun E_4_Contributes () Bool (and E_4 E_Contributes))
145 (define-fun E_5_Contributes () Bool (and E_5 E_Contributes))
146 (define-fun E_6_Contributes () Bool (and E_6 E_Contributes))
147 (define-fun E_7_Contributes () Bool (and E_7 E_Contributes))
148 (assert (ite B_1_Contributes (= SF_B_1 true) (= SF_B_1 false)))
149 (assert (ite B_3_Contributes (= CBF_t_plus_one_B_3 true) (=
      CBF_t_plus_one_B_3 false)))
150 (assert (ite C_1_Contributes (= CBF_t_plus_one_C_1 true) (=
      CBF_t_plus_one_C_1 false)))
151 (assert (ite C_5_Contributes (= SF_C_5 true) (= SF_C_5 false)))
152 (assert (ite C_7_Contributes (= CBF_t_plus_one_C_7 true) (=
      CBF_t_plus_one_C_7 false)))
153 (assert (ite C_8_Contributes (= CBF_t_plus_one_C_8 true) (=
      CBF_t_plus_one_C_8 false)))
154 (assert (ite C_10_Contributes (= RF_C_10 true) (= RF_C_10 false)))
155 (assert (ite D_3_Contributes (= BF_D_3 true) (= BF_D_3 false)))
156 (assert (ite D_10_Contributes (= CBF_t_D_10 true) (= CBF_t_D_10
      false)))
157 (assert (ite D_11_Contributes (= CBF_t_plus_one_D_11 true) (=
      CBF_t_plus_one_D_11 false)))
158 (assert (ite D_13_Contributes (= CBF_t_plus_one_D_13 true) (=
      CBF_t_plus_one_D_13 false)))
159 (assert (ite D_14_Contributes (= CBF_t_D_14 true) (= CBF_t_D_14
      false)))
160 (assert (ite D_6_Contributes (= SS_D_6 true) (= SS_D_6 false)))
161 (assert (ite E_3_Contributes (= CBF_t_plus_one_E_3 true) (=
      CBF_t_plus_one_E_3 false)))
162 (assert (ite E_5_Contributes (= BF_E_5 true) (= BF_E_5 false)))
163 (assert (ite E_6_Contributes (= SS_E_6 true) (= SS_E_6 false)))
164 (assert (and (or B_s C_s) (or D_s E_s)))
165 (assert (ite C_s (or C C_P) (not C)))
166 (assert (ite D_s (or D D_P) (not D)))
167 (assert (ite E_s (or E E_P) (not E)))
168 (assert (ite B_s B_PP false))
```

B.2.2 Detection Constraints for Failure of Feature C

Listing B.5: Feature Interaction for Failure of Feature C

```
1  (set-option :print-success true)
2  (define-fun min2 ((x1 Real) (x2 Real)) Real (/ (- (+ x1 x2) (abs (-
       x1 x2))) 2))
3  (define-fun max2 ((x1 Real) (x2 Real)) Real (/ (+ x1 x2 (abs (- x1
       x2))) 2))
4  (declare-const BF Real)
5  (declare-const CBF_t_plus_one_D_11 Bool)
6  (declare-const RF_C_10 Bool)
7  (declare-const CBF_t_plus_one_D_13 Bool)
8  (declare-const CBF_t_D_14 Bool)
9  (declare-const SF Real)
10 (declare-const CBF_t_plus_one Real)
11 (declare-const BF_D_3 Bool)
12 (declare-const CBF_t_plus_one_C_8 Bool)
13 (declare-const CBF_t_plus_one_C_7 Bool)
14 (declare-const SS Real)
15 (declare-const SF_B_1 Bool)
16 (declare-const CBF_t_plus_one_E_3 Bool)
17 (declare-const SF_C_5 Bool)
18 (declare-const CBF_t_plus_one_B_3 Bool)
19 (declare-const BF_E_5 Bool)
20 (declare-const CBF_t_plus_one_C_1 Bool)
21 (declare-const CBF_t_D_10 Bool)
22 (declare-const CBF_t Real)
23 (declare-const RF Real)
24 (declare-const C_s Bool)
25 (declare-const B_s Bool)
26 (declare-const E_s Bool)
27 (declare-const D_s Bool)
28 (declare-const SS_D_6 Bool)
29 (declare-const SS_E_6 Bool)
30 (define-fun V_SS () Bool (or (or false SS_D_6) SS_E_6))
31 (define-fun V_SF () Bool (or (or false SF_B_1) SF_C_5))
32 (define-fun V_CBF_t () Bool (or (or false CBF_t_D_10) CBF_t_D_14))
33 (define-fun V_RF () Bool (or false RF_C_10))
34 (define-fun V_BF () Bool (or (or false BF_D_3) BF_E_5))
35 (define-fun V_CBF_t_plus_one () Bool (or (or (or (or (or (or (or
       false CBF_t_plus_one_B_3) CBF_t_plus_one_C_1) CBF_t_plus_one_C_7
       ) CBF_t_plus_one_C_8) CBF_t_plus_one_D_11) CBF_t_plus_one_D_13)
       CBF_t_plus_one_E_3))
36 (define-fun B_1 () Bool (and (and (= SF CBF_t_plus_one) V_SF)
       V_CBF_t_plus_one))
37 (define-fun B_2 () Bool true)
```

```
38 (define-fun B_3 () Bool (and (> CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
39 (define-fun B () Bool (and (and (and true B_1) B_2) B_3))
40 (define-fun C_1 () Bool (and (> CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
41 (define-fun C_5 () Bool (and (and (= SF (* CBF_t_plus_one 0.2))
      V_SF) V_CBF_t_plus_one))
42 (define-fun C_6 () Bool true)
43 (define-fun C_7 () Bool (and (> CBF_t_plus_one 20.0)
      V_CBF_t_plus_one))
44 (define-fun C_3 () Bool (and (and (and true C_5) C_6) C_7))
45 (define-fun C_8 () Bool (and (< CBF_t_plus_one 50.0)
      V_CBF_t_plus_one))
46 (define-fun C_9 () Bool true)
47 (define-fun C_10 () Bool (and (and (= RF (* CBF_t_plus_one 0.8))
      V_RF) V_CBF_t_plus_one))
48 (define-fun C_4 () Bool (and (and (and true C_8) C_9) C_10))
49 (define-fun C_2 () Bool (or (or false C_3) C_4))
50 (define-fun C () Bool (and (and true C_1) C_2))
51 (define-fun A_1 () Bool (or (or false B) C))
52 (define-fun D_2 () Bool true)
53 (define-fun D_3 () Bool (and (> BF 0.0) V_BF))
54 (define-fun D_4 () Bool true)
55 (define-fun D_10 () Bool (and (= CBF_t 0.0) V_CBF_t))
56 (define-fun D_11 () Bool (and (and (= CBF_t_plus_one BF) V_BF)
      V_CBF_t_plus_one))
57 (define-fun D_12a () Bool true)
58 (define-fun D_8 () Bool (and (and (and true D_10) D_11) D_12a))
59 (define-fun D_12b () Bool true)
60 (define-fun D_13 () Bool (and (= CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
61 (define-fun D_14 () Bool (and (> CBF_t 0.0) V_CBF_t))
62 (define-fun D_9 () Bool (and (and (and true D_12b) D_13) D_14))
63 (define-fun D_5 () Bool (or (or false D_8) D_9))
64 (define-fun D_6 () Bool (and (= SS 1.0) V_SS))
65 (define-fun D_7 () Bool true)
66 (define-fun D () Bool (and (and (and (and (and (and true D_2) D_3)
      D_4) D_5) D_6) D_7))
67 (define-fun E_2 () Bool true)
68 (define-fun E_3 () Bool (and (and (= CBF_t_plus_one BF) V_BF)
      V_CBF_t_plus_one))
69 (define-fun E_4 () Bool true)
70 (define-fun E_5 () Bool (and (> BF 0.0) V_BF))
71 (define-fun E_6 () Bool (and (< SS 1.0) V_SS))
72 (define-fun E_7 () Bool true)
73 (define-fun E () Bool (and (and (and (and (and (and true E_2) E_3)
      E_4) E_5) E_6) E_7))
```

```
74 (define-fun A_2 () Bool (or (or false D) E))
75 (define-fun A () Bool (and (and true A_1) A_2))
76 (define-fun B_P () Bool (and true B_3))
77 (define-fun C_3_P () Bool (and true C_7))
78 (define-fun C_4_P () Bool (and true C_8))
79 (define-fun C_2_P () Bool (or (or false C_3_P) C_4_P))
80 (define-fun C_P () Bool (and (and true C_1) C_2_P))
81 (define-fun A_1_P () Bool (or (or false B_P) C_P))
82 (define-fun D_8_P () Bool (and true D_10))
83 (define-fun D_9_P () Bool (and true D_14))
84 (define-fun D_5_P () Bool (or (or false D_8_P) D_9_P))
85 (define-fun D_P () Bool (and (and (and true D_3) D_5_P) D_6))
86 (define-fun E_P () Bool (and (and true E_5) E_6))
87 (define-fun A_2_P () Bool (or (or false D_P) E_P))
88 (define-fun A_P () Bool (and (and true A_1_P) A_2_P))
89 (define-fun B_1_N () Bool (and (and (not (= SF CBF_t_plus_one))
      V_SF) V_CBF_t_plus_one))
90 (define-fun C_5_N () Bool (and (and (not (= SF (* CBF_t_plus_one
      0.2))) V_SF) V_CBF_t_plus_one))
91 (define-fun C_10_N () Bool (and (and (not (= RF (* CBF_t_plus_one
      0.8))) V_RF) V_CBF_t_plus_one))
92 (define-fun D_11_N () Bool (and (and (not (= CBF_t_plus_one BF))
      V_BF) V_CBF_t_plus_one))
93 (define-fun D_13_N () Bool (and (not (= CBF_t_plus_one 0.0))
      V_CBF_t_plus_one))
94 (define-fun E_3_N () Bool (and (and (not (= CBF_t_plus_one BF))
      V_BF) V_CBF_t_plus_one))
95 (define-fun B_PP () Bool (and (and true B_3) (or false B_1_N)))
96 (define-fun C_3_PP () Bool (and (and true C_7) (or false C_5_N)))
97 (define-fun C_4_PP () Bool (and (and true C_8) (or false C_10_N)))
98 (define-fun C_2_PP () Bool (or (or false C_3_PP) C_4_PP))
99 (define-fun C_PP () Bool (and (and true C_1) C_2_PP))
100 (define-fun A_1_PP () Bool (or (or false B_PP) C_PP))
101 (define-fun D_8_PP () Bool (and (and true D_10) (or false D_11_N)))
102 (define-fun D_9_PP () Bool (and (and true D_14) (or false D_13_N)))
103 (define-fun D_5_PP () Bool (or (or false D_8_PP) D_9_PP))
104 (define-fun D_PP () Bool (and (and (and true D_3) D_5_PP) D_6))
105 (define-fun E_PP () Bool (and (and (and true E_5) E_6) (or false
      E_3_N)))
106 (define-fun A_2_PP () Bool (or (or false D_PP) E_PP))
107 (define-fun A_PP () Bool (and (and true A_1_PP) A_2_PP))
108 (define-fun A_Contributes () Bool A)
109 (define-fun A_1_Contributes () Bool (and A_1 A_Contributes))
110 (define-fun B_Contributes () Bool B)
111 (define-fun B_1_Contributes () Bool (and B_1 B_Contributes))
112 (define-fun B_2_Contributes () Bool (and B_2 B_Contributes))
113 (define-fun B_3_Contributes () Bool (and B_3 B_Contributes))
```

```
114 (define-fun C_Contributes () Bool C)
115 (define-fun C_1_Contributes () Bool (and C_1 C_Contributes))
116 (define-fun C_2_Contributes () Bool (and C_2 C_Contributes))
117 (define-fun C_3_Contributes () Bool (and C_3 C_2_Contributes))
118 (define-fun C_5_Contributes () Bool (and C_5 C_3_Contributes))
119 (define-fun C_6_Contributes () Bool (and C_6 C_3_Contributes))
120 (define-fun C_7_Contributes () Bool (and C_7 C_3_Contributes))
121 (define-fun C_4_Contributes () Bool (and C_4 C_2_Contributes))
122 (define-fun C_8_Contributes () Bool (and C_8 C_4_Contributes))
123 (define-fun C_9_Contributes () Bool (and C_9 C_4_Contributes))
124 (define-fun C_10_Contributes () Bool (and C_10 C_4_Contributes))
125 (define-fun A_2_Contributes () Bool (and A_2 A_Contributes))
126 (define-fun D_Contributes () Bool D)
127 (define-fun D_2_Contributes () Bool (and D_2 D_Contributes))
128 (define-fun D_3_Contributes () Bool (and D_3 D_Contributes))
129 (define-fun D_4_Contributes () Bool (and D_4 D_Contributes))
130 (define-fun D_5_Contributes () Bool (and D_5 D_Contributes))
131 (define-fun D_8_Contributes () Bool (and D_8 D_5_Contributes))
132 (define-fun D_10_Contributes () Bool (and D_10 D_8_Contributes))
133 (define-fun D_11_Contributes () Bool (and D_11 D_8_Contributes))
134 (define-fun D_12a_Contributes () Bool (and D_12a D_8_Contributes))
135 (define-fun D_9_Contributes () Bool (and D_9 D_5_Contributes))
136 (define-fun D_12b_Contributes () Bool (and D_12b D_9_Contributes))
137 (define-fun D_13_Contributes () Bool (and D_13 D_9_Contributes))
138 (define-fun D_14_Contributes () Bool (and D_14 D_9_Contributes))
139 (define-fun D_6_Contributes () Bool (and D_6 D_Contributes))
140 (define-fun D_7_Contributes () Bool (and D_7 D_Contributes))
141 (define-fun E_Contributes () Bool E)
142 (define-fun E_2_Contributes () Bool (and E_2 E_Contributes))
143 (define-fun E_3_Contributes () Bool (and E_3 E_Contributes))
144 (define-fun E_4_Contributes () Bool (and E_4 E_Contributes))
145 (define-fun E_5_Contributes () Bool (and E_5 E_Contributes))
146 (define-fun E_6_Contributes () Bool (and E_6 E_Contributes))
147 (define-fun E_7_Contributes () Bool (and E_7 E_Contributes))
148 (assert (ite B_1_Contributes (= SF_B_1 true) (= SF_B_1 false)))
149 (assert (ite B_3_Contributes (= CBF_t_plus_one_B_3 true) (=
      CBF_t_plus_one_B_3 false)))
150 (assert (ite C_1_Contributes (= CBF_t_plus_one_C_1 true) (=
      CBF_t_plus_one_C_1 false)))
151 (assert (ite C_5_Contributes (= SF_C_5 true) (= SF_C_5 false)))
152 (assert (ite C_7_Contributes (= CBF_t_plus_one_C_7 true) (=
      CBF_t_plus_one_C_7 false)))
153 (assert (ite C_8_Contributes (= CBF_t_plus_one_C_8 true) (=
      CBF_t_plus_one_C_8 false)))
154 (assert (ite C_10_Contributes (= RF_C_10 true) (= RF_C_10 false)))
155 (assert (ite D_3_Contributes (= BF_D_3 true) (= BF_D_3 false)))
```

```
156 (assert (ite D_10_Contributes (= CBF_t_D_10 true) (= CBF_t_D_10
      false)))
157 (assert (ite D_11_Contributes (= CBF_t_plus_one_D_11 true) (=
      CBF_t_plus_one_D_11 false)))
158 (assert (ite D_13_Contributes (= CBF_t_plus_one_D_13 true) (=
      CBF_t_plus_one_D_13 false)))
159 (assert (ite D_14_Contributes (= CBF_t_D_14 true) (= CBF_t_D_14
      false)))
160 (assert (ite D_6_Contributes (= SS_D_6 true) (= SS_D_6 false)))
161 (assert (ite E_3_Contributes (= CBF_t_plus_one_E_3 true) (=
      CBF_t_plus_one_E_3 false)))
162 (assert (ite E_5_Contributes (= BF_E_5 true) (= BF_E_5 false)))
163 (assert (ite E_6_Contributes (= SS_E_6 true) (= SS_E_6 false)))
164 (assert (and (or B_s C_s) (or D_s E_s)))
165 (assert (ite B_s (or B B_P) (not B)))
166 (assert (ite D_s (or D D_P) (not D)))
167 (assert (ite E_s (or E E_P) (not E)))
168 (assert (ite C_s C_PP false))
```

B.2.3 Detection Constraints for Failure of Feature D

Listing B.6: Feature Interaction for Failure of Feature `D`

```
1  (set-option :print-success true)
2  (define-fun min2 ((x1 Real) (x2 Real)) Real (/ (- (+ x1 x2) (abs (-
       x1 x2))) 2))
3  (define-fun max2 ((x1 Real) (x2 Real)) Real (/ (+ x1 x2 (abs (- x1
      x2))) 2))
4  (declare-const BF Real)
5  (declare-const CBF_t_plus_one_D_11 Bool)
6  (declare-const RF_C_10 Bool)
7  (declare-const CBF_t_plus_one_D_13 Bool)
8  (declare-const CBF_t_D_14 Bool)
9  (declare-const SF Real)
10 (declare-const CBF_t_plus_one Real)
11 (declare-const BF_D_3 Bool)
12 (declare-const CBF_t_plus_one_C_8 Bool)
13 (declare-const CBF_t_plus_one_C_7 Bool)
14 (declare-const SS Real)
15 (declare-const SF_B_1 Bool)
16 (declare-const CBF_t_plus_one_E_3 Bool)
17 (declare-const SF_C_5 Bool)
18 (declare-const CBF_t_plus_one_B_3 Bool)
19 (declare-const BF_E_5 Bool)
20 (declare-const CBF_t_plus_one_C_1 Bool)
21 (declare-const CBF_t_D_10 Bool)
22 (declare-const CBF_t Real)
```

```
23 (declare-const RF Real)
24 (declare-const C_s Bool)
25 (declare-const B_s Bool)
26 (declare-const E_s Bool)
27 (declare-const D_s Bool)
28 (declare-const SS_D_6 Bool)
29 (declare-const SS_E_6 Bool)
30 (define-fun V_SS () Bool (or (or false SS_D_6) SS_E_6))
31 (define-fun V_SF () Bool (or (or false SF_B_1) SF_C_5))
32 (define-fun V_CBF_t () Bool (or (or false CBF_t_D_10) CBF_t_D_14))
33 (define-fun V_RF () Bool (or false RF_C_10))
34 (define-fun V_BF () Bool (or (or false BF_D_3) BF_E_5))
35 (define-fun V_CBF_t_plus_one () Bool (or (or (or (or (or (or
      false CBF_t_plus_one_B_3) CBF_t_plus_one_C_1) CBF_t_plus_one_C_7
      ) CBF_t_plus_one_C_8) CBF_t_plus_one_D_11) CBF_t_plus_one_D_13)
      CBF_t_plus_one_E_3))
36 (define-fun B_1 () Bool (and (and (= SF CBF_t_plus_one) V_SF)
      V_CBF_t_plus_one))
37 (define-fun B_2 () Bool true)
38 (define-fun B_3 () Bool (and (> CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
39 (define-fun B () Bool (and (and (and true B_1) B_2) B_3))
40 (define-fun C_1 () Bool (and (> CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
41 (define-fun C_5 () Bool (and (and (= SF (* CBF_t_plus_one 0.2))
      V_SF) V_CBF_t_plus_one))
42 (define-fun C_6 () Bool true)
43 (define-fun C_7 () Bool (and (> CBF_t_plus_one 20.0)
      V_CBF_t_plus_one))
44 (define-fun C_3 () Bool (and (and (and true C_5) C_6) C_7))
45 (define-fun C_8 () Bool (and (< CBF_t_plus_one 50.0)
      V_CBF_t_plus_one))
46 (define-fun C_9 () Bool true)
47 (define-fun C_10 () Bool (and (and (= RF (* CBF_t_plus_one 0.8))
      V_RF) V_CBF_t_plus_one))
48 (define-fun C_4 () Bool (and (and (and true C_8) C_9) C_10))
49 (define-fun C_2 () Bool (or (or false C_3) C_4))
50 (define-fun C () Bool (and (and true C_1) C_2))
51 (define-fun A_1 () Bool (or (or false B) C))
52 (define-fun D_2 () Bool true)
53 (define-fun D_3 () Bool (and (> BF 0.0) V_BF))
54 (define-fun D_4 () Bool true)
55 (define-fun D_10 () Bool (and (= CBF_t 0.0) V_CBF_t))
56 (define-fun D_11 () Bool (and (and (= CBF_t_plus_one BF) V_BF)
      V_CBF_t_plus_one))
57 (define-fun D_12a () Bool true)
58 (define-fun D_8 () Bool (and (and (and true D_10) D_11) D_12a))
```

```
59 (define-fun D_12b () Bool true)
60 (define-fun D_13 () Bool (and (= CBF_t_plus_one 0.0)
       V_CBF_t_plus_one))
61 (define-fun D_14 () Bool (and (> CBF_t 0.0) V_CBF_t))
62 (define-fun D_9 () Bool (and (and (and true D_12b) D_13) D_14))
63 (define-fun D_5 () Bool (or (or false D_8) D_9))
64 (define-fun D_6 () Bool (and (= SS 1.0) V_SS))
65 (define-fun D_7 () Bool true)
66 (define-fun D () Bool (and (and (and (and (and (and true D_2) D_3)
       D_4) D_5) D_6) D_7))
67 (define-fun E_2 () Bool true)
68 (define-fun E_3 () Bool (and (and (= CBF_t_plus_one BF) V_BF)
       V_CBF_t_plus_one))
69 (define-fun E_4 () Bool true)
70 (define-fun E_5 () Bool (and (> BF 0.0) V_BF))
71 (define-fun E_6 () Bool (and (< SS 1.0) V_SS))
72 (define-fun E_7 () Bool true)
73 (define-fun E () Bool (and (and (and (and (and (and true E_2) E_3)
       E_4) E_5) E_6) E_7))
74 (define-fun A_2 () Bool (or (or false D) E))
75 (define-fun A () Bool (and (and true A_1) A_2))
76 (define-fun B_P () Bool (and true B_3))
77 (define-fun C_3_P () Bool (and true C_7))
78 (define-fun C_4_P () Bool (and true C_8))
79 (define-fun C_2_P () Bool (or (or false C_3_P) C_4_P))
80 (define-fun C_P () Bool (and (and true C_1) C_2_P))
81 (define-fun A_1_P () Bool (or (or false B_P) C_P))
82 (define-fun D_8_P () Bool (and true D_10))
83 (define-fun D_9_P () Bool (and true D_14))
84 (define-fun D_5_P () Bool (or (or false D_8_P) D_9_P))
85 (define-fun D_P () Bool (and (and (and true D_3) D_5_P) D_6))
86 (define-fun E_P () Bool (and (and true E_5) E_6))
87 (define-fun A_2_P () Bool (or (or false D_P) E_P))
88 (define-fun A_P () Bool (and (and true A_1_P) A_2_P))
89 (define-fun B_1_N () Bool (and (and (not (= SF CBF_t_plus_one))
       V_SF) V_CBF_t_plus_one))
90 (define-fun C_5_N () Bool (and (and (not (= SF (* CBF_t_plus_one
       0.2))) V_SF) V_CBF_t_plus_one))
91 (define-fun C_10_N () Bool (and (and (not (= RF (* CBF_t_plus_one
       0.8))) V_RF) V_CBF_t_plus_one))
92 (define-fun D_11_N () Bool (and (and (not (= CBF_t_plus_one BF))
       V_BF) V_CBF_t_plus_one))
93 (define-fun D_13_N () Bool (and (not (= CBF_t_plus_one 0.0))
       V_CBF_t_plus_one))
94 (define-fun E_3_N () Bool (and (and (not (= CBF_t_plus_one BF))
       V_BF) V_CBF_t_plus_one))
95 (define-fun B_PP () Bool (and (and true B_3) (or false B_1_N)))
```

```
 96 (define-fun C_3_PP () Bool (and (and true C_7) (or false C_5_N)))
 97 (define-fun C_4_PP () Bool (and (and true C_8) (or false C_10_N)))
 98 (define-fun C_2_PP () Bool (or (or false C_3_PP) C_4_PP))
 99 (define-fun C_PP () Bool (and (and true C_1) C_2_PP))
100 (define-fun A_1_PP () Bool (or (or false B_PP) C_PP))
101 (define-fun D_8_PP () Bool (and (and true D_10) (or false D_11_N)))
102 (define-fun D_9_PP () Bool (and (and true D_14) (or false D_13_N)))
103 (define-fun D_5_PP () Bool (or (or false D_8_PP) D_9_PP))
104 (define-fun D_PP () Bool (and (and (and true D_3) D_5_PP) D_6))
105 (define-fun E_PP () Bool (and (and (and true E_5) E_6) (or false
      E_3_N)))
106 (define-fun A_2_PP () Bool (or (or false D_PP) E_PP))
107 (define-fun A_PP () Bool (and (and true A_1_PP) A_2_PP))
108 (define-fun A_Contributes () Bool A)
109 (define-fun A_1_Contributes () Bool (and A_1 A_Contributes))
110 (define-fun B_Contributes () Bool B)
111 (define-fun B_1_Contributes () Bool (and B_1 B_Contributes))
112 (define-fun B_2_Contributes () Bool (and B_2 B_Contributes))
113 (define-fun B_3_Contributes () Bool (and B_3 B_Contributes))
114 (define-fun C_Contributes () Bool C)
115 (define-fun C_1_Contributes () Bool (and C_1 C_Contributes))
116 (define-fun C_2_Contributes () Bool (and C_2 C_Contributes))
117 (define-fun C_3_Contributes () Bool (and C_3 C_2_Contributes))
118 (define-fun C_5_Contributes () Bool (and C_5 C_3_Contributes))
119 (define-fun C_6_Contributes () Bool (and C_6 C_3_Contributes))
120 (define-fun C_7_Contributes () Bool (and C_7 C_3_Contributes))
121 (define-fun C_4_Contributes () Bool (and C_4 C_2_Contributes))
122 (define-fun C_8_Contributes () Bool (and C_8 C_4_Contributes))
123 (define-fun C_9_Contributes () Bool (and C_9 C_4_Contributes))
124 (define-fun C_10_Contributes () Bool (and C_10 C_4_Contributes))
125 (define-fun A_2_Contributes () Bool (and A_2 A_Contributes))
126 (define-fun D_Contributes () Bool D)
127 (define-fun D_2_Contributes () Bool (and D_2 D_Contributes))
128 (define-fun D_3_Contributes () Bool (and D_3 D_Contributes))
129 (define-fun D_4_Contributes () Bool (and D_4 D_Contributes))
130 (define-fun D_5_Contributes () Bool (and D_5 D_Contributes))
131 (define-fun D_8_Contributes () Bool (and D_8 D_5_Contributes))
132 (define-fun D_10_Contributes () Bool (and D_10 D_8_Contributes))
133 (define-fun D_11_Contributes () Bool (and D_11 D_8_Contributes))
134 (define-fun D_12a_Contributes () Bool (and D_12a D_8_Contributes))
135 (define-fun D_9_Contributes () Bool (and D_9 D_5_Contributes))
136 (define-fun D_12b_Contributes () Bool (and D_12b D_9_Contributes))
137 (define-fun D_13_Contributes () Bool (and D_13 D_9_Contributes))
138 (define-fun D_14_Contributes () Bool (and D_14 D_9_Contributes))
139 (define-fun D_6_Contributes () Bool (and D_6 D_Contributes))
140 (define-fun D_7_Contributes () Bool (and D_7 D_Contributes))
141 (define-fun E_Contributes () Bool E)
```

```
142 (define-fun E_2_Contributes () Bool (and E_2 E_Contributes))
143 (define-fun E_3_Contributes () Bool (and E_3 E_Contributes))
144 (define-fun E_4_Contributes () Bool (and E_4 E_Contributes))
145 (define-fun E_5_Contributes () Bool (and E_5 E_Contributes))
146 (define-fun E_6_Contributes () Bool (and E_6 E_Contributes))
147 (define-fun E_7_Contributes () Bool (and E_7 E_Contributes))
148 (assert (ite B_1_Contributes (= SF_B_1 true) (= SF_B_1 false)))
149 (assert (ite B_3_Contributes (= CBF_t_plus_one_B_3 true) (=
      CBF_t_plus_one_B_3 false)))
150 (assert (ite C_1_Contributes (= CBF_t_plus_one_C_1 true) (=
      CBF_t_plus_one_C_1 false)))
151 (assert (ite C_5_Contributes (= SF_C_5 true) (= SF_C_5 false)))
152 (assert (ite C_7_Contributes (= CBF_t_plus_one_C_7 true) (=
      CBF_t_plus_one_C_7 false)))
153 (assert (ite C_8_Contributes (= CBF_t_plus_one_C_8 true) (=
      CBF_t_plus_one_C_8 false)))
154 (assert (ite C_10_Contributes (= RF_C_10 true) (= RF_C_10 false)))
155 (assert (ite D_3_Contributes (= BF_D_3 true) (= BF_D_3 false)))
156 (assert (ite D_10_Contributes (= CBF_t_D_10 true) (= CBF_t_D_10
      false)))
157 (assert (ite D_11_Contributes (= CBF_t_plus_one_D_11 true) (=
      CBF_t_plus_one_D_11 false)))
158 (assert (ite D_13_Contributes (= CBF_t_plus_one_D_13 true) (=
      CBF_t_plus_one_D_13 false)))
159 (assert (ite D_14_Contributes (= CBF_t_D_14 true) (= CBF_t_D_14
      false)))
160 (assert (ite D_6_Contributes (= SS_D_6 true) (= SS_D_6 false)))
161 (assert (ite E_3_Contributes (= CBF_t_plus_one_E_3 true) (=
      CBF_t_plus_one_E_3 false)))
162 (assert (ite E_5_Contributes (= BF_E_5 true) (= BF_E_5 false)))
163 (assert (ite E_6_Contributes (= SS_E_6 true) (= SS_E_6 false)))
164 (assert (and (or B_s C_s) (or D_s E_s)))
165 (assert (ite B_s (or B B_P) (not B)))
166 (assert (ite C_s (or C C_P) (not C)))
167 (assert (ite E_s (or E E_P) (not E)))
168 (assert (ite D_s D_PP false))
```

B.2.4 Detection Constraints for Failure of Feature E

Listing B.7: Feature Interaction for Failure of Feature `E`

```
1 (set-option :print-success true)
2 (define-fun min2 ((x1 Real) (x2 Real)) Real (/ (- (+ x1 x2) (abs (-
      x1 x2))) 2))
3 (define-fun max2 ((x1 Real) (x2 Real)) Real (/ (+ x1 x2 (abs (- x1
      x2))) 2))
4 (declare-const BF Real)
```

```
 5 (declare-const CBF_t_plus_one_D_11 Bool)
 6 (declare-const RF_C_10 Bool)
 7 (declare-const CBF_t_plus_one_D_13 Bool)
 8 (declare-const CBF_t_D_14 Bool)
 9 (declare-const SF Real)
10 (declare-const CBF_t_plus_one Real)
11 (declare-const BF_D_3 Bool)
12 (declare-const CBF_t_plus_one_C_8 Bool)
13 (declare-const CBF_t_plus_one_C_7 Bool)
14 (declare-const SS Real)
15 (declare-const SF_B_1 Bool)
16 (declare-const CBF_t_plus_one_E_3 Bool)
17 (declare-const SF_C_5 Bool)
18 (declare-const CBF_t_plus_one_B_3 Bool)
19 (declare-const BF_E_5 Bool)
20 (declare-const CBF_t_plus_one_C_1 Bool)
21 (declare-const CBF_t_D_10 Bool)
22 (declare-const CBF_t Real)
23 (declare-const RF Real)
24 (declare-const C_s Bool)
25 (declare-const B_s Bool)
26 (declare-const E_s Bool)
27 (declare-const D_s Bool)
28 (declare-const SS_D_6 Bool)
29 (declare-const SS_E_6 Bool)
30 (define-fun V_SS () Bool (or (or false SS_D_6) SS_E_6))
31 (define-fun V_SF () Bool (or (or false SF_B_1) SF_C_5))
32 (define-fun V_CBF_t () Bool (or (or false CBF_t_D_10) CBF_t_D_14))
33 (define-fun V_RF () Bool (or false RF_C_10))
34 (define-fun V_BF () Bool (or (or false BF_D_3) BF_E_5))
35 (define-fun V_CBF_t_plus_one () Bool (or (or (or (or (or (or (or
      false CBF_t_plus_one_B_3) CBF_t_plus_one_C_1) CBF_t_plus_one_C_7
      ) CBF_t_plus_one_C_8) CBF_t_plus_one_D_11) CBF_t_plus_one_D_13)
      CBF_t_plus_one_E_3))
36 (define-fun B_1 () Bool (and (and (= SF CBF_t_plus_one) V_SF)
      V_CBF_t_plus_one))
37 (define-fun B_2 () Bool true)
38 (define-fun B_3 () Bool (and (> CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
39 (define-fun B () Bool (and (and (and true B_1) B_2) B_3))
40 (define-fun C_1 () Bool (and (> CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
41 (define-fun C_5 () Bool (and (and (= SF (* CBF_t_plus_one 0.2))
      V_SF) V_CBF_t_plus_one))
42 (define-fun C_6 () Bool true)
43 (define-fun C_7 () Bool (and (> CBF_t_plus_one 20.0)
      V_CBF_t_plus_one))
```

```
44 (define-fun C_3 () Bool (and (and (and true C_5) C_6) C_7))
45 (define-fun C_8 () Bool (and (< CBF_t_plus_one 50.0)
      V_CBF_t_plus_one))
46 (define-fun C_9 () Bool true)
47 (define-fun C_10 () Bool (and (and (= RF (* CBF_t_plus_one 0.8))
      V_RF) V_CBF_t_plus_one))
48 (define-fun C_4 () Bool (and (and (and true C_8) C_9) C_10))
49 (define-fun C_2 () Bool (or (or false C_3) C_4))
50 (define-fun C () Bool (and (and true C_1) C_2))
51 (define-fun A_1 () Bool (or (or false B) C))
52 (define-fun D_2 () Bool true)
53 (define-fun D_3 () Bool (and (> BF 0.0) V_BF))
54 (define-fun D_4 () Bool true)
55 (define-fun D_10 () Bool (and (= CBF_t 0.0) V_CBF_t))
56 (define-fun D_11 () Bool (and (and (= CBF_t_plus_one BF) V_BF)
      V_CBF_t_plus_one))
57 (define-fun D_12a () Bool true)
58 (define-fun D_8 () Bool (and (and (and true D_10) D_11) D_12a))
59 (define-fun D_12b () Bool true)
60 (define-fun D_13 () Bool (and (= CBF_t_plus_one 0.0)
      V_CBF_t_plus_one))
61 (define-fun D_14 () Bool (and (> CBF_t 0.0) V_CBF_t))
62 (define-fun D_9 () Bool (and (and (and true D_12b) D_13) D_14))
63 (define-fun D_5 () Bool (or (or false D_8) D_9))
64 (define-fun D_6 () Bool (and (= SS 1.0) V_SS))
65 (define-fun D_7 () Bool true)
66 (define-fun D () Bool (and (and (and (and (and (and true D_2) D_3)
      D_4) D_5) D_6) D_7))
67 (define-fun E_2 () Bool true)
68 (define-fun E_3 () Bool (and (and (= CBF_t_plus_one BF) V_BF)
      V_CBF_t_plus_one))
69 (define-fun E_4 () Bool true)
70 (define-fun E_5 () Bool (and (> BF 0.0) V_BF))
71 (define-fun E_6 () Bool (and (< SS 1.0) V_SS))
72 (define-fun E_7 () Bool true)
73 (define-fun E () Bool (and (and (and (and (and (and true E_2) E_3)
      E_4) E_5) E_6) E_7))
74 (define-fun A_2 () Bool (or (or false D) E))
75 (define-fun A () Bool (and (and true A_1) A_2))
76 (define-fun B_P () Bool (and true B_3))
77 (define-fun C_3_P () Bool (and true C_7))
78 (define-fun C_4_P () Bool (and true C_8))
79 (define-fun C_2_P () Bool (or (or false C_3_P) C_4_P))
80 (define-fun C_P () Bool (and (and true C_1) C_2_P))
81 (define-fun A_1_P () Bool (or (or false B_P) C_P))
82 (define-fun D_8_P () Bool (and true D_10))
83 (define-fun D_9_P () Bool (and true D_14))
```

```
84 (define-fun D_5_P () Bool (or (or false D_8_P) D_9_P))
85 (define-fun D_P () Bool (and (and (and true D_3) D_5_P) D_6))
86 (define-fun E_P () Bool (and (and true E_5) E_6))
87 (define-fun A_2_P () Bool (or (or false D_P) E_P))
88 (define-fun A_P () Bool (and (and true A_1_P) A_2_P))
89 (define-fun B_1_N () Bool (and (and (not (= SF CBF_t_plus_one))
      V_SF) V_CBF_t_plus_one))
90 (define-fun C_5_N () Bool (and (and (not (= SF (* CBF_t_plus_one
      0.2))) V_SF) V_CBF_t_plus_one))
91 (define-fun C_10_N () Bool (and (and (not (= RF (* CBF_t_plus_one
      0.8))) V_RF) V_CBF_t_plus_one))
92 (define-fun D_11_N () Bool (and (and (not (= CBF_t_plus_one BF))
      V_BF) V_CBF_t_plus_one))
93 (define-fun D_13_N () Bool (and (not (= CBF_t_plus_one 0.0))
      V_CBF_t_plus_one))
94 (define-fun E_3_N () Bool (and (and (not (= CBF_t_plus_one BF))
      V_BF) V_CBF_t_plus_one))
95 (define-fun B_PP () Bool (and (and true B_3) (or false B_1_N)))
96 (define-fun C_3_PP () Bool (and (and true C_7) (or false C_5_N)))
97 (define-fun C_4_PP () Bool (and (and true C_8) (or false C_10_N)))
98 (define-fun C_2_PP () Bool (or (or false C_3_PP) C_4_PP))
99 (define-fun C_PP () Bool (and (and true C_1) C_2_PP))
100 (define-fun A_1_PP () Bool (or (or false B_PP) C_PP))
101 (define-fun D_8_PP () Bool (and (and true D_10) (or false D_11_N)))
102 (define-fun D_9_PP () Bool (and (and true D_14) (or false D_13_N)))
103 (define-fun D_5_PP () Bool (or (or false D_8_PP) D_9_PP))
104 (define-fun D_PP () Bool (and (and (and true D_3) D_5_PP) D_6))
105 (define-fun E_PP () Bool (and (and (and true E_5) E_6) (or false
      E_3_N)))
106 (define-fun A_2_PP () Bool (or (or false D_PP) E_PP))
107 (define-fun A_PP () Bool (and (and true A_1_PP) A_2_PP))
108 (define-fun A_Contributes () Bool A)
109 (define-fun A_1_Contributes () Bool (and A_1 A_Contributes))
110 (define-fun B_Contributes () Bool B)
111 (define-fun B_1_Contributes () Bool (and B_1 B_Contributes))
112 (define-fun B_2_Contributes () Bool (and B_2 B_Contributes))
113 (define-fun B_3_Contributes () Bool (and B_3 B_Contributes))
114 (define-fun C_Contributes () Bool C)
115 (define-fun C_1_Contributes () Bool (and C_1 C_Contributes))
116 (define-fun C_2_Contributes () Bool (and C_2 C_Contributes))
117 (define-fun C_3_Contributes () Bool (and C_3 C_2_Contributes))
118 (define-fun C_5_Contributes () Bool (and C_5 C_3_Contributes))
119 (define-fun C_6_Contributes () Bool (and C_6 C_3_Contributes))
120 (define-fun C_7_Contributes () Bool (and C_7 C_3_Contributes))
121 (define-fun C_4_Contributes () Bool (and C_4 C_2_Contributes))
122 (define-fun C_8_Contributes () Bool (and C_8 C_4_Contributes))
123 (define-fun C_9_Contributes () Bool (and C_9 C_4_Contributes))
```

```
124 (define-fun C_10_Contributes () Bool (and C_10 C_4_Contributes))
125 (define-fun A_2_Contributes () Bool (and A_2 A_Contributes))
126 (define-fun D_Contributes () Bool D)
127 (define-fun D_2_Contributes () Bool (and D_2 D_Contributes))
128 (define-fun D_3_Contributes () Bool (and D_3 D_Contributes))
129 (define-fun D_4_Contributes () Bool (and D_4 D_Contributes))
130 (define-fun D_5_Contributes () Bool (and D_5 D_Contributes))
131 (define-fun D_8_Contributes () Bool (and D_8 D_5_Contributes))
132 (define-fun D_10_Contributes () Bool (and D_10 D_8_Contributes))
133 (define-fun D_11_Contributes () Bool (and D_11 D_8_Contributes))
134 (define-fun D_12a_Contributes () Bool (and D_12a D_8_Contributes))
135 (define-fun D_9_Contributes () Bool (and D_9 D_5_Contributes))
136 (define-fun D_12b_Contributes () Bool (and D_12b D_9_Contributes))
137 (define-fun D_13_Contributes () Bool (and D_13 D_9_Contributes))
138 (define-fun D_14_Contributes () Bool (and D_14 D_9_Contributes))
139 (define-fun D_6_Contributes () Bool (and D_6 D_Contributes))
140 (define-fun D_7_Contributes () Bool (and D_7 D_Contributes))
141 (define-fun E_Contributes () Bool E)
142 (define-fun E_2_Contributes () Bool (and E_2 E_Contributes))
143 (define-fun E_3_Contributes () Bool (and E_3 E_Contributes))
144 (define-fun E_4_Contributes () Bool (and E_4 E_Contributes))
145 (define-fun E_5_Contributes () Bool (and E_5 E_Contributes))
146 (define-fun E_6_Contributes () Bool (and E_6 E_Contributes))
147 (define-fun E_7_Contributes () Bool (and E_7 E_Contributes))
148 (assert (ite B_1_Contributes (= SF_B_1 true) (= SF_B_1 false)))
149 (assert (ite B_3_Contributes (= CBF_t_plus_one_B_3 true) (=
      CBF_t_plus_one_B_3 false)))
150 (assert (ite C_1_Contributes (= CBF_t_plus_one_C_1 true) (=
      CBF_t_plus_one_C_1 false)))
151 (assert (ite C_5_Contributes (= SF_C_5 true) (= SF_C_5 false)))
152 (assert (ite C_7_Contributes (= CBF_t_plus_one_C_7 true) (=
      CBF_t_plus_one_C_7 false)))
153 (assert (ite C_8_Contributes (= CBF_t_plus_one_C_8 true) (=
      CBF_t_plus_one_C_8 false)))
154 (assert (ite C_10_Contributes (= RF_C_10 true) (= RF_C_10 false)))
155 (assert (ite D_3_Contributes (= BF_D_3 true) (= BF_D_3 false)))
156 (assert (ite D_10_Contributes (= CBF_t_D_10 true) (= CBF_t_D_10
      false)))
157 (assert (ite D_11_Contributes (= CBF_t_plus_one_D_11 true) (=
      CBF_t_plus_one_D_11 false)))
158 (assert (ite D_13_Contributes (= CBF_t_plus_one_D_13 true) (=
      CBF_t_plus_one_D_13 false)))
159 (assert (ite D_14_Contributes (= CBF_t_D_14 true) (= CBF_t_D_14
      false)))
160 (assert (ite D_6_Contributes (= SS_D_6 true) (= SS_D_6 false)))
161 (assert (ite E_3_Contributes (= CBF_t_plus_one_E_3 true) (=
      CBF_t_plus_one_E_3 false)))
```

```
(assert (ite E_5_Contributes (= BF_E_5 true) (= BF_E_5 false)))
(assert (ite E_6_Contributes (= SS_E_6 true) (= SS_E_6 false)))
(assert (and (or B_s C_s) (or D_s E_s)))
(assert (ite B_s (or B B_P) (not B)))
(assert (ite C_s (or C C_P) (not C)))
(assert (ite D_s (or D D_P) (not D)))
(assert (ite E_s E_PP false))
```

B.3 Run-Time Detection Code (C++)

This section of the appendix includes the C++ code output for detecting feature interactions at run time in Chapter 8. All four features in the braking system goal model are detected with this single code listing.

Listing B.8: Feature Interaction for Failure of Feature `B`, `C`, `D`, and `E`

```
double BF;
bool CBF_t_plus_one_D_11;
bool D_precondition;
bool RF_C_10;
bool C_precondition;
bool CBF_t_plus_one_D_13;
bool E_precondition;
bool CBF_t_D_14;
double SF;
bool topLevelSatisfation;
bool D_fails;
double CBF_t_plus_one;
bool B_fails;
bool BF_D_3;
bool CBF_t_plus_one_C_8;
bool CBF_t_plus_one_C_7;
double SS;
bool SF_B_1;
bool CBF_t_plus_one_E_3;
bool SF_C_5;
bool CBF_t_plus_one_B_3;
bool BF_E_5;
bool CBF_t_plus_one_C_1;
bool E_fails;
bool CBF_t_D_10;
bool C_fails;
double CBF_t;
double RF;
bool B_precondition;
```

```
30 bool SS_D_6;
31 bool SS_E_6;
32 bool V_SS() {
33     return true;
34 }
35 bool V_SF() {
36     return true;
37 }
38 bool V_CBF_t() {
39     return true;
40 }
41 bool V_RF() {
42     return true;
43 }
44 bool V_BF() {
45     return true;
46 }
47 bool V_CBF_t_plus_one() {
48     return true;
49 }
50 bool B_1() {
51     return (((std::abs(SF - CBF_t_plus_one) < 0.0001)
52               && V_SF()) && V_CBF_t_plus_one());
53 }
54 bool B_2() {
55     return true;
56 }
57 bool B_3() {
58     return ((CBF_t_plus_one > 0.0) && V_CBF_t_plus_one());
59 }
60 bool B() {
61     return (((true && B_1()) && B_2()) && B_3());
62 }
63 bool C_1() {
64     return ((CBF_t_plus_one > 0.0) && V_CBF_t_plus_one());
65 }
66 bool C_5() {
67     return (((std::abs(SF - (CBF_t_plus_one * 0.2)) < 0.0001)
68               && V_SF()) && V_CBF_t_plus_one());
69 }
70 bool C_6() {
71     return true;
72 }
73 bool C_7() {
74     return ((CBF_t_plus_one > 0.2) && V_CBF_t_plus_one());
75 }
76 bool C_3() {
```

```
77      return (((true && C_5()) && C_6()) && C_7());
78  }
79  bool C_8() {
80      return ((CBF_t_plus_one < 0.5) && V_CBF_t_plus_one());
81  }
82  bool C_9() {
83      return true;
84  }
85  bool C_10() {
86      return (((std::abs(RF - (CBF_t_plus_one * 0.8)) < 0.0001)
87               && V_RF()) && V_CBF_t_plus_one());
88  }
89  bool C_4() {
90      return (((true && C_8()) && C_9()) && C_10());
91  }
92  bool C_2() {
93      return ((false || C_3()) || C_4());
94  }
95  bool C() {
96      return ((true && C_1()) && C_2());
97  }
98  bool A_1() {
99      return ((false || B()) || C());
100 }
101 bool D_2() {
102     return true;
103 }
104 bool D_3() {
105     return ((BF > 0.0) && V_BF());
106 }
107 bool D_4() {
108     return true;
109 }
110 bool D_10() {
111     return ((std::abs(CBF_t - 0.0) < 0.0001) && V_CBF_t());
112 }
113 bool D_11() {
114     return (((std::abs(CBF_t_plus_one - BF) < 0.0001)
115              && V_BF()) && V_CBF_t_plus_one());
116 }
117 bool D_12a() {
118     return true;
119 }
120 bool D_8() {
121     return (((true && D_10()) && D_11()) && D_12a());
122 }
123 bool D_12b() {
```

```
124     return true;
125 }
126 bool D_13() {
127     return ((std::abs(CBF_t_plus_one - 0.0) < 0.0001)
128             && V_CBF_t_plus_one());
129 }
130 bool D_14() {
131     return ((CBF_t > 0.0) && V_CBF_t());
132 }
133 bool D_9() {
134     return (((true && D_12b()) && D_13()) && D_14());
135 }
136 bool D_5() {
137     return ((false || D_8()) || D_9());
138 }
139 bool D_6() {
140     return ((std::abs(SS - 1.0) < 0.0001) && V_SS());
141 }
142 bool D_7() {
143     return true;
144 }
145 bool D() {
146     return ((((((true && D_2()) && D_3()) && D_4())
147               && D_5()) && D_6()) && D_7());
148 }
149 bool E_2() {
150     return true;
151 }
152 bool E_3() {
153     return (((std::abs(CBF_t_plus_one - BF) < 0.0001)
154              && V_BF()) && V_CBF_t_plus_one());
155 }
156 bool E_4() {
157     return true;
158 }
159 bool E_5() {
160     return ((BF > 0.0) && V_BF());
161 }
162 bool E_6() {
163     return ((SS < 1.0) && V_SS());
164 }
165 bool E_7() {
166     return true;
167 }
168 bool E() {
169     return ((((((true && E_2()) && E_3()) && E_4())
170               && E_5()) && E_6()) && E_7());
```

```
171 }
172 bool A_2() {
173     return ((false || D()) || E());
174 }
175 bool A() {
176     return ((true && A_1()) && A_2());
177 }
178 bool B_P() {
179     return (true && B_3());
180 }
181 bool C_3_P() {
182     return (true && C_7());
183 }
184 bool C_4_P() {
185     return (true && C_8());
186 }
187 bool C_2_P() {
188     return ((false || C_3_P()) || C_4_P());
189 }
190 bool C_P() {
191     return ((true && C_1()) && C_2_P());
192 }
193 bool A_1_P() {
194     return ((false || B_P()) || C_P());
195 }
196 bool D_8_P() {
197     return (true && D_10());
198 }
199 bool D_9_P() {
200     return (true && D_14());
201 }
202 bool D_5_P() {
203     return ((false || D_8_P()) || D_9_P());
204 }
205 bool D_P() {
206     return (((true && D_3()) && D_5_P()) && D_6());
207 }
208 bool E_P() {
209     return ((true && E_5()) && E_6());
210 }
211 bool A_2_P() {
212     return ((false || D_P()) || E_P());
213 }
214 bool A_P() {
215     return ((true && A_1_P()) && A_2_P());
216 }
217 bool B_1_N() {
```

```
218     return (((!(std::abs(SF - CBF_t_plus_one) < 0.0001))
219              && V_SF()) && V_CBF_t_plus_one());
220 }
221 bool C_5_N() {
222     return (((!(std::abs(SF - (CBF_t_plus_one * 0.2)) < 0.0001))
223              && V_SF()) && V_CBF_t_plus_one());
224 }
225 bool C_10_N() {
226     return (((!(std::abs(RF - (CBF_t_plus_one * 0.8)) < 0.0001))
227              && V_RF()) && V_CBF_t_plus_one());
228 }
229 bool D_11_N() {
230     return (((!(std::abs(CBF_t_plus_one - BF) < 0.0001))
231              && V_BF()) && V_CBF_t_plus_one());
232 }
233 bool D_13_N() {
234     return ((!(std::abs(CBF_t_plus_one - 0.0) < 0.0001))
235             && V_CBF_t_plus_one());
236 }
237 bool E_3_N() {
238     return (((!(std::abs(CBF_t_plus_one - BF) < 0.0001))
239              && V_BF()) && V_CBF_t_plus_one());
240 }
241 bool B_PP() {
242     return ((true && B_3()) && (false || B_1_N()));
243 }
244 bool C_3_PP() {
245     return ((true && C_7()) && (false || C_5_N()));
246 }
247 bool C_4_PP() {
248     return ((true && C_8()) && (false || C_10_N()));
249 }
250 bool C_2_PP() {
251     return ((false || C_3_PP()) || C_4_PP());
252 }
253 bool C_PP() {
254     return ((true && C_1()) && C_2_PP());
255 }
256 bool A_1_PP() {
257     return ((false || B_PP()) || C_PP());
258 }
259 bool D_8_PP() {
260     return ((true && D_10()) && (false || D_11_N()));
261 }
262 bool D_9_PP() {
263     return ((true && D_14()) && (false || D_13_N()));
264 }
```

```
265 bool D_5_PP() {
266     return ((false || D_8_PP()) || D_9_PP());
267 }
268 bool D_PP() {
269     return (((true && D_3()) && D_5_PP()) && D_6());
270 }
271 bool E_PP() {
272     return (((true && E_5()) && E_6()) && (false || E_3_N()));
273 }
274 bool A_2_PP() {
275     return ((false || D_PP()) || E_PP());
276 }
277 bool A_PP() {
278     return ((true && A_1_PP()) && A_2_PP());
279 }
280 bool A_Contributes() {
281     return A();
282 }
283 bool A_1_Contributes() {
284     return (A_1() && A_Contributes());
285 }
286 bool B_Contributes() {
287     return (B() && A_1_Contributes());
288 }
289 bool B_1_Contributes() {
290     return (B_1() && B_Contributes());
291 }
292 bool B_2_Contributes() {
293     return (B_2() && B_Contributes());
294 }
295 bool B_3_Contributes() {
296     return (B_3() && B_Contributes());
297 }
298 bool C_Contributes() {
299     return (C() && A_1_Contributes());
300 }
301 bool C_1_Contributes() {
302     return (C_1() && C_Contributes());
303 }
304 bool C_2_Contributes() {
305     return (C_2() && C_Contributes());
306 }
307 bool C_3_Contributes() {
308     return (C_3() && C_2_Contributes());
309 }
310 bool C_5_Contributes() {
311     return (C_5() && C_3_Contributes());
```

```
312 }
313 bool C_6_Contributes() {
314     return (C_6() && C_3_Contributes());
315 }
316 bool C_7_Contributes() {
317     return (C_7() && C_3_Contributes());
318 }
319 bool C_4_Contributes() {
320     return (C_4() && C_2_Contributes());
321 }
322 bool C_8_Contributes() {
323     return (C_8() && C_4_Contributes());
324 }
325 bool C_9_Contributes() {
326     return (C_9() && C_4_Contributes());
327 }
328 bool C_10_Contributes() {
329     return (C_10() && C_4_Contributes());
330 }
331 bool A_2_Contributes() {
332     return (A_2() && A_Contributes());
333 }
334 bool D_Contributes() {
335     return (D() && A_2_Contributes());
336 }
337 bool D_2_Contributes() {
338     return (D_2() && D_Contributes());
339 }
340 bool D_3_Contributes() {
341     return (D_3() && D_Contributes());
342 }
343 bool D_4_Contributes() {
344     return (D_4() && D_Contributes());
345 }
346 bool D_5_Contributes() {
347     return (D_5() && D_Contributes());
348 }
349 bool D_8_Contributes() {
350     return (D_8() && D_5_Contributes());
351 }
352 bool D_10_Contributes() {
353     return (D_10() && D_8_Contributes());
354 }
355 bool D_11_Contributes() {
356     return (D_11() && D_8_Contributes());
357 }
358 bool D_12a_Contributes() {
```

```
359     return (D_12a() && D_8_Contributes());
360 }
361 bool D_9_Contributes() {
362     return (D_9() && D_5_Contributes());
363 }
364 bool D_12b_Contributes() {
365     return (D_12b() && D_9_Contributes());
366 }
367 bool D_13_Contributes() {
368     return (D_13() && D_9_Contributes());
369 }
370 bool D_14_Contributes() {
371     return (D_14() && D_9_Contributes());
372 }
373 bool D_6_Contributes() {
374     return (D_6() && D_Contributes());
375 }
376 bool D_7_Contributes() {
377     return (D_7() && D_Contributes());
378 }
379 bool E_Contributes() {
380     return (E() && A_2_Contributes());
381 }
382 bool E_2_Contributes() {
383     return (E_2() && E_Contributes());
384 }
385 bool E_3_Contributes() {
386     return (E_3() && E_Contributes());
387 }
388 bool E_4_Contributes() {
389     return (E_4() && E_Contributes());
390 }
391 bool E_5_Contributes() {
392     return (E_5() && E_Contributes());
393 }
394 bool E_6_Contributes() {
395     return (E_6() && E_Contributes());
396 }
397 bool E_7_Contributes() {
398     return (E_7() && E_Contributes());
399 }
400 void updateAssertions() {
401         (B_1_Contributes() ? (SF_B_1 = true):(SF_B_1 = false));
402         (B_3_Contributes() ? (CBF_t_plus_one_B_3 = true):
403     (CBF_t_plus_one_B_3 = false));
404         (C_1_Contributes() ? (CBF_t_plus_one_C_1 = true):
405     (CBF_t_plus_one_C_1 = false));
```

```
406         (C_5_Contributes() ? (SF_C_5 = true):(SF_C_5 = false));
407         (C_7_Contributes() ? (CBF_t_plus_one_C_7 = true):
408       (CBF_t_plus_one_C_7 = false));
409         (C_8_Contributes() ? (CBF_t_plus_one_C_8 = true):
410       (CBF_t_plus_one_C_8 = false));
411         (C_10_Contributes() ? (RF_C_10 = true):(RF_C_10 = false));
412         (D_3_Contributes() ? (BF_D_3 = true):(BF_D_3 = false));
413         (D_10_Contributes() ? (CBF_t_D_10 = true):(CBF_t_D_10 =
              false));
414         (D_11_Contributes() ? (CBF_t_plus_one_D_11 = true):
415       (CBF_t_plus_one_D_11 = false));
416         (D_13_Contributes() ? (CBF_t_plus_one_D_13 = true):
417       (CBF_t_plus_one_D_13 = false));
418         (D_14_Contributes() ? (CBF_t_D_14 = true):(CBF_t_D_14 =
              false));
419         (D_6_Contributes() ? (SS_D_6 = true):(SS_D_6 = false));
420         (E_3_Contributes() ? (CBF_t_plus_one_E_3 = true):
421       (CBF_t_plus_one_E_3 = false));
422         (E_5_Contributes() ? (BF_E_5 = true):(BF_E_5 = false));
423         (E_6_Contributes() ? (SS_E_6 = true):(SS_E_6 = false));
424         (topLevelSatisfation = ((B() || C()) && (D() || E())));
425         (B_precondition = B_P());
426         (C_precondition = C_P());
427         (D_precondition = D_P());
428         (E_precondition = E_P());
429         (B_fails = B_PP());
430         (C_fails = C_PP());
431         (D_fails = D_PP());
432         (E_fails = E_PP());
433 }
```

Appendix C

Non-Functional Feature Interaction Artifacts

This appendix presents the goal model specifications as well as the satisfiability-modulo theory (SMT) solver and code (C++) outputs from the *Phorcys* and *Thoosa* tools used by *Soter* for non-functional feature interaction detection from Chapter 9.

C.1 Goal Model Specifications

This section of the appendix includes the XML schema for representing goal models in XML, as well as XML representations of the braking system goal model use in Chapter 9.

C.1.1 Goal Model Schema

Listing C.1: XSD Goal Model XML Schema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3      elementFormDefault="qualified"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5    <xs:element name="goalmodel">
6      <xs:complexType>
7        <xs:sequence>
8          <xs:element ref="goal"/>
9          <xs:element ref="environment"/>
10       </xs:sequence>
```

```
11      </xs:complexType>
12    </xs:element>
13    <xs:element name="environment">
14      <xs:complexType>
15        <xs:sequence>
16          <xs:element minOccurs="0" maxOccurs="unbounded" ref="env"/>
17        </xs:sequence>
18      </xs:complexType>
19    </xs:element>
20    <xs:element name="env">
21      <xs:complexType>
22        <xs:attribute name="name" use="required" type="xs:NCName"/>
23        <xs:attribute name="relationship" use="required" type="xs:
              string"/>
24      </xs:complexType>
25    </xs:element>
26    <xs:element name="goal">
27      <xs:complexType>
28        <xs:choice>
29          <xs:element maxOccurs="unbounded" ref="goal"/>
30          <xs:element ref="agent"/>
31        </xs:choice>
32        <xs:attribute name="contents" use="required" type="xs:string
              "/>
33        <xs:attribute name="name" use="required" type="xs:NCName"/>
34        <xs:attribute name="relationship" use="required" type="xs:
              string"/>
35        <xs:attribute name="type" use="required" type="xs:NCName"/>
36      </xs:complexType>
37    </xs:element>
38    <xs:element name="agent">
39      <xs:complexType>
40        <xs:attribute name="contents" use="required" type="xs:string
              "/>
41        <xs:attribute name="location" use="required" type="xs:NCName
              "/>
42        <xs:attribute name="name" use="required" type="xs:NCName"/>
43      </xs:complexType>
44    </xs:element>
45  </xs:schema>
```

C.1.2 Goal Model for Braking System in Chapter 9

Listing C.2: Woven Braking System Goal Model in Chapter 9

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- created with XMLSpear -->
```

```
3 <goalmodel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation='goalmodel.xsd'>
4
5      <goal name="AA" contents="Woven Model" relationship="" type="
           AND">
6
7          <goal name="WM" contents="AND WEAK MITIGATIONS"
               relationship="" type="AND">
8              <goal name="WMS" contents="Weak Mitigation: Minimize
                   Collision" relationship="" type="OR">
9              <goal name="TMS" contents="NA" relationship="" type="
                   AND">
10                 <goal name="TMS_PT" contents="NA" relationship=""
                       type="OR">
11                     <goal name="TMS_PT_1" contents="(&lt; FDSensor
                           0.0)" relationship="" type="PRE">
12                         <agent name="TMS_PT_1_Agent" contents="
                               Battery Charge" location="environment"/>
13                     </goal>
14                     <goal name="TMS_PT_2" contents="(= FDSensor
                           0.0)" relationship="" type="PRE">
15                         <agent name="TMS_PT_2_Agent" contents="
                               Battery Charge" location="environment"/>
16                     </goal>
17                 </goal>
18                 <goal name="TMS_M" contents="M" relationship=""
                       type="AND">
19                     <goal name="TMS_M_1" contents="(= SF 1.0)"
                           relationship="" type="POST">
20                         <agent name="TMS_M_1_Agent" contents="
                               Standard Brakes" location="system"/>
21                     </goal>
22                     <goal name="TMS_M_2" contents="(= RF 0.0)"
                           relationship="" type="POST">
23                         <agent name="TMS_M_2_Agent" contents="Regen
                               Brakes" location="system"/>
24                     </goal>
25                 </goal>
26             </goal>
27             <goal name="TMS_NOT_PT" contents="(&gt; FDSensor 0)"
                   relationship="" type="PRE">
28                 <agent name="TMS_PT_Agent" contents="" location="
                       environemnt"/>
29             </goal>
30         </goal>
31
```

```
32          <goal name="WMP" contents="Weak Mitigation: Maintain
                Charge" relationship="" type="OR">
33          <goal name="TMP" contents="Trigger and Mitigation"
                relationship="" type="AND">
34           <goal name="TMP_PT_1" contents="(&amp;lt; BCHARGE 0.5)"
                 relationship="" type="PRE">
35            <agent name="TMP_PT_Agent" contents="Battery
                 Charge" location="environment"/>
36           </goal>
37           <goal name="TMP_M" contents="M" relationship=""
                type="AND">
38            <goal name="TMP_M_1" contents="Achieve(Apply
                 Full Regen Force)" relationship="" type="REQ
                 ">
39             <agent name="TM_M_1_Agent" contents="Regen
                 Brakes" location="system"/>
40            </goal>
41            <goal name="TMP_M_2" contents="(= RF (*
                 CBF_t_plus_one 1.0))" relationship="" type="
                 POST">
42             <agent name="TM_M_2_Agent" contents="Regen
                 Brakes" location="system"/>
43            </goal>
44           </goal>
45          </goal>
46          <goal name="TMP_NOT_PT" contents="NA" relationship=""
                type="AND">
47           <goal name="TMP_NOT_PT_1" contents="(&amp;gt; BCHARGE
                0.5)" relationship="" type="POST">
48            <agent name="TM_NOT_PT_1_Agent" contents="
                 Battery Charge" location="environment"/>
49           </goal>
50           <goal name="TMP_NOT_PT_2" contents="(= BCHARGE 0.5)
                " relationship="" type="POST">
51            <agent name="TM_NOT_PT_2_Agent" contents="
                 Battery Charge" location="environment"/>
52           </goal>
53          </goal>
54        </goal>
55        </goal>
56
57        <goal name="SMF" contents="AND STRONG MITIGATIONS
              FUNCTIONAL" relationship="" type="AND">
58          <goal name="SM" contents="Strong Mitigation: Maintain
                Safe Charge" relationship="" type="OR">
59
```

```
60              <goal name="TM_PT" contents="NA" relationship="" type="
                    AND">
61                  <goal name="NOT_PT_M" contents="(&lt; BCHARGE 1.0)"
                        relationship="" type="POST">
62                      <agent name="NOT_PT_M_Agent" contents="Battery
                            Charge" location="environment"/>
63                  </goal>
64
65                  <goal name="M" contents="NA" relationship="" type="
                        AND">
66
67                      <goal name="M_1" contents="(&gt; BCHARGE 0.99)"
                            relationship="" type="PRE">
68                          <agent name="M_1_Agent" contents="Battery
                                Charge" location="environment"/>
69                      </goal>
70                      <goal name="M_2" contents="NA" relationship=""
                            type="REQ">
71                          <agent name="M_2_Agent" contents="Battery
                                Charge" location="environment"/>
72                      </goal>
73                      <goal name="M_3" contents="(= RF 0.0)"
                            relationship="" type="POST">
74                          <agent name="M_3_Agent" contents="Regen
                                Force" location="environment"/>
75                      </goal>
76
77                  </goal>
78              </goal>
79
80              <goal name="NOT_PT" contents="(&lt; BCHARGE 1.0)"
                    relationship="" type="PRE">
81                  <agent name="NOT_PT_Agent" contents="Battery Charge
                        " location="environment"/>
82              </goal>
83
84          </goal>
85
86              <goal name="A" contents="Maintain(Brake System)"
                    relationship="" type="AND">
87              <goal name="A_1" contents="Maintain(Brake Force)"
                    relationship="" type="OR">
88                  <goal name="B" contents="Achieve(Standard Force
                        Braking)" relationship="" type="AND">
89                      <goal name="B_1" contents="(= SF CBF_t_plus_one
                            )" relationship="" type="POST">
```

```
90                      <agent name="B_1_Agent" contents="Hydraulic
                            Brake Sensor" location="environment"/>
91                  </goal>
92                  <goal name="B_2" contents="Achieve(Apply
                        Standard Force)" relationship="" type="REQ">
93                      <agent name="B_2_Agent" contents="Hydraulic
                            Brake Actuator" location="system"/>
94                  </goal>
95                  <goal name="B_3" contents="(&gt; CBF_t_plus_one
                        0.0)" relationship="" type="PRE">
96                      <agent name="B_3_Agent" contents="CBF Value
                            " location="environment"/>
97                  </goal>
98              </goal>
99              <goal name="C" contents="Achieve(Regen Braking)"
                    relationship="" type="AND">
100                 <goal name="C_1" contents="(&gt; CBF_t_plus_one
                        0.0)" relationship="" type="PRE">
101                     <agent name="C_1_Agent" contents="CBF Value
                            " location="environment"/>
102                 </goal>
103                 <goal name="C_2" contents="Achieve(Regen and
                        Standard Force)" relationship="" type="OR">
104                     <goal name="C_3" contents="Achieve(Standard
                            Force Braking)" relationship="" type="
                            AND">
105                         <goal name="C_5" contents="(= SF (*
                                CBF_t_plus_one 0.2))" relationship
                                ="" type="POST">
106                             <agent name="C_5_Agent" contents="
                                    Hydraulic Brake Sensor" location
                                    ="environment"/>
107                         </goal>
108                         <goal name="C_6" contents="Achieve(
                                Apply Standard Force)" relationship
                                ="" type="REQ">
109                             <agent name="C_6_Agent" contents="
                                    Hydraulic Brake Actuator"
                                    location="system"/>
110                         </goal>
111                         <goal name="C_7" contents="(&gt;
                                CBF_t_plus_one 0.2)" relationship=""
                                 type="PRE">
112                             <agent name="C_7_Agent" contents="
                                    CBF Value" location="environment
                                    "/>
113                         </goal>
```

```
114
115                         </goal>
116                         <goal name="C_4" contents="Achieve(Regen
                                Force)" relationship="" type="AND">
117                             <goal name="C_8" contents="(&lt;
                                    CBF_t_plus_one 0.5)" relationship=""
                                     type="PRE">
118                                 <agent name="C_7_Agent" contents="
                                        CBF Value" location="environment
                                        "/>
119                             </goal>
120                             <goal name="C_9" contents="Achieve(
                                    Apply Regen Force)" relationship=""
                                    type="REQ">
121                                 <agent name="C_6_Agent" contents="
                                        Regeneration Brake Actuator"
                                        location="system"/>
122                             </goal>
123                             <goal name="C_10" contents="(= RF (*
                                    CBF_t_plus_one 0.8))" relationship
                                    ="" type="POST">
124                                 <agent name="C_5_Agent" contents="
                                        Regeneration Brake Sensor"
                                        location="environment"/>
125                             </goal>
126                         </goal>
127                     </goal>
128                 </goal>
129             </goal>
130         <goal name="A_2" contents="Maintain(Brake Command)"
                relationship="" type="OR">
131             <goal name="D" contents="Achieve(Brake-by-Wire)"
                    relationship="" type="AND">
132                 <goal name="D_2" contents="Achieve(Read CBF)"
                        relationship="" type="REQ">
133                     <agent name="D_2_Agent" contents="Memory (
                            CBF)" location="system"/>
134                 </goal>
135                 <goal name="D_3" contents="(&gt; BF 0.0)"
                        relationship="" type="PRE">
136                     <agent name="D_3_Agent" contents="BF Value"
                             location="environment"/>
137                 </goal>
138                 <goal name="D_4" contents="Achieve(Read Brake
                        Force)" relationship="" type="REQ">
139                     <agent name="D_4_Agent" contents="Brake
                            Pedal Sensor (BF)" location="system"/>
```

```
140                     </goal>
141                     <goal name="D_5" contents="Achieve(Brake Pulse)
                            " relationship="" type="OR">
142                         <goal name="D_8" contents="Achieve(Brake On
                                )" relationship="" type="AND">
143                             <goal name="D_10" contents="(= CBF_t
                                    0.0)" relationship="" type="PRE">
144                                 <agent name="D_10_Agent" contents="
                                        CBF Value" location="environment
                                        "/>
145                             </goal>
146                             <goal name="D_11" contents="(=
                                    CBF_t_plus_one BF)" relationship=""
                                    type="POST">
147                                 <agent name="D_11_Agent" contents="
                                        Brake Pedal Sensor (BF)"
                                        location="environment"/>
148                             </goal>
149                             <goal name="D_12a" contents="Achieve(
                                    Brake Force Change)" relationship=""
                                     type="REQ">
150                                 <agent name="D_12a_Agent" contents
                                        ="Memory (CBF_t_plus_one)"
                                        location="system"/>
151                             </goal>
152                         </goal>
153                         <goal name="D_9" contents="Achieve(Brake
                                Off)" relationship="" type="AND">
154                             <goal name="D_12b" contents="Achieve(
                                    Brake Force Change)" relationship=""
                                     type="REQ">
155                                 <agent name="D_12b_Agent" contents
                                        ="Memory (CBF_t_plus_one)"
                                        location="system"/>
156                             </goal>
157                             <goal name="D_13" contents="(=
                                    CBF_t_plus_one 0.0)" relationship=""
                                     type="POST">
158                                 <agent name="D_13_Agent" contents="
                                        CBF_t_plus_one Value" location="
                                        environment"/>
159                             </goal>
160                             <goal name="D_14" contents="(&amp;gt; CBF_t
                                     0.0)" relationship="" type="PRE">
161                                 <agent name="D_14_Agent" contents="
                                        CBF Value" location="environment
                                        "/>
```

```
162                         </goal>
163                     </goal>
164                 </goal>
165                 <goal name="D_6" contents="(= SS 1.0)"
                        relationship="" type="PRE">
166                     <agent name="D_6_Agent" contents="SS Value"
                            location="environment"/>
167                 </goal>
168                 <goal name="D_7" contents="Achieve(Read SS)"
                        relationship="" type="REQ">
169                     <agent name="D_7_Agent" contents="Slip
                            Sensor" location="system"/>
170                 </goal>
171             </goal>
172             <goal name="E" contents="Achieve(Anti-Lock Braking)
                    " relationship="" type="AND">
173                 <goal name="E_2" contents="Achieve(Read CBF)"
                        relationship="" type="REQ">
174                     <agent name="E_2_Agent" contents="Memory (
                            CBF)" location="system"/>
175                 </goal>
176                 <goal name="E_3" contents="(= CBF_t_plus_one BF
                        )" relationship="" type="POST">
177                     <agent name="E_3_Agent" contents="BF Value"
                            location="environment"/>
178                 </goal>
179                 <goal name="E_4" contents="Achieve(Read Brake
                        Force)" relationship="" type="REQ">
180                     <agent name="E_4_Agent" contents="Brake
                            Pedal Sensor (BF)" location="system"/>
181                 </goal>
182                 <goal name="E_5" contents="(&amp;gt; BF 0.0)"
                        relationship="" type="PRE">
183                     <agent name="E_5_Agent" contents="BF Value"
                            location="environment"/>
184                 </goal>
185                 <goal name="E_6" contents="(&amp;lt; SS 1.0)"
                        relationship="" type="PRE">
186                     <agent name="E_6_Agent" contents="SS Value"
                            location="environment"/>
187                 </goal>
188                 <goal name="E_7" contents="Achieve(Read SS)"
                        relationship="" type="REQ">
189                     <agent name="E_7_Agent" contents="Slip
                            Sensor" location="system"/>
190                 </goal>
191             </goal>
```

```
192                 </goal>
193             </goal>
194             </goal>
195         </goal>
196
197         <environment/>
198 </goalmodel>
```

C.2 Run-Time Detection Code (C++)

This section of the appendix includes the C++ code output for detecting feature interactions at run time in Chapter 8. All four features in the braking system goal model are detected with this single code listing.

Listing C.3: Feature Interaction for Failure of Feature `B`, `C`, `D`, and `E`

```
1  bool WMP_fails;
2  double FDSensor;
3  bool RF_TMS_M_2;
4  bool C_precondition;
5  bool E_precondition;
6  bool BCHARGE_M_1;
7  bool WMP_precondition;
8  bool BCHARGE_NOT_PT_M;
9  bool FDSensor_TMS_NOT_PT;
10 double CBF_t_plus_one;
11 bool B_fails;
12 bool WMS_fails;
13 bool CBF_t_plus_one_C_8;
14 bool CBF_t_plus_one_C_7;
15 bool BCHARGE_TMP_PT_1;
16 bool SF_B_1;
17 bool CBF_t_plus_one_E_3;
18 bool CBF_t_plus_one_C_1;
19 bool C_fails;
20 double BCHARGE;
21 double CBF_t;
22 double RF;
23 bool B_precondition;
24 bool SM_precondition;
25 bool RF_TMP_M_2;
26 bool SS_E_6;
27 bool RF_M_3;
28 double BF;
29 bool BCHARGE_NOT_PT;
```

```
30 bool CBF_t_plus_one_D_11;
31 bool D_precondition;
32 bool SF_TMS_M_1;
33 bool RF_C_10;
34 bool CBF_t_plus_one_D_13;
35 bool CBF_t_D_14;
36 double SF;
37 bool topLevelSatisfation;
38 bool D_fails;
39 bool WMS_precondition;
40 bool SM_fails;
41 bool BF_D_3;
42 double SS;
43 bool SF_C_5;
44 bool CBF_t_plus_one_B_3;
45 bool BF_E_5;
46 bool E_fails;
47 bool CBF_t_D_10;
48 bool BCHARGE_TMP_NOT_PT_1;
49 bool FDSensor_TMS_PT_2;
50 bool BCHARGE_TMP_NOT_PT_2;
51 bool FDSensor_TMS_PT_1;
52 bool SS_D_6;
53 bool V_SS() {
54     return true;
55 }
56 bool V_SF() {
57     return true;
58 }
59 bool V_BCHARGE() {
60     return true;
61 }
62 bool V_CBF_t() {
63     return true;
64 }
65 bool V_RF() {
66     return true;
67 }
68 bool V_BF() {
69     return true;
70 }
71 bool V_FDSensor() {
72     return true;
73 }
74 bool V_CBF_t_plus_one() {
75     return true;
76 }
```

```
77 bool TMS_PT_1() {
78     return ((FDSensor < 0.0) && V_FDSensor());
79 }
80 bool TMS_PT_2() {
81     return ((std::abs(FDSensor - 0.0) < 0.0001) && V_FDSensor());
82 }
83 bool TMS_PT() {
84     return ((false || TMS_PT_1()) || TMS_PT_2());
85 }
86 bool TMS_M_1() {
87     return ((std::abs(SF - 1.0) < 0.0001) && V_SF());
88 }
89 bool TMS_M_2() {
90     return ((std::abs(RF - 0.0) < 0.0001) && V_RF());
91 }
92 bool TMS_M() {
93     return ((true && TMS_M_1()) && TMS_M_2());
94 }
95 bool TMS() {
96     return ((true && TMS_PT()) && TMS_M());
97 }
98 bool TMS_NOT_PT() {
99     return ((FDSensor > 0.0) && V_FDSensor());
100 }
101 bool WMS() {
102     return ((false || TMS()) || TMS_NOT_PT());
103 }
104 bool TMP_PT_1() {
105     return ((BCHARGE < 0.5) && V_BCHARGE());
106 }
107 bool TMP_M_1() {
108     return true;
109 }
110 bool TMP_M_2() {
111     return (((std::abs(RF - (CBF_t_plus_one * 1.0)) < 0.0001) &&
            V_RF()) && V_CBF_t_plus_one());
112 }
113 bool TMP_M() {
114     return ((true && TMP_M_1()) && TMP_M_2());
115 }
116 bool TMP() {
117     return ((true && TMP_PT_1()) && TMP_M());
118 }
119 bool TMP_NOT_PT_1() {
120     return ((BCHARGE > 0.5) && V_BCHARGE());
121 }
122 bool TMP_NOT_PT_2() {
```

```
123     return ((std::abs(BCHARGE - 0.5) < 0.0001) && V_BCHARGE());
124 }
125 bool TMP_NOT_PT() {
126     return ((true && TMP_NOT_PT_1()) && TMP_NOT_PT_2());
127 }
128 bool WMP() {
129     return ((false || TMP()) || TMP_NOT_PT());
130 }
131 bool WM() {
132     return ((true && WMS()) && WMP());
133 }
134 bool NOT_PT_M() {
135     return ((BCHARGE < 1.0) && V_BCHARGE());
136 }
137 bool M_1() {
138     return ((BCHARGE > 0.99) && V_BCHARGE());
139 }
140 bool M_2() {
141     return true;
142 }
143 bool M_3() {
144     return ((std::abs(RF - 0.0) < 0.0001) && V_RF());
145 }
146 bool M() {
147     return (((true && M_1()) && M_2()) && M_3());
148 }
149 bool TM_PT() {
150     return ((true && NOT_PT_M()) && M());
151 }
152 bool NOT_PT() {
153     return ((BCHARGE < 1.0) && V_BCHARGE());
154 }
155 bool SM() {
156     return ((false || TM_PT()) || NOT_PT());
157 }
158 bool B_1() {
159     return (((std::abs(SF - CBF_t_plus_one) < 0.0001) && V_SF()) &&
            V_CBF_t_plus_one());
160 }
161 bool B_2() {
162     return true;
163 }
164 bool B_3() {
165     return ((CBF_t_plus_one > 0.0) && V_CBF_t_plus_one());
166 }
167 bool B() {
168     return (((true && B_1()) && B_2()) && B_3());
```

```
169 }
170 bool C_1() {
171     return ((CBF_t_plus_one > 0.0) && V_CBF_t_plus_one());
172 }
173 bool C_5() {
174     return (((std::abs(SF - (CBF_t_plus_one * 0.2)) < 0.0001) &&
          V_SF()) && V_CBF_t_plus_one());
175 }
176 bool C_6() {
177     return true;
178 }
179 bool C_7() {
180     return ((CBF_t_plus_one > 0.2) && V_CBF_t_plus_one());
181 }
182 bool C_3() {
183     return (((true && C_5()) && C_6()) && C_7());
184 }
185 bool C_8() {
186     return ((CBF_t_plus_one < 0.5) && V_CBF_t_plus_one());
187 }
188 bool C_9() {
189     return true;
190 }
191 bool C_10() {
192     return (((std::abs(RF - (CBF_t_plus_one * 0.8)) < 0.0001) &&
          V_RF()) && V_CBF_t_plus_one());
193 }
194 bool C_4() {
195     return (((true && C_8()) && C_9()) && C_10());
196 }
197 bool C_2() {
198     return ((false || C_3()) || C_4());
199 }
200 bool C() {
201     return ((true && C_1()) && C_2());
202 }
203 bool A_1() {
204     return ((false || B()) || C());
205 }
206 bool D_2() {
207     return true;
208 }
209 bool D_3() {
210     return ((BF > 0.0) && V_BF());
211 }
212 bool D_4() {
213     return true;
```

```
214 }
215 bool D_10() {
216     return ((std::abs(CBF_t - 0.0) < 0.0001) && V_CBF_t());
217 }
218 bool D_11() {
219     return (((std::abs(CBF_t_plus_one - BF) < 0.0001) && V_BF()) &&
            V_CBF_t_plus_one());
220 }
221 bool D_12a() {
222     return true;
223 }
224 bool D_8() {
225     return (((true && D_10()) && D_11()) && D_12a());
226 }
227 bool D_12b() {
228     return true;
229 }
230 bool D_13() {
231     return ((std::abs(CBF_t_plus_one - 0.0) < 0.0001) &&
           V_CBF_t_plus_one());
232 }
233 bool D_14() {
234     return ((CBF_t > 0.0) && V_CBF_t());
235 }
236 bool D_9() {
237     return (((true && D_12b()) && D_13()) && D_14());
238 }
239 bool D_5() {
240     return ((false || D_8()) || D_9());
241 }
242 bool D_6() {
243     return ((std::abs(SS - 1.0) < 0.0001) && V_SS());
244 }
245 bool D_7() {
246     return true;
247 }
248 bool D() {
249     return ((((((true && D_2()) && D_3()) && D_4()) && D_5()) &&
           D_6()) && D_7());
250 }
251 bool E_2() {
252     return true;
253 }
254 bool E_3() {
255     return (((std::abs(CBF_t_plus_one - BF) < 0.0001) && V_BF()) &&
            V_CBF_t_plus_one());
256 }
```

```
257 bool E_4() {
258     return true;
259 }
260 bool E_5() {
261     return ((BF > 0.0) && V_BF());
262 }
263 bool E_6() {
264     return ((SS < 1.0) && V_SS());
265 }
266 bool E_7() {
267     return true;
268 }
269 bool E() {
270     return ((((((true && E_2()) && E_3()) && E_4()) && E_5()) &&
            E_6()) && E_7());
271 }
272 bool A_2() {
273     return ((false || D()) || E());
274 }
275 bool A() {
276     return ((true && A_1()) && A_2());
277 }
278 bool SMF() {
279     return ((true && SM()) && A());
280 }
281 bool AA() {
282     return ((true && WM()) && SMF());
283 }
284 bool TMS_PT_P() {
285     return ((false || TMS_PT_1()) || TMS_PT_2());
286 }
287 bool TMS_M_P() {
288     return true;
289 }
290 bool TMS_P() {
291     return ((true && TMS_PT_P()) && TMS_M_P());
292 }
293 bool WMS_P() {
294     return ((false || TMS_P()) || TMS_NOT_PT());
295 }
296 bool TMP_M_P() {
297     return true;
298 }
299 bool TMP_P() {
300     return ((true && TMP_PT_1()) && TMP_M_P());
301 }
302 bool TMP_NOT_PT_P() {
```

```
303     return true;
304 }
305 bool WMP_P() {
306     return ((false || TMP_P()) || TMP_NOT_PT_P());
307 }
308 bool WM_P() {
309     return ((true && WMS_P()) && WMP_P());
310 }
311 bool M_P() {
312     return (true && M_1());
313 }
314 bool TM_PT_P() {
315     return (true && M_P());
316 }
317 bool SM_P() {
318     return ((false || TM_PT_P()) || NOT_PT());
319 }
320 bool B_P() {
321     return (true && B_3());
322 }
323 bool C_3_P() {
324     return (true && C_7());
325 }
326 bool C_4_P() {
327     return (true && C_8());
328 }
329 bool C_2_P() {
330     return ((false || C_3_P()) || C_4_P());
331 }
332 bool C_P() {
333     return ((true && C_1()) && C_2_P());
334 }
335 bool A_1_P() {
336     return ((false || B_P()) || C_P());
337 }
338 bool D_8_P() {
339     return (true && D_10());
340 }
341 bool D_9_P() {
342     return (true && D_14());
343 }
344 bool D_5_P() {
345     return ((false || D_8_P()) || D_9_P());
346 }
347 bool D_P() {
348     return (((true && D_3()) && D_5_P()) && D_6());
349 }
```

```
350 bool E_P() {
351     return ((true && E_5()) && E_6());
352 }
353 bool A_2_P() {
354     return ((false || D_P()) || E_P());
355 }
356 bool A_P() {
357     return ((true && A_1_P()) && A_2_P());
358 }
359 bool SMF_P() {
360     return ((true && SM_P()) && A_P());
361 }
362 bool AA_P() {
363     return ((true && WM_P()) && SMF_P());
364 }
365 bool TMS_M_1_N() {
366     return ((!(std::abs(SF - 1.0) < 0.0001)) && V_SF());
367 }
368 bool TMS_M_2_N() {
369     return ((!(std::abs(RF - 0.0) < 0.0001)) && V_RF());
370 }
371 bool TMP_M_2_N() {
372     return (((!(std::abs(RF - (CBF_t_plus_one * 1.0)) < 0.0001)) &&
            V_RF()) && V_CBF_t_plus_one());
373 }
374 bool TMP_NOT_PT_1_N() {
375     return ((!(BCHARGE > 0.5)) && V_BCHARGE());
376 }
377 bool TMP_NOT_PT_2_N() {
378     return ((!(std::abs(BCHARGE - 0.5) < 0.0001)) && V_BCHARGE());
379 }
380 bool NOT_PT_M_N() {
381     return ((!(BCHARGE < 1.0)) && V_BCHARGE());
382 }
383 bool M_3_N() {
384     return ((!(std::abs(RF - 0.0) < 0.0001)) && V_RF());
385 }
386 bool B_1_N() {
387     return (((!(std::abs(SF - CBF_t_plus_one) < 0.0001)) && V_SF())
            && V_CBF_t_plus_one());
388 }
389 bool C_5_N() {
390     return (((!(std::abs(SF - (CBF_t_plus_one * 0.2)) < 0.0001)) &&
            V_SF()) && V_CBF_t_plus_one());
391 }
392 bool C_10_N() {
```

```
393     return (((!(std::abs(RF - (CBF_t_plus_one * 0.8)) < 0.0001)) &&
            V_RF()) && V_CBF_t_plus_one());
394 }
395 bool D_11_N() {
396     return (((!(std::abs(CBF_t_plus_one - BF) < 0.0001)) && V_BF())
            && V_CBF_t_plus_one());
397 }
398 bool D_13_N() {
399     return ((!(std::abs(CBF_t_plus_one - 0.0) < 0.0001)) &&
           V_CBF_t_plus_one());
400 }
401 bool E_3_N() {
402     return (((!(std::abs(CBF_t_plus_one - BF) < 0.0001)) && V_BF())
            && V_CBF_t_plus_one());
403 }
404 bool TMS_PT_PP() {
405     return ((false || TMS_PT_1()) || TMS_PT_2());
406 }
407 bool TMS_M_PP() {
408     return (true && ((false || TMS_M_1_N()) || TMS_M_2_N()));
409 }
410 bool TMS_PP() {
411     return ((true && TMS_PT_PP()) && TMS_M_PP());
412 }
413 bool WMS_PP() {
414     return ((false || TMS_PP()) || TMS_NOT_PT());
415 }
416 bool TMP_M_PP() {
417     return (true && (false || TMP_M_2_N()));
418 }
419 bool TMP_PP() {
420     return ((true && TMP_PT_1()) && TMP_M_PP());
421 }
422 bool TMP_NOT_PT_PP() {
423     return (true && ((false || TMP_NOT_PT_1_N()) || TMP_NOT_PT_2_N
           ()));
424 }
425 bool WMP_PP() {
426     return ((false || TMP_PP()) || TMP_NOT_PT_PP());
427 }
428 bool WM_PP() {
429     return ((true && WMS_PP()) && WMP_PP());
430 }
431 bool M_PP() {
432     return ((true && M_1()) && (false || M_3_N()));
433 }
434 bool TM_PT_PP() {
```

```
435     return ((true && M_PP()) && (false || NOT_PT_M_N()));
436 }
437 bool SM_PP() {
438     return ((false || TM_PT_PP()) || NOT_PT());
439 }
440 bool B_PP() {
441     return ((true && B_3()) && (false || B_1_N()));
442 }
443 bool C_3_PP() {
444     return ((true && C_7()) && (false || C_5_N()));
445 }
446 bool C_4_PP() {
447     return ((true && C_8()) && (false || C_10_N()));
448 }
449 bool C_2_PP() {
450     return ((false || C_3_PP()) || C_4_PP());
451 }
452 bool C_PP() {
453     return ((true && C_1()) && C_2_PP());
454 }
455 bool A_1_PP() {
456     return ((false || B_PP()) || C_PP());
457 }
458 bool D_8_PP() {
459     return ((true && D_10()) && (false || D_11_N()));
460 }
461 bool D_9_PP() {
462     return ((true && D_14()) && (false || D_13_N()));
463 }
464 bool D_5_PP() {
465     return ((false || D_8_PP()) || D_9_PP());
466 }
467 bool D_PP() {
468     return (((true && D_3()) && D_5_PP()) && D_6());
469 }
470 bool E_PP() {
471     return (((true && E_5()) && E_6()) && (false || E_3_N()));
472 }
473 bool A_2_PP() {
474     return ((false || D_PP()) || E_PP());
475 }
476 bool A_PP() {
477     return ((true && A_1_PP()) && A_2_PP());
478 }
479 bool SMF_PP() {
480     return ((true && SM_PP()) && A_PP());
481 }
```

```
482 bool AA_PP() {
483     return ((true && WM_PP()) && SMF_PP());
484 }
485 bool AA_Contributes() {
486     return AA();
487 }
488 bool WM_Contributes() {
489     return (WM() && AA_Contributes());
490 }
491 bool WMS_Contributes() {
492     return WMS();
493 }
494 bool TMS_Contributes() {
495     return (TMS() && WMS_Contributes());
496 }
497 bool TMS_PT_Contributes() {
498     return (TMS_PT() && TMS_Contributes());
499 }
500 bool TMS_PT_1_Contributes() {
501     return (TMS_PT_1() && TMS_PT_Contributes());
502 }
503 bool TMS_PT_2_Contributes() {
504     return (TMS_PT_2() && TMS_PT_Contributes());
505 }
506 bool TMS_M_Contributes() {
507     return (TMS_M() && TMS_Contributes());
508 }
509 bool TMS_M_1_Contributes() {
510     return (TMS_M_1() && TMS_M_Contributes());
511 }
512 bool TMS_M_2_Contributes() {
513     return (TMS_M_2() && TMS_M_Contributes());
514 }
515 bool TMS_NOT_PT_Contributes() {
516     return (TMS_NOT_PT() && WMS_Contributes());
517 }
518 bool WMP_Contributes() {
519     return WMP();
520 }
521 bool TMP_Contributes() {
522     return (TMP() && WMP_Contributes());
523 }
524 bool TMP_PT_1_Contributes() {
525     return (TMP_PT_1() && TMP_Contributes());
526 }
527 bool TMP_M_Contributes() {
528     return (TMP_M() && TMP_Contributes());
```

```
529 }
530 bool TMP_M_1_Contributes() {
531     return (TMP_M_1() && TMP_M_Contributes());
532 }
533 bool TMP_M_2_Contributes() {
534     return (TMP_M_2() && TMP_M_Contributes());
535 }
536 bool TMP_NOT_PT_Contributes() {
537     return (TMP_NOT_PT() && WMP_Contributes());
538 }
539 bool TMP_NOT_PT_1_Contributes() {
540     return (TMP_NOT_PT_1() && TMP_NOT_PT_Contributes());
541 }
542 bool TMP_NOT_PT_2_Contributes() {
543     return (TMP_NOT_PT_2() && TMP_NOT_PT_Contributes());
544 }
545 bool SMF_Contributes() {
546     return (SMF() && AA_Contributes());
547 }
548 bool SM_Contributes() {
549     return SM();
550 }
551 bool TM_PT_Contributes() {
552     return (TM_PT() && SM_Contributes());
553 }
554 bool NOT_PT_M_Contributes() {
555     return (NOT_PT_M() && TM_PT_Contributes());
556 }
557 bool M_Contributes() {
558     return (M() && TM_PT_Contributes());
559 }
560 bool M_1_Contributes() {
561     return (M_1() && M_Contributes());
562 }
563 bool M_2_Contributes() {
564     return (M_2() && M_Contributes());
565 }
566 bool M_3_Contributes() {
567     return (M_3() && M_Contributes());
568 }
569 bool NOT_PT_Contributes() {
570     return (NOT_PT() && SM_Contributes());
571 }
572 bool A_Contributes() {
573     return (A() && SMF_Contributes());
574 }
575 bool A_1_Contributes() {
```

```
576     return (A_1() && A_Contributes());
577 }
578 bool B_Contributes() {
579     return B();
580 }
581 bool B_1_Contributes() {
582     return (B_1() && B_Contributes());
583 }
584 bool B_2_Contributes() {
585     return (B_2() && B_Contributes());
586 }
587 bool B_3_Contributes() {
588     return (B_3() && B_Contributes());
589 }
590 bool C_Contributes() {
591     return C();
592 }
593 bool C_1_Contributes() {
594     return (C_1() && C_Contributes());
595 }
596 bool C_2_Contributes() {
597     return (C_2() && C_Contributes());
598 }
599 bool C_3_Contributes() {
600     return (C_3() && C_2_Contributes());
601 }
602 bool C_5_Contributes() {
603     return (C_5() && C_3_Contributes());
604 }
605 bool C_6_Contributes() {
606     return (C_6() && C_3_Contributes());
607 }
608 bool C_7_Contributes() {
609     return (C_7() && C_3_Contributes());
610 }
611 bool C_4_Contributes() {
612     return (C_4() && C_2_Contributes());
613 }
614 bool C_8_Contributes() {
615     return (C_8() && C_4_Contributes());
616 }
617 bool C_9_Contributes() {
618     return (C_9() && C_4_Contributes());
619 }
620 bool C_10_Contributes() {
621     return (C_10() && C_4_Contributes());
622 }
```

```
623 bool A_2_Contributes() {
624     return (A_2() && A_Contributes());
625 }
626 bool D_Contributes() {
627     return D();
628 }
629 bool D_2_Contributes() {
630     return (D_2() && D_Contributes());
631 }
632 bool D_3_Contributes() {
633     return (D_3() && D_Contributes());
634 }
635 bool D_4_Contributes() {
636     return (D_4() && D_Contributes());
637 }
638 bool D_5_Contributes() {
639     return (D_5() && D_Contributes());
640 }
641 bool D_8_Contributes() {
642     return (D_8() && D_5_Contributes());
643 }
644 bool D_10_Contributes() {
645     return (D_10() && D_8_Contributes());
646 }
647 bool D_11_Contributes() {
648     return (D_11() && D_8_Contributes());
649 }
650 bool D_12a_Contributes() {
651     return (D_12a() && D_8_Contributes());
652 }
653 bool D_9_Contributes() {
654     return (D_9() && D_5_Contributes());
655 }
656 bool D_12b_Contributes() {
657     return (D_12b() && D_9_Contributes());
658 }
659 bool D_13_Contributes() {
660     return (D_13() && D_9_Contributes());
661 }
662 bool D_14_Contributes() {
663     return (D_14() && D_9_Contributes());
664 }
665 bool D_6_Contributes() {
666     return (D_6() && D_Contributes());
667 }
668 bool D_7_Contributes() {
669     return (D_7() && D_Contributes());
```

```
670 }
671 bool E_Contributes() {
672     return E();
673 }
674 bool E_2_Contributes() {
675     return (E_2() && E_Contributes());
676 }
677 bool E_3_Contributes() {
678     return (E_3() && E_Contributes());
679 }
680 bool E_4_Contributes() {
681     return (E_4() && E_Contributes());
682 }
683 bool E_5_Contributes() {
684     return (E_5() && E_Contributes());
685 }
686 bool E_6_Contributes() {
687     return (E_6() && E_Contributes());
688 }
689 bool E_7_Contributes() {
690     return (E_7() && E_Contributes());
691 }
692 void updateAssertions() {
693     (TMS_PT_1_Contributes() ? (FDSensor_TMS_PT_1 = true):(
        FDSensor_TMS_PT_1 = false));
694     (TMS_PT_2_Contributes() ? (FDSensor_TMS_PT_2 = true):(
        FDSensor_TMS_PT_2 = false));
695     (TMS_M_1_Contributes() ? (SF_TMS_M_1 = true):(SF_TMS_M_1 =
        false));
696     (TMS_M_2_Contributes() ? (RF_TMS_M_2 = true):(RF_TMS_M_2 =
        false));
697     (TMS_NOT_PT_Contributes() ? (FDSensor_TMS_NOT_PT = true):(
        FDSensor_TMS_NOT_PT = false));
698     (TMP_PT_1_Contributes() ? (BCHARGE_TMP_PT_1 = true):(
        BCHARGE_TMP_PT_1 = false));
699     (TMP_M_2_Contributes() ? (RF_TMP_M_2 = true):(RF_TMP_M_2 =
        false));
700     (TMP_NOT_PT_1_Contributes() ? (BCHARGE_TMP_NOT_PT_1 = true):(
        BCHARGE_TMP_NOT_PT_1 = false));
701     (TMP_NOT_PT_2_Contributes() ? (BCHARGE_TMP_NOT_PT_2 = true):(
        BCHARGE_TMP_NOT_PT_2 = false));
702     (NOT_PT_M_Contributes() ? (BCHARGE_NOT_PT_M = true):(
        BCHARGE_NOT_PT_M = false));
703     (M_1_Contributes() ? (BCHARGE_M_1 = true):(BCHARGE_M_1 = false)
        );
704     (M_3_Contributes() ? (RF_M_3 = true):(RF_M_3 = false));
```

```
705     (NOT_PT_Contributes() ? (BCHARGE_NOT_PT = true):(BCHARGE_NOT_PT
            = false));
706     (B_1_Contributes() ? (SF_B_1 = true):(SF_B_1 = false));
707     (B_3_Contributes() ? (CBF_t_plus_one_B_3 = true):(
            CBF_t_plus_one_B_3 = false));
708     (C_1_Contributes() ? (CBF_t_plus_one_C_1 = true):(
            CBF_t_plus_one_C_1 = false));
709     (C_5_Contributes() ? (SF_C_5 = true):(SF_C_5 = false));
710     (C_7_Contributes() ? (CBF_t_plus_one_C_7 = true):(
            CBF_t_plus_one_C_7 = false));
711     (C_8_Contributes() ? (CBF_t_plus_one_C_8 = true):(
            CBF_t_plus_one_C_8 = false));
712     (C_10_Contributes() ? (RF_C_10 = true):(RF_C_10 = false));
713     (D_3_Contributes() ? (BF_D_3 = true):(BF_D_3 = false));
714     (D_10_Contributes() ? (CBF_t_D_10 = true):(CBF_t_D_10 = false))
            ;
715     (D_11_Contributes() ? (CBF_t_plus_one_D_11 = true):(
            CBF_t_plus_one_D_11 = false));
716     (D_13_Contributes() ? (CBF_t_plus_one_D_13 = true):(
            CBF_t_plus_one_D_13 = false));
717     (D_14_Contributes() ? (CBF_t_D_14 = true):(CBF_t_D_14 = false))
            ;
718     (D_6_Contributes() ? (SS_D_6 = true):(SS_D_6 = false));
719     (E_3_Contributes() ? (CBF_t_plus_one_E_3 = true):(
            CBF_t_plus_one_E_3 = false));
720     (E_5_Contributes() ? (BF_E_5 = true):(BF_E_5 = false));
721     (E_6_Contributes() ? (SS_E_6 = true):(SS_E_6 = false));
722     (topLevelSatisfation = ((WMS() && WMP()) && (SM() && ((B() || C
            ()) && (D() || E()))))));
723     (B_precondition = B_P());
724     (C_precondition = C_P());
725     (D_precondition = D_P());
726     (E_precondition = E_P());
727     (SM_precondition = SM_P());
728     (WMP_precondition = WMP_P());
729     (WMS_precondition = WMS_P());
730     (B_fails = B_PP());
731     (C_fails = C_PP());
732     (D_fails = D_PP());
733     (E_fails = E_PP());
734     (SM_fails = SM_PP());
735     (WMP_fails = WMP_PP());
736     (WMS_fails = WMS_PP());
737 }
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Requirements-driven deployment - customizing the requirements model for the host environment. *Software and System Modeling*, 13(1):433–456, 2014.

[2] Dalal Alrajeh, Jeff Kramer, Axel van Lamsweerde, Alessandra Russo, and Sebastian Uchitel. Generating obstacle conditions for requirements completeness. In *Proceedings of the 34th International Conference on Software Engineering*, pages 705–715. IEEE Press, 2012.

[3] Zaid Altahat, Tzilla Elrad, Luay Tahat, and Nada Almasri. Detection of syntactic aspect interaction in UML state diagrams using critical pair analysis in graph transformation. *arXiv preprint arXiv:1312.6939*, 2013.

[4] Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave. Feature Interactions: The Next Generation (Dagstuhl Seminar 14281). *Dagstuhl Reports*, 4(7):1–24, 2014.

[5] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, pages 1–8. ACM, 2013.

[6] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander Von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 372–375. IEEE Computer Society, 2011.

[7] Sven Apel, Alexander Von Rhein, Thomas Thüm, and Christian Kästner. Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399–2409, 2013.

[8] João Araújo, Ana Moreira, Isabel Brito, and Awais Rashid. Aspect-oriented requirements with UML. In *Workshop on Aspect-oriented Modeling with UML*, volume 7. Citeseer, 2002.

[9] Carlos Areces, Wiet Bouma, and Maarten de Rijke. Feature interaction as a satisfiability problem, 2000.

[10] Silky Arora, Prahladavaradan Sampath, and S Ramesh. Resolving uncertainty in automotive feature interactions. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 21–30. IEEE, 2012.

[11] Elisa Baniassad, Paul C Clements, and João Araújo. Discovering early aspects.

[12] C. Barrett, A. Stump, and C. Tinelli. The *smt-lib* Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, 2010.

[13] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[14] Nikolaj Bjørner and Anh-Dung Phan. νz-maximal satisfaction with z3. In *SCSS*, pages 1–9, 2014.

[15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νz-an optimizing smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015.

[16] Johan Blom, Roland N Bol, and Lars Kempe. Automatic detection of feature interactions in temporal logic. In *FIW*, pages 1–19, 1995.

[17] Cecylia Bocovich and Joanne M. Atlee. Variable-specific resolutions for feature interactions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 553–563, 2014.

[18] Thomas F Bowen, FS Dworack, Ching-Hua Chow, Nancy Griffeth, Gary E Herman, and Y-J Lin. The feature interaction problem in telecommunications systems. In *Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89., Seventh International Conference on*, pages 59–62. IET, 1989.

[19] Isabel Sofia Brito, Ana Moreira, Rita A Ribeiro, and João Araújo. Handling conflicts in aspect-oriented requirements engineering. In *Aspect-Oriented Requirements Engineering*, pages 225–241. Springer, 2013.

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

[21] Muffy Calder and Alice Miller. Feature interaction detection by pairwise analysis of ltl properties-a case study. *Formal Methods in System Design*, 28(3):213–261, 2006.

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

[23] Betty H. C. Cheng and Joanne M Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, 2007.

[24] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Model Driven Engineering Languages and Systems*, pages 468–483. Springer, 2009.

[25] Ruzanna Chitchyan, Awais Rashid, Pete Sawyer, Alessandro Garcia, M Pinto Alarcon, Jethro Bakker, Bedir Tekinerdogan, Siobhán Clarke, and Andrew Jackson. Survey of aspect-oriented analysis and design approaches. 2015.

[26] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 321–330. ACM, 2011.

[27] David R Cok. The *SMT-LIBv2* language and tools: A tutorial. *Language C*, pages 2010–2011, 2011.

[28] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[29] Michael A Cusumano. Reflections on the toyota debacle. *Communications of the ACM*, 54(1):33–35, 2011.

[30] Fabiano Dalpiaz, Alexander Borgida, Jennifer Horkoff, and John Mylopoulos. Runtime goal models: Keynote. In *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*, pages 1–11. IEEE, 2013.

[31] Robert Darimont and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 179–190. ACM, 1996.

[32] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient *SMT* solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[33] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[34] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. Wiley, 2005.

[35] Kalyanmoy Deb and Ram B Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9(3):1–15, 1994.

[36] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[37] Byron DeVries and Betty H. C. Cheng. Automatic detection of incomplete requirements via symbolic analysis. In *19th International Conference on Model Driven Engineering Languages and Systems, MODELS 2016, Proceedings, Saint-Malo, France, October 2-7*, pages 385–395. ACM, 2016.

[38] Byron DeVries and Betty H. C. Cheng. Automatic detection of incomplete requirements using symbolic analysis and evolutionary computation. In *International Symposium on Search Based Software Engineering*, pages 49–64. Springer, 2017.

[39] Byron DeVries and Betty H. C. Cheng. Using models at run time to detect incomplete and inconsistent requirements. In *Proceedings of the 21th International Workshop on Models@run.time co-located with 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, Texas, September 18*, 2017.

[40] Byron DeVries and Betty H. C. Cheng. Automatic detection of feature interactions using symbolic analysis and evolutionary computation (in review). In *Submitted for Publication*, 2018.

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

[42] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *International Conference on Generative Programming and Component Engineering*, pages 173–188. Springer, 2002.

[43] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[44] Alessandro Fantechi, Stefania Gnesi, and Laura Semini. Optimizing feature interaction detection. In *Critical Systems: Formal Methods and Automated Verification*, pages 201–216. Springer, 2017.

[45] Martin S Feather, Stephen Fickas, Axel van Lamsweerde, and Cristophe Ponsard. Reconciling system requirements and runtime behavior. In *Proceedings of the 9th international workshop on Software specification and design*, page 50. IEEE Computer Society, 1998.

[46] Alessio Ferrari, Felice dell'Orletta, Giorgio Oronzo Spagnolo, and Stefania Gnesi. Measuring and improving the completeness of natural language requirements. In *Requirements Engineering: Foundation for Software Quality*, pages 23–38. Springer, 2014.

[47] Stephen Fickas and Martin S Feather. Requirements monitoring in dynamic environments. In *Requirements Engineering, Proceedings of the Second IEEE International Symposium on*, pages 140–147. IEEE, 1995.

[48] Erik M Fredericks, Byron DeVries, and Betty H. C. Cheng. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 17–26. ACM, 2014.

[49] Erik M Fredericks, Byron DeVries, and Betty HC Cheng. Autorelax: automatically relaxing a goal model to address uncertainty. *Empirical Software Engineering*, 19(5):1466–1501, 2014.

[50] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[51] DE Goldberg. Genetic algorithms in search, optimization, and machine learning, addison-wesley, reading, ma, 1989.

[52] Heather J Goldsby and Betty H. C. Cheng. Automatically discovering properties that specify the latent behavior of UML models. In *Model Driven Engineering Languages and Systems*, pages 316–330. Springer, 2010.

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

[54] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.

[55] Jonathan D. Hay and Joanne M. Atlee. Composing features and resolving interactions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 110–119, New York, NY, USA, 2000. ACM.

[56] Mats PE Heimdahl and Nancy G Leveson. Completeness and consistency in hierarchical state-based requirements. *Software Engineering, IEEE Transactions on*, 22(6):363–377, 1996.

[57] Russell Impagliazzo and Ramamohan Paturi. Complexity of k-sat. In *Computational Complexity, 1999. Proceedings. Fourteenth Annual IEEE Conference on*, pages 237–240. IEEE, 1999.

[58] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng.*, 24(10):831–847, October 1998.

[59] Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *Model Driven Engineering Languages and Systems*, pages 151–165. Springer, 2007.

[60] Adam C Jensen, Betty H. C. Cheng, Heather J Goldsby, and Edward C Nelson. A toolchain for the detection of structural and behavioral latent system properties. In *Model Driven Engineering Languages and Systems*, pages 683–698. Springer, 2011.

[61] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien. A technique for agile and automatic interaction testing for product lines. In *Testing Software and Systems*, pages 39–54. Springer, 2012.

[62] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[63] Phil Koopman. A case study of *Toyota* unintended acceleration and software safety. *Presentation. Sept*, 2014.

[64] Alexei Lapouchnian and John Mylopoulos. Modeling domain variability in requirements engineering with contexts. In *International Conference on Conceptual Modeling*, pages 115–130. Springer, 2009.

[65] Edward A Lee. Cyber physical systems: Design challenges. In *Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on*, pages 363–369. IEEE, 2008.

[66] Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

[67] Nancy Leveson. Completeness in formal specification language design for process-control systems. In *Proceedings of the Third Workshop on Formal methods in software practice*, pages 75–87. ACM, 2000.

[68] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240. Carnegie Mellon University, 2009.

[69] Igor Menzel, Mark Mueller, Anne Gross, and Joerg Doerr. An experimental comparison regarding the completeness of functional requirements specifications. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 15–24. IEEE, 2010.

[70] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[71] Gunter Mussbacher, Jon Whittle, and Daniel Amyot. Towards semantic-based aspect interaction detection. *1st Intl. Wksh. on Non-functional System Properties in Domain Specific Modeling Languages (NFPinDSML2008)*, 2008.

[72] Gunter Mussbacher, Jon Whittle, and Daniel Amyot. Semantic-based interaction detection in aspect-oriented scenarios. In *2009 17th IEEE International Requirements Engineering Conference*, pages 203–212. IEEE, 2009.

[73] Gunter Mussbacher, Jon Whittle, and Daniel Amyot. Modeling and detecting semantic-based interactions in aspect-oriented scenarios. *Requirements Engineering*, 15(2):197–214, 2010.

[74] M Nakamura and S Reiff-Marganiec. Semantic-based aspect interaction detection with goal models (position paper). *Feature Interactions in Software and Communication Systems X*, page 176, 2009.

[75] Andres J Ramirez and Betty H. C. Cheng. Automatic derivation of utility functions for monitoring software requirements. In *Model Driven Engineering Languages and Systems*, pages 501–516. Springer, 2011.

[76] Silvio Ranise and Cesare Tinelli. The *SMT-LIB* standard: Version 1.2. Technical report, Technical report, Department of Computer Science, The University of Iowa, 2006. Available at www. SMT-LIB. org, 2006.

[77] Awais Rashid, Peter Sawyer, Ana Moreira, and João Araújo. Early aspects: A model for aspect-oriented requirements engineering. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 199–202. IEEE, 2002.

[78] William N Robinson. Monitoring software requirements using instrumented code. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 3967–3976. IEEE, 2002.

[79] Américo Sampaio, Ruzanna Chitchyan, Awais Rashid, and Paul Rayson. Ea-miner: a tool for automating aspect-oriented requirements identification. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 352–355. ACM, 2005.

[80] Alberto Sardinha, Ruzanna Chitchyan, João Araújo, Ana Moreira, and Awais Rashid. Conflict identification with ea-analyzer. In *Aspect-Oriented Requirements Engineering*, pages 209–224. Springer, 2013.

[81] Vítor E Silva Souza, Alexei Lapouchnian, William N Robinson, and John Mylopoulos. Awareness requirements for adaptive systems. In *Proceedings of the 6th international symposium on Software engineering for adaptive and self-managing systems*, pages 60–69. ACM, 2011.

[82] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In *International Symposium on Formal Methods*, pages 532–546. Springer, 2009.

[83] Vítor E Silva Souza, Alexei Lapouchnian, William N Robinson, and John Mylopoulos. Awareness requirements. In *Software Engineering for Self-Adaptive Systems II*, pages 133–161. Springer, 2013.

[84] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104. ACM, 2007.

[85] Sebastian Uchitel and Marsha Chechik. Merging partial behavioural models. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 43–52. ACM, 2004.

[86] Rob van der Linden. Using an architecture to help beat feature interaction. In *FIW*, pages 24–35, 1994.

[87] Axel van Lamsweerde et al. *Requirements engineering: from system goals to UML models to software specifications*. Wiley, 2009.

[88] John Viega and Jeffrey Voas. Can aspect-oriented programming lead to more reliable software? *IEEE Software*, 17(6):19, 2000.

[89] William E Walsh, Gerald Tesauro, Jeffrey O Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *Proceedings of the International Conference on Autonomic Computing*, pages 70–77. IEEE, 2004.

[90] Nathan Weston, Francois Taiani, and Awais Rashid. Interaction analysis for fault-tolerance in aspect-oriented programming. 2007.

[91] Jon Whittle and Praveen Jayaraman. Mata: A tool for aspect-oriented modeling based on graph transformation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 16–27. Springer, 2007.

[92] Jon Whittle, Pete Sawyer, Nelly Bencomo, and Betty H. C. Cheng. A language for self-adaptive system requirements. In *Service-Oriented Computing: Consequences for Engineering Requirements, 2008. SOCCER'08. International Workshop on*, pages 24–29. IEEE, 2008.

[93] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. Relax: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 15(2):177–196, 2010.

[94] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty HC Cheng, and Jean-Michel Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 79–88. IEEE, 2009.

[95] Eric Yu. Modelling strategic relationships for process reengineering. *Social Modeling for Requirements Engineering*, 11:2011, 2011.

[96] Eric SK Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*, pages 226–235. IEEE, 1997.

[97] Yijun Yu, Julio CSP Leite, and John Mylopoulos. From goals to aspects: discovering aspects from requirements goal models. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 38–47. IEEE, 2004.

[98] Marina MN Zenun and Geilson Loureiro. A framework for dependability and completeness in requirements engineering. In *Latin American Symposium on Dependable Computing*, pages 1–4, 2013.

[99] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work and challenges. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 88–94, 2008.

[100] M Hadi Zibaeenejad, Chi Zhang, and Joanne M Atlee. Continuous variable-specific resolutions of feature interactions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 408–418. ACM, 2017.