Automatic Detection of Incomplete Requirements via Symbolic Analysis

Byron DeVries
devri117@cse.msu.edu

Betty H. C. Cheng
chengb@cse.msu.edu

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI, 48823, USA

ABSTRACT

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 paper 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.

CCS Concepts

•Software and its engineering → Requirements analysis; *Software verification and validation*;

Keywords

requirements, requirements completeness, symbolic analysis

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 paper 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, 6, 13, 19, 29]. 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 [9], 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 paper describes *Ares*, a symbolic analysis approach to automatically identify 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MODELS '16, October 02 - 07, 2016, Saint-Malo, France

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4321-3/16/10...\$15.00

DOI: <http://dx.doi.org/10.1145/2976767.2976791>

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

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 [20] 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 [20]. 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, if there are any environmental conditions 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 automotive application, and incomplete requirements decompositions are summarized and presented to the system designer.

The contributions of this paper are as follows:

- We introduce 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 paper is organized into the following sections. Section 2 provides an overview of background information. Section 3 details the approach. Section 4 describes the results of a case study, and Section 5 details related work. Finally, Section 6 discusses the conclusions and avenues of future work.

2. BACKGROUND

This section provides background on the topics of hierarchical requirements modeling, utility functions, SMT solvers, and an Adaptive Cruise Control (ACC) system provided by our industrial collaborators.

2.1 Hierarchical Requirements Modeling

While the concepts of this paper are applicable to any hierarchical requirements modeling framework, including i^* [28], KAOS goal modeling [24], or simply hierarchical requirements modeling [22], the case studies in this paper make use of the KAOS goal modeling notation.² 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 of the environment. Decomposition within KAOS goal modeling allows for both AND and OR decompositions [24]. 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 [24].

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 [15]. Since *Ares* is 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, but which are unsatisfied for the high-level objectives of the system-to-be [2]. An individual requirement is completely decomposed when the aggregate decomposed requirements 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 in the following expression [24]:

$$\{R_1, R_2, R_3, R_{...}, R_n, Dom\} \models R. \quad (1)$$

If a requirement cannot be discharged or handled by a single agent, then it must be decomposed into additional aggregate requirements that indicate *how* more specific requirements can be used 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 paper, 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 [25] have been used as run-time moni-

²For this work we do not use the KAOS formal decomposition patterns.

tors [12, 14] 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* [26]. Previously, Athena [20] 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. 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 [7]. 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 *Ares* can be used with no additional documentation required.

2.3 Satisfiability Modulo Theories Solvers

Satisfiability Modulo Theories (SMT) solvers solve constraints defined in either SMT-LIB [21] or SMT-LIB version 2 [3] standards [8]. 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 [11]. *Ares* uses the Microsoft Z3 SMT Solver [10].

2.4 Adaptive Cruise Control Systems

The Adaptive Cruise Control (ACC) 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 1 is a KAOS goal model for the ACC system. Keywords *Maintain* and *Achieve* are shortened to *M* and *A*, respectively. Table 1 identifies the agents referenced in Figure 1. The ACC model defined here was 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.³

In Figure 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

Table 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

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 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 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 1 are defined in Table 3. 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**,

³Conversations with automotive industrial practitioners included reviewing our goal modeling approach to automotive systems, including review of the specific ACC requirements model included in this paper, over the course of several meetings.

Table 3: ENV, MON, and REL Properties

	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

ment (R), assuming the domain properties and assumptions (Dom) hold. When using utility functions to assess satisficement, Boolean logic is not sufficient due to 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 [20]. The measure of decomposition completeness for assessing satisficement for Expression (2) (i.e., AND decomposition) is captured in Expression (4):

$$\begin{aligned} \min(&Satisficement(R_1), \\ &Satisficement(R_2), \\ &Satisficement(R_3), \\ &Satisficement(R_{...}), \\ &Satisficement(R_n)) \implies Satisficement(R), \end{aligned} \quad (4)$$

where the *Satisficement* function is the corresponding utility function for the respective requirement parameter. For example, *Satisficement*(**B.2**) is the **REL** expression from row **B.2** in Table 3. Similarly, the measure of decomposition completeness for assessing satisficement for Expression (3) (i.e., OR decomposition) is captured in Expression (5):

$$\begin{aligned} \max(&Satisficement(R_1), \\ &Satisficement(R_2), \\ &Satisficement(R_3), \\ &Satisficement(R_{...}), \\ &Satisficement(R_n)) \implies Satisficement(R) \end{aligned} \quad (5)$$

In both Expressions (4) and (5), 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. 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.

The remainder of this section describes the *Ares* process for detecting incomplete requirements decompositions by finding counterexamples to Expressions (4) and (5).

3.1 Ares Process

Figure 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 [20] to generate utility functions from a goal model (e.g., Figure 1) and **ENV**, **MON**, and **REL** properties (e.g., Table 3). Step 1 of Figure 2 accepts a goal model and the Athena-derived utility functions to produce a set of satisficement relationships for all the hierarchically decomposed requirements. Step 2 applies the symbolic analysis to the satisficement expressions to identify decomposition counterexamples.

3.1.1 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’

Figure 2: Ares Data Flow Diagram

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 (6) based on the **REL** property specified by the requirements engineer in Table 3 and taken from the generated utility function.

$$Satisficement(\mathbf{B.4}) = Throttle\ Actuator < Throttle\ Pedal\ Sensor \quad (6)$$

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 (5)). For example, the requirements OR-decomposed from **B.8** are:

$$Satisficement_{Decomp}(\mathbf{B.8}) = \max(Satisficement(\mathbf{B.9}), Satisficement(\mathbf{B.10})). \quad (7)$$

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 (4)). For example, the satisficement of requirements AND-decomposed from **B.4** are:

$$Satisficement_{Decomp}(\mathbf{B.4}) = \min(Satisficement(\mathbf{B.5}), Satisficement(\mathbf{B.6}), Satisficement(\mathbf{B.7})). \quad (8)$$

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 (9) is used to determine if goal **B.4** has a completeness counterexample.

$$\neg(Satisficement_{Decomp}(\mathbf{B.4}) \implies Satisficement(\mathbf{B.4})). \quad (9)$$

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 completeness counterexample (in case there is more than one completeness counterexample for a given decomposed goal). For example, maximizing Expression (10) identifies the most significant completeness counterexample for the decomposition of **B.4**:

$$\text{Satisficement}_{\text{Decomp}}(\mathbf{B.4}) - \text{Satisficement}(\mathbf{B.4}). \quad (10)$$

Identifying the maximum of Expression (10) yields counterexamples to Expressions (4) and (5), 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 (10) for **B.4**) are encoded in SMT version 2 [3] along with additional optimization commands (e.g., *maximize*) [4, 5] that maximize a designated expression (e.g., Expression (10) for **B.4**). The encoded expressions along with the decomposition operator are output to Step 2 in Figure 2.

3.1.2 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 (6)),
- Measures of satisficement for requirements based on their decomposed requirement's utility functions (e.g., Expressions (7) and (8)), and
- Measures of the maximum numerical difference between the satisficement of a requirement and its aggregate decomposed requirements (e.g., Expression (10)).

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 be 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 re-

quirements, or correcting individual requirements.

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 functions. 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 increasing the 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, 17]. 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, *Ares* can provide scalable analysis, even for large requirements specifications. The execution time of the case study detailed in Section 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.

4. CASE STUDY

This section describes the case study results of applying *Ares* to the ACC system in Figure 1, referred to as goal model *M*.

4.1 Incomplete AND Decomposition: Goal D.1

The ACC system defined in Figure 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. 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 requirements 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*). 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-M) Generate Detection Logic for Model M

We start by maximizing a version of Expression (10), instantiated for **D.1**:

$$\text{Satisficement}_{\text{Decomp}}(\mathbf{D.1}) - \text{Satisficement}(\mathbf{D.1}). \quad (11)$$

Expanding $\text{Satisficement}_{\text{Decomp}}$ via Expression (8) for **D.1** yields the following expression:

$$\begin{aligned} &\min(\text{Satisficement}(\mathbf{D.2}), \\ &\quad \text{Satisficement}(\mathbf{D.5}), \\ &\quad \text{Satisficement}(\mathbf{D.3}) - \text{Satisficement}(\mathbf{D.1}). \end{aligned} \quad (12)$$

Figure 3: Example AND Decomposition in Goal Model M

Employing the utility functions defined in Table 3, where *true* is mapped to 1.0 and *false* is mapped to 0.0, Expression (13) is generated where the parent requirement is subtracted from the minimum (i.e., equivalent to an AND operation) of the decomposed requirements.

$$\begin{aligned} \min(&Speed_t == Desired\ Speed \wedge \\ &Distance > Safe\ Distance, \\ &true, \\ &Throttle\ Actuator == Throttle\ Pedal\ Sensor) - \\ &Speed_t == Speed_{t+1} \end{aligned} \quad (13)$$

2-M) Process Logical Expressions & Summarize

The completeness counterexample expression (Expression (13)) is maximized, while ensuring the domain properties and assumptions (i.e., $Speed_t$, $Speed_{t+1}$, and $Distance$) in Table 3 are maintained, resulting in a maximized value of 1.0 (i.e., *true*). This 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 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, 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.

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 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.

Table 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

Figure 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:

$$Satisficement(\mathbf{D.New1}) = \quad (14)$$

$$Brake\ Actuator == Brake\ Pedal\ Sensor$$

$$Satisficement(\mathbf{D.New2}) = true, \quad (15)$$

where $Satisficement(\mathbf{D.New1})$ includes two **MON** properties: *Brake Actuator* and *Brake Pedal Sensor*. Goal **D.New2** includes no properties.

Inserting these additional utility functions (Expressions (14) and (15)) to those defined in Table 3, results in the following completeness counterexample detection expression:

$$\min(&Speed_t == Desired\ Speed \wedge \quad (16)$$

$$&Distance > Safe\ Distance,$$

$$&true,$$

$$&Throttle\ Actuator == Throttle\ Pedal\ Sensor,$$

$$&Brake\ Actuator == Brake\ Pedal\ Sensor,$$

$$true) - Speed_t == Speed_{t+1}.$$

2- M') Process Logical Expressions & Summarize

Maximizing Expression (16) 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 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**.

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 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 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 (10), but instantiated for **B.3**:

$$Satisficement_{Decomp}(\mathbf{B.3}) - Satisficement(\mathbf{B.3}). \quad (17)$$

Expanding Expression (17) via Expression (8) and the decomposition specific for goal **B.3** results in:

$$\max(Satisficement(\mathbf{B.4}), Satisficement(\mathbf{B.15})) - Satisficement(\mathbf{B.3}). \quad (18)$$

Using the corresponding utility functions defined in Table 3 defines the following incompleteness detection expression:

$$\min(Throttle\ Actuator < Throttle\ Pedal\ Sensor, Brake\ Actuator > Brake\ Pedal\ Sensor) - (Speed_t > Speed_{t+1} \vee (Speed_t == MIN \wedge Speed_{t+1} == MIN)). \quad (19)$$

2- M') Process Logical Expressions & Summarize

Maximizing the completeness counterexample Expression (19), while ensuring the domain properties and assumptions (e.g., $Speed_t$, and $Speed_{t+1}$) in Table 3 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 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 unsatisfied, 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 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 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 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 (18) is updated to use the minimum of the decomposed requirements (i.e., AND) instead of the maximum (i.e., OR) in the following equation:

$$\min(Satisficement(\mathbf{B.4}), Satisficement(\mathbf{B.15})) - Satisficement(\mathbf{B.3}). \quad (20)$$

Using the corresponding utility functions defined in Table

3 defines the following incompleteness detection expression:

$$\begin{aligned} & \min(\text{Throttle Actuator} < \text{Throttle Pedal Sensor}, \quad (21) \\ & \quad \text{Brake Actuator} > \text{Brake Pedal Sensor}) - \\ & (\text{Speed}_t > \text{Speed}_{t+1} \vee \\ & (\text{Speed}_t == \text{MIN} \wedge \text{Speed}_{t+1} == \text{MIN})). \end{aligned}$$

2- M') Process Logical Expressions & Summarize

Maximizing the updated completeness counterexample expression (i.e., Expression (21), where the maximum of the satisficement of the aggregate decomposed requirements is used, rather than the minimum) 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 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**.

4.3 Summary

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.

4.4 Threats to Validity

Ares depends on the accuracy of the utility functions provided by Athena [20]. 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.

5. RELATED WORK

While a number of projects have dealt with partial specifications and uncertainty about the requirements (e.g., [23, 27]), 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 approach used by *Ares*.

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 require-

ment 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 [16] and the application of theorem provers to formally described requirements [24]. However, methods that employ formal proofs are heavyweight solutions that require expertise with theorem proving to describe requirements [24] or low-level functional details [16]. *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 [9] 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 [18], 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 [29] and analysis of natural language requirements [13, 19] 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.

6. CONCLUSIONS

In this paper, 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.

Future research directions include applying *Ares* to additional case studies, including case studies in different domains. In addition, we will investigate applying *Ares* to other hierarchical requirement frameworks (e.g., i^*) to further demonstrate *Ares*'s applicability to generalized hierarchical requirement model analysis.

7. ACKNOWLEDGMENTS

The authors would like to thank the MODELS reviewers for their feedback on this work. This work has been supported in part by NSF grants CNS-1305358 and DBI-0939454, Ford Motor Company, and General Motors Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Ford, GM, or other research sponsors.

8. REFERENCES

- [1] R. Ali, F. Dalpiaz, and P. Giorgini. Requirements-driven deployment - customizing the requirements model for the host environment. *Software and System Modeling*, 13(1):433–456, 2014.
- [2] D. Alrajeh, J. Kramer, A. v. Lamsweerde, A. Russo, and S. Uchitel. Generating obstacle conditions for requirements completeness. In *Proceedings of the 34th International Conference on Software Engineering*, pages 705–715. IEEE Press, 2012.
- [3] 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.
- [4] N. Bjørner and A.-D. Phan. νz -maximal satisfaction with $z3$. In *SCSS*, pages 1–9, 2014.
- [5] N. Bjørner, A.-D. Phan, and L. Fleckenstein. νz -an optimizing smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015.
- [6] B. H. C. Cheng and J. M. Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, 2007.
- [7] B. H. C. Cheng, P. Sawyer, N. Bencomo, and J. 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.
- [8] D. R. Cok. The *SMT-LIBv2* language and tools: A tutorial. *Language C*, pages 2010–2011, 2011.
- [9] R. Darimont and A. Van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 179–190. ACM, 1996.
- [10] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [11] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.
- [12] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. 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.
- [13] A. Ferrari, F. dell’Orletta, G. O. Spagnolo, and S. Gnesi. Measuring and improving the completeness of natural language requirements. In *Requirements Engineering: Foundation for Software Quality*, pages 23–38. Springer, 2014.
- [14] S. Fickas and M. S. Feather. Requirements monitoring in dynamic environments. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 140–147. IEEE, 1995.
- [15] E. M. Fredericks, B. DeVries, and B. 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.
- [16] M. P. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *Software Engineering, IEEE Transactions on*, 22(6):363–377, 1996.
- [17] A. Lapouchnian and J. Mylopoulos. Modeling domain variability in requirements engineering with contexts. In *International Conference on Conceptual Modeling*, pages 115–130. Springer, 2009.
- [18] N. 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.
- [19] I. Menzel, M. Mueller, A. Gross, and J. 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.
- [20] A. J. Ramirez and B. H. C. Cheng. Automatic derivation of utility functions for monitoring software requirements. In *Model Driven Engineering Languages and Systems*, pages 501–516. Springer, 2011.
- [21] S. Ranise and C. 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.
- [22] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *International Symposium on Formal Methods*, pages 532–546. Springer, 2009.
- [23] S. Uchitel and M. Chechik. Merging partial behavioural models. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 43–52. ACM, 2004.
- [24] A. Van Lamsweerde et al. Requirements engineering: from system goals to uml models to software specifications. 2009.
- [25] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 70–77. IEEE, 2004.
- [26] J. Whittle, P. Sawyer, N. Bencomo, and B. 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.
- [27] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel. Relax: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 15(2):177–196, 2010.
- [28] E. S. 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.
- [29] M. M. Zenun and G. Loureiro. A framework for dependability and completeness in requirements engineering. In *Latin American Symposium on Dependable Computing*, pages 1–4, 2013.