ADDRESSING OFF-NOMINAL BEHAVIORS IN REQUIREMENTS FOR EMBEDDED SYSTEMS

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Daniel Aceituna

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Software Engineering

December 2014

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Addressing Off-Nominal Behaviors in Requirements for Embedded Systems

By

Daniel Aceituna

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Hyunsook Do
Chair

Dr. Sudarshan Srinivasan

Dr. Gursimran Walia

Dr. David Rogers

Approved:

January 21, 2015
Date

Dr. Brian M. Slator
Department Chair

ABSTRACT

System requirements are typically specified on the assumption that the system's operating environment will behave in what is considered to be an expected and nominal manner. When gathering requirements, one concern is whether the requirements are too ambiguous to account for every possible, unintended, Off-Nominal Behavior (ONB) that the operating environment can create, which results in an undesired system state. In this dissertation, we present two automated approaches which can expose, within a set of embedded requirements, whether an ONB can result in an undesired system state. Both approaches employ a modeling technique developed as part of this dissertation called the Causal Component Model (CCM).

The first approach described, uses model checking as the means of property checking requirements using temporal logic properties specifically created to oppose ONBs. To facilitate the use of model checking by requirements engineers and non-technical stakeholders who are the system domain experts, a framework for the model checker interface was developed using the CCM. The CCM serves as both a cognitive friendly input and output to the model checker. The second approach extends the CCM into a dedicated ONB property checker, which overcomes the limitations of the model checker, by not only exposing ONBs but also facilitating the correction of those ONBs. We demonstrate how both approaches can expose and help correct potential Off-Nominal Behavior problems using requirements that represent real-world products. Our case studies show that both approaches can expose a system’s susceptibility to ONBs and provide enough information to correct the potential problems that can be caused by those ONBs.

ACKNOWLEDGEMENTS

I would like to thank the following people for their assistance and support during the course of my doctoral studies. My foremost thanks go to my advisor, Dr. Hyunsook Do, who provided her unselfish commitment of time, valuable advice, constant support, and encouragement. I want to thank my committee members, Dr. Gursimran Walia, Dr. Sudarshan Srinivasan, and Dr. David Rogers. Dr. Walia was a great encouragement to me as I undertook the process of designing and implementing experiments; his expertise in that area was very valuable and resulted in a profitable learning experience. Dr. Srinivasan's expertise in the area of formal verification was highly valuable and provided the basis for the final area of focus that my research evolved into. I thank Dr. Rogers for his participation as the graduate school appointee. I want to also thank Dr. Kenneth Magel and Dr. Kendall Nygard for their occasional encouragement and advice. Finally, I want to thank the staff of the Computer Science Department for their help during the past years, going back to when I started as a master's student.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES ix

LIST OF FIGURES x

INTRODUCTION 1

Motivation 1

The Off-Nominal Behavior Problem 1

Limitations in Addressing Off-Nominal Behavior Problem 4

Contributions of this Research 6

Dissertation Outline 7

BACKGROUND AND RELATED WORK 8

Issues in Requirements Engineering 8

Verification/Validation 10

Ambiguities 10

Susceptibility to Off-Nominal Behaviors (ONB) 11

Requirements Modeling of Embedded Systems 14

Cognitive Approach to Off-Nominal Behavior Problem 17

Off-Nominal Testing 18

The System Modeling Language (SysML) 19

Formal Specification Languages 19

Model Checking 23

RESEARCH GROUNDWORK 25

Overview of Preliminary Research 25

SQ Querying 29

Experimental Origin of CCM 30
The Three NLtoSTD Experiments 34
Virtual Requirements Prototyping (VRP) 41
THE CCM APPROACH 44
Overview of the Causal Component Model CCM 44
Creation of a CCM 46
Definition of Terms and Constraints 48
Components 48
Component(State) 48
System State 48
Component(State) Transitions 49
System Cause 49
Environmental Cause 50
Compound Cause 50
Causal Relationships 50
CCM Translation Rules 51
Absorption and Propagation Operations 54
CCM Symbolic Rules 54
CCM Numeric Representation 55
Labeled Transition System (LTS) 56
State Profiles 58
Example Application of CCM 60
Levels of Abstraction Possible with the CCM 64
Overview of the Approaches to the ONBP 65
Property Checking Approach 65
State Profiling Approach 66

Fundamental Differences Between Approaches 68
CCM PROPERTY CHECKING APPROACH (PCA) 70
Overview 70
PCA Preliminaries 72
Support Tool Implementation 74
Skid Loader Case Study 75
Creating the CCM 77
Creating the NuSMV Script 81
Creating the NuSMV Temporal Properties 83
Model Checking Results 86
Discussion and Limitations of PCA 92
CCM STATE PROFILING APPROACH (SPA) 94
Overview 95
SPA Preliminaries 98
The Expanded Rules Algorithm 98
The State Profiling Algorithm 99
Support Tool Implementation 107
Excavator Case Study 108
Observations from a Follow-up Study 114
Discussion and Limitations of SPA 117
CONCLUSIONS AND FUTURE DIRECTIONS 120
Merits in Requirements Representation 120
Merits in Requirements Automated Reasoning 120
Merits in Modeling Separation of Concerns 121
Merits in Addressing System ONB Susceptibility 121
Merits as a Model Checker Front End 122

Impact of this Research 122
Stakeholder Involvement 122
Addressing ONBs 122
Future Directions 123
Potential Improvements 123
Other Directions 125
REFERENCES 127

LIST OF TABLES

Table Page

1. The various formal methods used with embedded systems 21

2. Component state entry showing numeric value of component states 49

3. Comparison between the property checking and state profiling approaches 69

4. The nouns and verbs from the skid loader requirements 125

LIST OF FIGURES

Figure Page

1. Research progression behind the dissertation 25
2. The initial conception that eventually led to CCM 30
3. The two steps involved in NLtoSTD translation 32
4. Example of how NLtoSTD would be used 33
5. Design of NLtoSTD experiment one 35
6. Design of NLtoSTD experiment two 37
7. An overview of Virtual Requirement Prototype 43
8. Example of a Causal Component Model (CCM) 45
9. The CCM formal notation is based on function notation 51
10. Three transitions within a component (C1). C2 is a second component 52
11. Simple CCM example 53
12. Symbolic and numeric representation of CCM 56
13. Comparison of CCM (A) and a Labeled Transition System (B) 57
14. Conversion of CCM to Labeled Transition System 58
15. Example of state profiling an LTS 59
16. Example of CCM derived from a set of NL requirements 60
17. Example of the four step CCM creation process 62
18. Example use of a CCM entry table 63
19. Levels of abstraction possible 64
20. Overview of the Property Checking Approach (PCA) 71
21. Overview of model checking 72
22. A typical skid loader with the load bucket up 77
23. The partial CCM of the Skid Loader 80
24. An example of partial NuSMV script 81

25. Counterexample from first run through model checker 87
26. The modified CCM in response to the counterexample of 24 88
27. Counterexample from running the first corrected set of requirements 90
28. Two snapshots in CCM-based counterexample simulation 91
29. Additional restriction is placed on the unlocking of the seatBarLock 92
30. Overview of CCM state profiling approach 96
31. System state with labeled in/out degrees 99
32. A Causal Component Model, of the requirements 102
33. The components used in the excavator case study 109
34. Component State Grid (CSG), used to enter the component states entry table 109
35. The Translation Rules Grid (TRG), used to create the translation rules by the stakeholders 110

INTRODUCTION

Motivation

The motivation behind the research presented in this dissertation was the result of three major observations made from working within the embedded software industry. The first observation is that vehicles such as tractors and skid loaders can experience undesired behaviors, due to human operators behaving in an unintended manner, what is hence referred to as the Off Nominal Behavior Problem (ONBP). The second observation is that to address the ONBP during the requirements phase, stakeholders that are domain expects, best familiar with nominal behaviors, should be involved. The third observation is that if one was to attempt to involve all stakeholders, it would have to be done in a way that can be readily used by industry, which includes a minimal learning curve. This dissertation describes how various areas of research, combined with the mentioned observations, led to addressing the ONBP in a way that allows for stakeholder involvement using an industry-friendly approach. This dissertation describes two approaches to addressing the ONBP, both revolving around a developed modeling technique called Causal Component Modeling. The first approach takes a Property Checking Approach (PCA), the second approach is referred to as the State Profiling Approach (SPA). We begin by defining the Off Nominal Behavior Problem.

The Off-Nominal Behavior Problem

While all activities in the software development life cycle should be carefully performed to build high quality and reliable software, the requirements engineering phase is a very important and critical phase since it has a large bearing on whether the system-to-be is correctly implemented [3, 4]. It is well known that catching an error in the later stage of software development is usually more time consuming and costly than if the error is caught in the early stage [10, 11, 12, 13]. For instance, Boehm and Basili [10] indicate that finding and correcting defects after product release is often 100 times more expensive than correcting them during the requirements and modeling phase. Also, one survey

performed by Hofmann and Lehner [14] found that successful IT projects have spent about 28% of the effort on the requirements phase.

Much research has been performed in developing ways to verify whether a set of requirements is correct (verification) and will result in the system desired by the stakeholders (validation). Lack of adequate specifications is often due to incompleteness, ambiguities [1], and inconsistencies [61] that commonly occur in a set of natural language (NL) requirements. There are also errors due to the inadequate anticipation of Off-Nominal Behaviors (ONB) produced by the system's operating environment. An Off Nominal Behavior (ONB) is a behavior that is not within the realm of an intended behavior. In the literature, an Off Nominal Behavior is often defined as a situation in which an equipment operator, such as an airline pilot, behaves in a manner that was not intended by the designers of the equipment in question [155]. The machine, having an intended normal mode of operation, is then placed in a potentially unsafe situation. As it applies to a set of embedded requirements, an ONB is a behavior that was not anticipated for by the requirements and typically results in undesired states.

If we view a system as a state transition system, where system states are connected by transitions caused by both the environment and the system itself, then an ONB is a behavior in the operating environment that results in the system transitioning into an undesired state from which it has not been designed to recover from. Undesired states are not necessarily states that create a safety hazard, but can also be states that were not intended by a given sequence of transitions. For example, in a system consisting of two sensors and a push button, the following ONB scenarios can occur:

1) A sensor gets stuck, producing the same input voltage continuously and consequently restricting all possible system state transitions to a subset of the system's entire state space. This scenario can lead to the system getting a state that was not intended, given the sequence of actions produced by the environment.

2) A push button switch is pressed and released in rapid succession, resulting in the operator inducing a state transition faster than the system. This scenario can also occur if the system is slower than usual in responding to an action from the environment. In this case, the system can end up in a state other than the state the system would have transitioned to, had the system responded in a timely manner.

3) Two sensors causing two different transitions from the same state at the same exact time. This introduces an element of unpredictability. The problem arises when sensor *A* and sensor *B* can cause a transition from a given state to two different states and it is sensor *A* that is the intended cause of the transition, but somehow sensor B interfered and became the cause instead of *A*.

Note that in the above scenarios, part of the question is one of a system's susceptibility to an ONB. For example, in the third scenario (item 3, above), the goal would be to expose the possibility of this scenario occurring. In this case, the analysis would expose system states that have two competing environmental causes. In the second scenario (item2), the analysis would expose states where an environmental cause is competing with a system cause, which amounts to; exposing the system's susceptibility to ONBs from the environment.

Armstrong et al. have addressed the ONB problem by studying how human operators interact with machinery, and why they may deviate from an intended operating procedure [149]. The result is more of an ergonomic, psychological approach to the prevention of an ONB problem. An alternative approach is to make the equipment more robust, or "foolproof," so that in spite of an operator's off nominal behavior, the equipment cannot be placed in an undesired state of operation. Of the two approaches just mentioned, this research addresses the ONB problem from the system's perspective. To be specific, the goal here is to give stakeholders an indication of how susceptible a given set of embedded requirements is to the environment causing an undesired state. This is a form of "fool-proofing" the system against an ONB problem.

Limitations in Addressing Off-Nominal Behavior Problem

Limitations in addressing ONBP can be summarized as follows. The majority of the efforts in addressing the ONBP is to either minimize a human operator's unpredictable reaction to an off-nominal situation, (such as may arise in a busy airport runway scenario) or perform off-nominal testing in a finished product. Addressing ONBP from the human's standpoint emphasizes that most ONBPs are perceived as originating from human behavior within the system's operating environment. This places the focus of the solution within the operating environment and less within the system itself. While addressing ONBs in the environment should still be addressed, ultimately, regardless of the environment's behavior, an ONB can become a problem if the system allows the ONB to have an adverse effect in how the system achieves it designed objective. Thus, more effort should be made in addressing how the system reacts to and recovers from an ONB. In doing so the system is made more robust and foolproof.

Addressing the ONBP from the standpoint of off-nominal testing emphasizes that some ONBPs are addressed as an after-thought, after the system has been designed and implemented. While off-nominal testing should still be performed in highly safety critical systems, more effort should be made in addressing ONBs early in the development cycle. A preferred phase would be the requirements phase where access to the system domain expert is more readily made. Stakeholders, such as domain experts, are a good source of information when addressing what can go wrong during a human operator's use of a system.

There is also a limitation in how ONBs are exposed and accounted for. This is due to the fact that many ONBs are anticipated by human manual assessment. Little has been done in automating the process of exposing ONBs, with the few efforts using a formal approach such as model checking [4, 6], which is not readily conductive to use during the requirements phase. Thus, there is a need for cognitive friendly and industry friendly approaches to the ONBP. Cognitive friendliness enables the non-technical stakeholders to get directly involved, reducing the strict role of the requirements engineer as the

middleman, and instead allowing both requirements engineer and stakeholders to work as collaborators. The requirement engineer and stakeholders can then produce the specification without the need for extensive experience and training in formal methods, while gaining the incremental feedback that an automatic methodology can provide.

However, achieving industry-friendliness has its own set of challenges. Having reviewed a good portion of the proposed methodologies and tools meant to improve the deficiencies in requirements engineering, and comparing those proposals to the real-world mindset of the requirements engineers and their managers, the following major observation is drawn. Most everyone in the software industry recognizes the need to improve the way requirements are performed, be it the IT or the embedded world; however, most hesitate to adapted automated and tool-based solutions, beyond those that manage the elicitation, and the basic inspection of requirements.

The major reasons for this hesitation are time and money. Both of which are needed to train personnel, and convert the policies and procedures, to a different way of doing things. This resistance to change is further enforced by the fact that a lot of the proposed improvements coming out of academic research require an amount of time and effort to understanding and implement beyond what most managers are willing to invest. Yes, there are software companies, such as those with a CMMI level 5 that would invest the time and effort in order to formally improve the way they perform requirements. But this represents a smaller percentage of the industry, and they tend to hire personnel that are trained in the area of formal methods.

In part, the motivation behind this research is to develop a methodology that requires a minimal amount of time and money to introduce into an industry setting, while affording the defect-exposing capacities of a formal method. Toward this end, the two proposed approaches have been developed as a cognitive friendly approach that can be ultimately used with an existing tool, such as Matlab/Simulink. In the meantime, a stand-alone support tool has been developed, which implements the steps involved in exposing and removing ONB susceptibility. The tool also provides various graphical representations of

the process steps, in order to allow engineers and non-technical stakeholders direct interaction with the tool.

Contributions of this Research

This research has produced two approaches that are both cognitive friendly to both technical and non-technical stakeholders, and is able to determine whether a system, as specified, can be placed in an undesired or unintended state by off-nominal behaviors in the operating environment. This has resulted in other, secondary, contributions that can be extended into future research endeavors. They are listed here:

1) A means to bridge the gap between (informal) natural language requirements and their formal representation through an ease-of-use requirements model called the Causal Component Model (CCM). The CCM can capture system behavior as expressed in a set of requirements. It is cognitive friendly for three reasons:
    a. The CCM focuses on what a system is supposed to do, and not how it is supposed to do it. Quite often engineers will think ahead, and begin to incorporate a design approach during the specifying of requirements. This produces requirements that specify how a system is to be designed, something better suited for the design phase. The CCM forces the focus on what the system is supposed to do, which is more in line with the mindset of non-technical stakeholders, particularly domain expects who have a better understanding of what the system is supposed to do.
    b. The construction of the CCM involves a simple four step process, which focuses on the identification of system components and their respective states. We call these component states, which are readily identifiable by technical and non-technical stakeholders alike.
    c. The ability to model various levels of abstraction. This affords a stakeholder the ability to model the system as they best conceive it, as well as help manage the size of the model.

2) A rules-based language for the CCM, which allows for the automated manipulation of the CCM. The CCM rules can also be used to systematically convert a CCM into Labeled Transition System with global system states and/or into a Petri-Net.
3) The CCM rules can be used to create an expert system from a set of requirements. Such an expert system can then be queried via forward chaining or backward chaining, allowing for further analysis of the requirements.

As further contributions from this research we have developed a means of using the CCM to graphically interpret counterexamples produced by a model checker, as well a means to translate a corrected CCM into a set of structured language requirements. In summary, the goal is to provide the requirements engineering community with a modeling "framework" that can be used to automatically expose and help correct a system's adverse reaction to ONBs as early as the requirements phase.

Dissertation Outline

This dissertation is organized as follows. In Chapter 2, we provide the background information required to put this research in its proper context, and the related work that has been previously performed. In Chapter 3, we present the research groundwork that was performed in route to the development of the two proposed CCM based-approaches. Chapter 4 presents an overview of the CCM and how it can be applied. This overview will help the reader better understand subsequent chapters as well as provide a definition of terms. Chapter 5 provides an overview of the CCM-based Property Checking Approach (PCA), along with a case study that demonstrates the PCA's ability to expose ONBs. Chapter 6, presents the CCM-centric State Profiling Approach (SPA), and provides a case study demonstrating how SPA can expose an ONB and suggest a solution. Chapter 7 concludes with a summary of the research, some lessons learned, and possible future research directions.

BACKGROUND AND RELATED WORK

This section provides the background information needed to better understand the concepts introduced in this dissertation. We begin by reviewing some of the common issues that arise in the area of requirements engineering.

Issues in Requirements Engineering

It is well recognized that requirements engineering is the most important and critical phase to achieving success [54, 68]. Catching an error in the later stage of software development is usually more time consuming and costly than if the error is caught in the early stage [16, 66, 77]. Yet in spite of its importance, the requirements engineering phase of any given project is often the most neglected. The problem can stem from non-favorable perceptions about requirements [100]. An informal survey of software engineering students, by the author, while teaching a requirements class, found that none of the students had any intention of focusing their future career in the area of requirements engineering. Some of the reasons given were *"I'm not going to school to learn how to write documents, but how to write code."* Another student expressed that they would rather spend the day *"Dealing with a computer than dealing with people"* [as in stakeholders, and customers]. These perceptions can sometimes extend beyond graduation into industry.

Engineering will sometimes view the requirements document as a living document subject to "many changes to come." That being the case, some may question whether it is productive to spend time compiling, writing, and reviewing something that will need to be revised again and again. Project management will sometimes perceive that progress is not being made until the system enters the design and implementation phase [100]. Subsequently, projects are often budgeted with more time and money allocated to software implementation and testing than to the requirements phase [101]. Part of what drives the desire to "Start coding as soon as possible" is that people tend to become engineers because they want to solve problems; they prefer to work in the solution domain.

This is where the act of designing and coding takes place. Requirements engineering, by contrast, operates in the problem domain [93]; its goal is to define and understand the problem thoroughly enough to produce the best-fit solution. Due to a human trait known as Optimistic Bias [69] there is often a tendency to feel that a problem has been understood well enough to start making progress on the solution. However, less time spent in the problem domain, means less time spent understanding the problem. There is also an interesting observation made by renowned computer scientist Frederick P. Brooks [70, 71]; he states something that all engineers know from firsthand experience, but often fail to consider due to their optimistic bias. That is, the lesson learned from examining many projects is that we can't truly understand the problem we are solving until we actually start working on the solution [71]. The human mind needs to see a solution's attempt at addressing a problem, in order to overcome the optimistic bias and get a real sense of the problem's true difficulty level and the challenges that need to be addressed.

This suggests that we can't fully define requirements until we start designing. This may seem to justify the idea of spending as little time as possible in requirements, and get on with design and coding. However, there is still the fact that, compared to the requirements phase, errors in design cost 3-6 times more to fix, while the cost during implementation can be up to 10 times more. Dr. Brooks suggestion is not to spend less time in the requirements phase, but to extend the requirements phase to include more analysis and even some design in the form of a throw away prototype [70, 71]. In this approach a prototype, and the valuable information learned from interacting with it, is used to clearly define the requirements. Akin to developing a prototype is the modeling of requirements.

Various tools now exist that capture requirements in the form of a modeling language, do some degree of validation, and then generate code automatically [88]. Such tools enable the engineer to spend more time understanding the problem domain, and perhaps even eliminate the ambiguity-prone natural language document all together. Some of this research includes more ways to use the classic Artificial Intelligence (AI) technique of knowledge representation and reasoning. This means that requirements

engineering has been trending toward the use of requirements modeling and the automatic analysis of the resulting models via computer.

Verification/Validation

Requirements engineering is often described as consisting of four phases: elicitation, analysis (sometimes referred to as negotiation), documentation, and verification/validation. For our purposes, verification asks *"Are we specifying the system-to-be correctly?"*, while validation asks, *"Are we specifying the correct system*. While it is recognized that sufficient time should be spent in all four phases that is not typically the case in the real world. Elicitation (the gathering of requirements) and its subsequent documentation is a necessity, and thus are always performed. Analysis is often bypassed or merged with the design phase. Verification/Validation often amounts to just human inspection of the requirements document by the stakeholders.

While it can be argued that verification can be perform during the later development phases (at perhaps a greater expense), Validation should definitely occur in the requirements phase to assure that the system desired by the stakeholders is correctly specified. To date, many researchers and practitioners have proposed and developed numerous requirements validation techniques to improve requirements verification/validation processes and their effectiveness, such as reviews, inspections, prototyping, and formal methods [6, 7, 17, 42, 55]. Most organizations will, at least, conduct inspections of the requirements document and/or models [6, 86, 87]. The most commons defects found in a set of requirements can be classified as incompleteness, inconsistencies, or ambiguities. A specification's susceptibility to ONB can be classified as a form of incompleteness because the susceptibility can occur due to a lack of contingencies in addressing unintended behaviors. Unintended system behaviors can also be cause by ambiguities in a specification.

Ambiguities

Ambiguities relates to how the specifications are interpreted by others, particularly further down the development cycle. Because requirements are typically initially expressed in a natural language, they

are highly susceptible to the ambiguities possible within that language. The presence of ONB susceptibility can be viewed as a result of ambiguity, because every contingency for handling an ONB was not explicitly stated in the requirements. For example, imagine a system with water pump that is controlled by the state of one or more water valves. Furthermore imagine that the water pump's behavior is specified during a specific situation, without explicitly specifying the state of a related valve switch. This could result in unintended interactions between the pump and the valve. The valve switch's behavior, in relation to the pump, would be left up to multiple interpretations and thus ambiguously defined. One could also argue that this same scenario would be an example of incompletely defining the interaction between the pump and valve.

Ambiguities are arguably the most challenging defect to expose, in part, because ambiguities are more a product of semantics than syntax. A formal approach is better suited to exposing syntactical errors (which are symbolic in nature), but lacks in its ability to expose errors in the meaning (semantics) behind the symbols. However, in this research we have defined ambiguity as being the opposite of explicitness. In doing so, we equate an ambiguity to an instance of a component-state that has not been explicitly defined. In the proposed Causal Component Model of requirements, the model's numerical representation non-explicitly defined component states appear as zeroes. Thus, we have a correlation between the presence of zeroes and the existence of ambiguities. By "filling-in" the zeroes we eliminate ambiguities. This concept is closely tied to the exposure of off-nominal behavior susceptibility, which we will cover in the next section.

Susceptibility to Off-Nominal Behaviors (ONB)

In a study conducted by Boehm and company, they concluded that *"Many requirements defects are due to Off-nominal behavior, unaccounted for in the requirements"* [10]. In embedded systems, such as the embedded system controller a skid loader, ONBs can posed a safety hazard, if and when an operator behaves in an off-nominal manner which had not been anticipated by the stakeholders. Off-Nominal Behaviors are defined as follows:

"Off-nominal behavior is defined as the class of system behaviors that are outside of the boundaries of the intended behavior (based on purpose and objectives), or the expected behavior (based on performance of realized system)" [151]. While it may be argued that most if not all ONBs will eventually be exposed and addressed prior to product's release, like all other fault classes, the sooner an ONB is exposed, the lesser the time and cost to address. In the literature, ONBs are more often addressed from the standpoint of the system's operating environment. Specifically, the focus is on studying ways to prevent human operators from causing an ONB related problem [137, 138]. ONBs can occur for various reasons. For example, a system's human operator is confronted with an unexpected scenario, forcing the operator to react in an abnormal way, or the system's sensor data can unexpectedly be outside the range or of a different data type. Furthermore, the assumed preconditions for the system's operation could be different than anticipated or non-existent. These ONB problems are often addressed from the environmental standpoint, while focusing on the Human Machine Interface (HMI), because human behavior cannot be fully predictable and is prone to unexpected behaviors [147, 155].

A major industry that has invested considerable efforts to address ONBs is aviation where erroneous human behavior can have catastrophic results [137, 160]. In this context, ONBs can also be called: Off-Nominal cases, Off-Nominal events, mitigation strategies, Off-Nominal events conditions Off-Nominal events operations, Off-Nominal scenarios, Off-Nominal operational requirements, Off-Nominal recovery, or Off-Nominal situations. The aviation industry has tried to address ONB problems during the requirement phase, focusing on anticipating and specifying contingencies (in the form of scenarios) that address potential pilot errors [155]. Neerinex has taken a cognitive engineering approach to address the problem by trying to anticipate an operator's actions and responses to off-nominal scenarios [153]. Giese and Kruger have suggested an iterative methodology to develop a scenario specification from an initial set of nominal scenarios, which are subsequently generalized and used to produce additional off-nominal scenarios [146]. Fraccone et al. have used simulation-based models to create off-nominal conditions for air-traffic procedures [148]. Scenarios and simulation-based models

have also been used to anticipate what contingencies have not specified been and improve the system's robustness, making them more "foolproof" [64].

Compared to the human operator's role, relatively little research has been conducted is addressing the ONBP from the system's standpoint (which adds to the value of this research). One example is the Systematic Off-NOMinal Requirements Analysis SONOMA, which addresses ONBs at the system's design level. However, ONBs should be addressed during the requirements phase for two major reasons 1) There is always a cost and time saving when defects are addressed in the requirements phase. 2) In the requirements phase there is access to stakeholders who are domain experts and have a clearer grasp of the problem being addressed by the system being specified. These domain experts greatly help in determining intended behaviors while exposing unintended behaviors. For example, when specifying the user interaction with an ATM machine, a stakeholder familiar with the details of bank transactions would in better position to describe how the expected behavior of the ATM's user. Recall that in the proposed CCM approach the goal is to explicitly specify the intended behaviors and therefore mitigate all unintended behaviors. This reduces a system's susceptibility to ONBs. A stakeholder familiar with all the intended behaviors is highly valued in this regard.

When addressing ONBs from a system standpoint, there are various methods that focus on systems that have already been implemented (in contrast to our method which focuses on the requirement phase). The other methods include Fault Tree Analysis (FTA) [157], Event Tree Analysis (ETA) [158], Cause-Consequence Analysis (CCA) [159], and the use of model checking [91, 161]. ETAs require (historical) knowledge of an existing system, whereas we are trying to assess unintended behavior from requirement ambiguities. FTAs use a cause-and-effect diagram with digital logic symbols to deal with known failures and try to assess the root cause [157]. CCA integrates fault trees and event trees to predict the effect of a failure scenario. All these methods require knowledge that pertains to a designed or implemented system, as opposed to a system that is being specified and not yet implemented. Model checking is well suited for determining if an undesired state can be reached, a

reach-ability problem, but in our case, we also want to assess both the cause of an undesired state and whether it is non-recoverable. Using our CCM rule-based approach can readily address these questions without needing the temporal-logic knowledge required for model checking [150].

Finally, there is off-nominal behavior testing which typically occurs at the end of the development cycle during software testing, which can involve scenarios designed to expose the software's response to an off-nominal case [155]. Verma et al.'s study utilizes off-nominal behavior test cases for airplane runway operations to expose possible ONBs [147]. As with testing in general, off-nominal behavior testing can have limited effectiveness because exhaustive testing is not practical [154].

It is not unusual for an embedded system's user to find a bug after the system's release into the field. When inquiries about the user behaviors are made, it is often discovered that the user found the bug by interacting with the system in a way other than the "happy paths" [152]. "Happy paths" refer to scenarios under which systems are typically tested and for which risk assessments are made. However, software systems tend to have more transition paths than are typically tested, and there are many scenarios that are not accounted for, in particular ones that are off-nominal [151]. A common consensus is that exhaustive system testing, using every possible test scenario, is unpractical, thus the normal, expected, and intended scenarios are used due to the lack of testing time [152, 154]. Of course, we are not excluding the use of formal methods which often strive to prove the correctness of a system's critical parts [82] but typically require a considerable learning curve. Our approach can be considered a lightweight formal method that is designed for the stakeholders' ease of use.

Requirements Modeling of Embedded Systems

The modeling of requirements has become a very useful endeavor which helps to expose ambiguities and enables the automatic validation of requirements. For our discussion, a model is an abstract representation of a system, in which a particular aspect of that system is emphasized in the representation [87]. That being the case, it is often necessary to model a complex system using more than one model; each model would then offer a different view of the system. This enables an analyst of

that system to focus on one system view at a time. Models are commonly used in the design phase of a systems development, where often the UML, with its multiple modeling views, allows for the complete modeling of a system, both from a structural and behavioral perspective. Models can also be used during the requirements engineering phase as a more formal alternative (and/or compliment) to natural language representation of requirements.

Modeling a set of requirements helps convey information to other stakeholders in a non-ambiguous manner. It also allows for the automated analysis of requirements. Requirements can be modeled mathematically, descriptively, or graphically. A mathematical representation involves the use of a formal and symbolic language that can be manipulated using a set of logic-based rules. Descriptive models can include the use of scenarios, which many in industry have found to be very effective in capturing desired system behavior from non-technical stakeholders [6, 7]. Graphical representation involves the use of diagrams, the most popular in embedded system requirements being variations of a finite state machine. This is due to the state transition nature of embedded systems; embedded systems don't typically terminate, but tend to be in one or more concurrent states at any given time.

Modeling has been accepted as a fundamental activity throughout requirements engineering processes [4, 6, 7]. Typically the models are "virtual" in the sense that they exist in the computer, as mathematical or logical representations of the requirements. Some of the "virtual" models that have been used with embedded systems include Petri Nets [57, 104], Message Sequence Charts [8, 37], and the FSM structure associated with model checking [58]. However, these methods tend to limit stakeholders' involvement, due to up-front amount of technical on-boarding, and they are not necessarily requirements centric. Also, building models often requires NL translation and this translation process can be problematic due to the inherent incompleteness and ambiguities of NL [1, 52, 108]. To address this problem, many researchers have proposed various modeling techniques including partial automated NL translation approaches [6, 7, 8, 53]. These methods include approaches based on translating goals to state machines [6], scenarios to state machines [7], and NL to UML [106].

One method involves analyzing NL requirements (in the form of scenarios) with computational linguistics and generating Message Sequence Charts (MSC) [8, 38]. The MSC charts are then used for verification purposes [A8]. MSCs are interactive diagrams that belong to the Specification and Description Language (SDL). They are similar to the sequence diagrams found in UML. They have become a popular means of specifying scenarios, which describe interactions between objects in a system. They are regarded as useful early in the development stage [38]. An approach by Damas et al. [6, 26] addresses a problem on how to automate a modeling process using scenarios collected from end-users. Their approach captures scenarios using message sequence charts (MSCs) and then builds a state model in the form of a labeled transition system (LST) using MSCs. Similarly, Letier et al. [27] utilize MSCs and LST to analyze requirements, but they focus on detecting implied scenarios. Sutcliffe et al. [29, 25] present a method that converts use case diagrams into scenarios semi-automatically and validates scenarios using rule-based frames that detect incomplete/incorrect event patterns. Another target model that has been used is the Object Oriented Analysis Model (OOAM). One approach uses a tool that automatically creates OOAMs for NL requirements that have been rewritten in a constraint language that facilitates the conversion process [39]. Another approach is to use a conceptual model (an ontology) [40]. When modeling the requirements of an embedded system, the emphasis is often on modeling the behavior of the system. An embedded system typically has more than one type of behavior, each of which should be taken into account when modeling requirements. The following list is not meant to be exhaustive, but rather is meant to illustrate some of the behaviors that can be modeled.

- The cause and effect between the user's action and the actuators, being controlled by the system.
- The behavior of the system's control unit in response to the user's behavior.
- The cause and effect between sensors and the actuators, being controlled by the system.
- The behavior of the system's control unit in response to a sensor's behavior.
- The closed loop behaviors in the system.

- The cause and effect between interrupts, service routines, and possibly actuators being controlled by the system.
- Concurrent behaviors of the various system components.

The CCM was developed to capture the above listed behaviors of an embedded system. The CCM explicitly models the following separation of concerns (note that a component can be a sensor, controller, or actuator).

- The behavior of system component X as it affects system component Y.
- The concurrent behaviors of system components X and Y.
- The behavior of the system's operating environment as it affects the system.
- The reaction of the system to its operating environment.

Cognitive Approach to Off-Nominal Behavior Problem

ONBs can occur for various reasons, including the following three situations [155]. A system's human operator is confronted with an unexpected scenario, forcing the operator to react in an abnormal way. The system's sensor data can be unexpectedly outside the range or of a different data type. The assumed preconditions for the system's operation could be different than anticipated or nonexistent.

The ONB problem is often addressed from the environmental standpoint, while focusing on the Human Machine Interface (HMI), because human behavior cannot be fully predictable and is prone to unexpected behaviors [147]. A major industry that has invested considerable efforts to address ONBs is aviation where erroneous human behavior can have catastrophic results [160, 137]. NASA has also recognized the importance of performing off-nominal testing as a means to assure that its software can deal with unpredictable contingencies [154]. The aviation industry has tried to address ONB problems within the requirement phase, focusing on anticipating and specifying contingencies (in the form of scenarios) that address potential pilot errors [149]. Some researchers have taken a cognitive engineering approach.

These approaches address the problem from the system operator's perspective by trying to anticipate the operator's actions and responses to off-nominal scenarios [153]. Others have suggested an iterative methodology to develop a scenario specification from an initial set of nominal scenarios. These scenarios are subsequently generalized and used to produce additional off-nominal scenarios [146]. Generalization and refinement are performed by subjecting a given scenario to a set of issues in the form of question/action pairs [146]. Some have used simulation-based models to create off-nominal conditions in air traffic procedures [148]. The use of scenarios and simulation-based models, emphasize the need to try to anticipate what can go wrong in the field, and assure that the system can handle such occurrences.

Off-Nominal Testing

Many efforts have been made in the area of off-nominal testing which typically occurs at the end of the development cycle during software testing [133]. These tests can involve scenarios designed to expose the software's response to an off-nominal case [155]. Some claim that Off-nominal testing can improve the understanding of how humans interact with machines, or may expose design issues. [133]. As with testing in general, off-nominal testing can have limitations for its effectiveness because exhaustive testing is not practical [154]. It is not unusual for an embedded system's user to find a system bug after the system's release into the field.

When inquiries about the user behaviors are made, it is often discovered that the user found the bug by interacting with the system in a way other than the "happy paths" [152]. "Happy paths" refer to scenarios under which systems are typically tested and for which risk assessments are made. However, software systems tend to have a lot more transition paths than are typically tested, and there are many scenarios that are not accounted for, in particular those that are off-nominal [151]. A common consensus is that exhaustive testing of a system, using every possible test scenario, is undoable, thus the normal, expected, and intended scenarios are used due to the lack of time [152, 154].

The System Modeling Language (SysML)

The Systems Modeling Language (SysML) is used to model systems engineering applications [151] . The semantics of SysML are similar to, but more expressive than UML, allowing for the specification, analysis, design, verification and validation of a broad range of systems. This makes it useful in requirements engineering. While SysML is typically used to model intended behavior, some researchers have developed a means of modeling off-nominal behaviors by them as the "inverse of intent", and deriving this from the activity models. But this approach requires the use of more than one model, such as Parametric Diagrams, Internal Block Diagrams, Activity Diagrams, Block Definition Diagrams, and State Effects Diagrams. Some of the models are variations of the standard UML models, and require further learning on the user's part.

Formal Specification Languages

Compared to manual inspection, a formal verification of requirements provides a more non-subjective concept of correctness, and can be automated by a support tool. Formal verification also avoids the pitfalls of strict human intervention such as the tendency to skim and make erroneous assumptions. Formal methods often strive to prove the correctness of a system's critical parts [82] but typically require a large learning curve. Our approach can be considered a light-weight formal method that is designed for the stakeholders' ease of use. Compared to manual inspection, a formal approach to the verification/validation of requirements provides a more non-subjective concept of correctness, and can be automated by a support tool. A formal approach also avoids the pitfalls of manual human inspection such as the tendency to skim and make erroneous assumptions. However, to apply a formal approach to set of requirements involves formalizing Natural Language, which a very informal means of knowledge representation. Thus, a major question in the selection of a formal representation for use with requirements is "How easily can a set of NL requirements be formalized."

The next question concerns the means in which the formalized requirements can be checked for correctness, defined as completeness, consistency, lack of ambiguities, and the contingency for all off-

nominal behaviors. The following seven formal specification languages are commonly used for real-time embedded systems: Z, Object Z, VDM++, LUSTRE, ASTRAL, CSP, RAISE, and Petri-Net. Table 1 shows a comparison of these seven formal languages. CCM has been added to the last row, for comparative purposes. Even though CCM has not been referred to as formal specification language, it does formalize a set of requirements, and has characteristics of the other languages as expressed by the parameters listed in Table 1. These parameters are:

Type – The type of representation used in the language. State-based means that the system is represented as a state-machine, Object-based means the system is represented as a set of classes. Data-flow refers to the system being represented as data handler. Finally, a process algebra is a mathematical representation.

- Sync/Async? – The type of system the language can model, whether synchronous (clock based transitions that occur periodically, in a predictable manner), or Asynchronous, (transitions that occur a-periodically; somewhat unpredictable).
- Concurrency? – Asks whether the language can model concurrent processes.
- Executable? – Asks whether the language can be used as a prototype, in that the system behavior can be simulated.
- Time Model – Describes whether the language models a system in discrete time (variables transition from one discrete state to another) or continuous time (variables can assume an infinite number of values between two given system states).

Table 1. The various formal methods used with embedded systems

Language	Type	Sync/ Async?	Concurrency?	Executable?	Time Model
Z	State-based	Sync	NO	NO	Discrete
Object Z	Object-based	Sync	YES	NO	Discrete
VDM++	Object-based	Sync	YES	YES	Discrete
LUSTRE	Dataflow	Sync	YES	YES	Discrete
ASTRAL	State-based	Async	YES	YES	Discrete
CSP	Process Algebra	Sync	YES	NO	Discrete
RAISE	Semi Object-based	Async	YES	NO	Discrete
Petri-Net	State-based	Sync	YES	YES	Discrete
CCM	Rule-Based State-Based	Sync (Async possible)	YES	YES	Discrete

It is reasonable to assume that the parameters of Table 1 are considered important when formally modeling embedded systems. Note that they somewhat complement the embedded system characteristics mentioned at the bottom of section 2.3. To facilitate stakeholder interaction, CCM was conceived with ability to prototype a set of requirements. Recall from section 2.1 that one of the concepts that led to CCM was Virtual Requirements Prototype, which sought a way to validation requirements the same way hardware can be validated by allowing stakeholder interaction via a prototype. Thus, the ability to simulate behaviors is an important feature to have. Since Z, Object Z, CSP, and RAISE are not executable, we will dismiss them as viable competitors to CCM. Instead, we will compare CCM to ASTRAL, VDM++, LUSTRE, and Petri-Net, in greater detail.

ASTRAL defines a system as a set of state machine specifications using a modular abstraction hierarchy approach. This makes it suitable for large systems.

By contrast CCM's scalability comes from the ability to represent a system at different levels of abstractions. Astral has a module construct good for large systems and deals with one transition at a time. CCM focuses on "What" the system is supposed to do; captures minimal details"

VDM++ , The Vienna Development Method (VDM), is one of the longest-established formal method for the development of computer-based systems. Computing systems may be modeled in VDM-SL at a higher level of abstraction than is achievable using programming languages, allowing the analysis of designs and identification of key features, including defects, at an early stage of system development. Models that have been validated can be transformed into detailed system designs through a refinement process. The language has a formal semantics, enabling proof of the properties of models to a high level of assurance. It also has an executable subset, so that models may be analyzed by testing and can be executed through graphical user interfaces, so that models can be evaluated by experts who are not necessarily familiar with the modeling language itself.

LUSTRE is a declarative language that can be used to specify software used in synchronous real-time applications. Being declarative, Lustre does not supports loops, nor recursive calls; it specifies what to do, rather than how to do it, which makes it suitable for use in requirements. A Lustre program is written using nodes and variables, where the variables change value as a function of time. It's syntactical structure is less intuitive that the rule-based structure of CCM. Using Lustre amounts to learning a programming language, which limits its usage to personnel trained in the language, contrary to the goal of encouraging participation of all stakeholders.

A Petri-Net (also known as a place/transition net or P/T net) is one of several mathematical modeling languages for the description of distributed systems. A Petri net is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, signified by bars) and places (i.e. conditions, signified by circles). The directed arcs describe which places are pre- and/or post conditions for which transitions (signified by arrows). Some sources state that Petri nets were by Carl Adam Petri for the purpose of describing chemical processes.

Model Checking

One of the approaches described in this research is model checking, which is a form of property checking. The model checking process is defined as follows. Let M be the FSM representing a concurrent finite-state system, and F be a temporal logic formula that expresses a desired property (specification) that M must satisfy. The model checking problem is to determine all the system state traversals that satisfy F. For example, a temporal specification may specify that a given system state must eventually be reached. The model checking algorithm traverses every path in the FSM and determines whether that system state is eventually reached, satisfying the temporal property. If a property is satisfied, the model checker returns true, if not satisfied, the model checker provides a counter example (a path that shows why the property was not satisfied). Model checking has successfully been used to find previously undetected errors in hardware implementations, such as the IEEE Futurebus+ cache coherence protocol [128] and the IEEE scalable coherent interface [141].

It has also been used in protocol verification. One of the primary problems associated with model checking is the state explosion problem, in which the number of possible system states grows exponentially with each addition system element and the number of states it can produce [91]. Subsequently, there are various forms of model checking, each trying to manage the problem of state explosion, such as partial order reduction [144], bounded model checking [121], and symbolic model checking [127]. Among these approaches, symbolic model checking that uses a symbolic model verifier (SMV) has been a de facto model checking approach [127]. A SMV represents the FSM structure implicitly using a propositional logic formulas, and has been able to model thousands of states [91], without having to construct a FSM, with potentially thousands of states, explicitly. A SMV will also reduce the number of states in the model, using a binary decision diagram.

Another model checking related tool that compares somewhat with CCM is UPPAAL [167], which is used for the simulation and verification of real-time systems. UPPAAL is an integrated tool designed to answer reach-ability questions and consists of a description language, a simulator, and a

model checker. A system in UPPAAL is composed of concurrent processes, each of them modeled as an automaton. The synchronization mechanism in UPPAAL is a hand-shaking synchronization: two processes take a transition at the same time.

With CCM the transitions into the next system state is dependent on the system's present state. Another difference between UPPAAL and CCM is that, while UPPAAL models separate processes, CCM models separate component behaviors. With UPPAAL the separate processes are transitioned by common events; with CCM the separate component behaviors are transitions by other component states. The notation used with UPPAAL is C language-based and has a steeper learning curve than the rules-based notation used with CCM. Of particular usefulness in addressing ONB problems is that CCM can distinguish between E and T transitions more easily than UPAAL.

RESEARCH GROUNDWORK

Overview of Preliminary Research

While it cannot be claimed that all the research in this dissertation was completely pre-planned from beginning to end (rarely does research occur in such a manner), it can be said that every idea that was explored contributed to the dissertation's final form. Figure 1 shows the dissertation's progression from the broad concept of requirement verification, to the specific act of addressing off-nominal Behaviors.

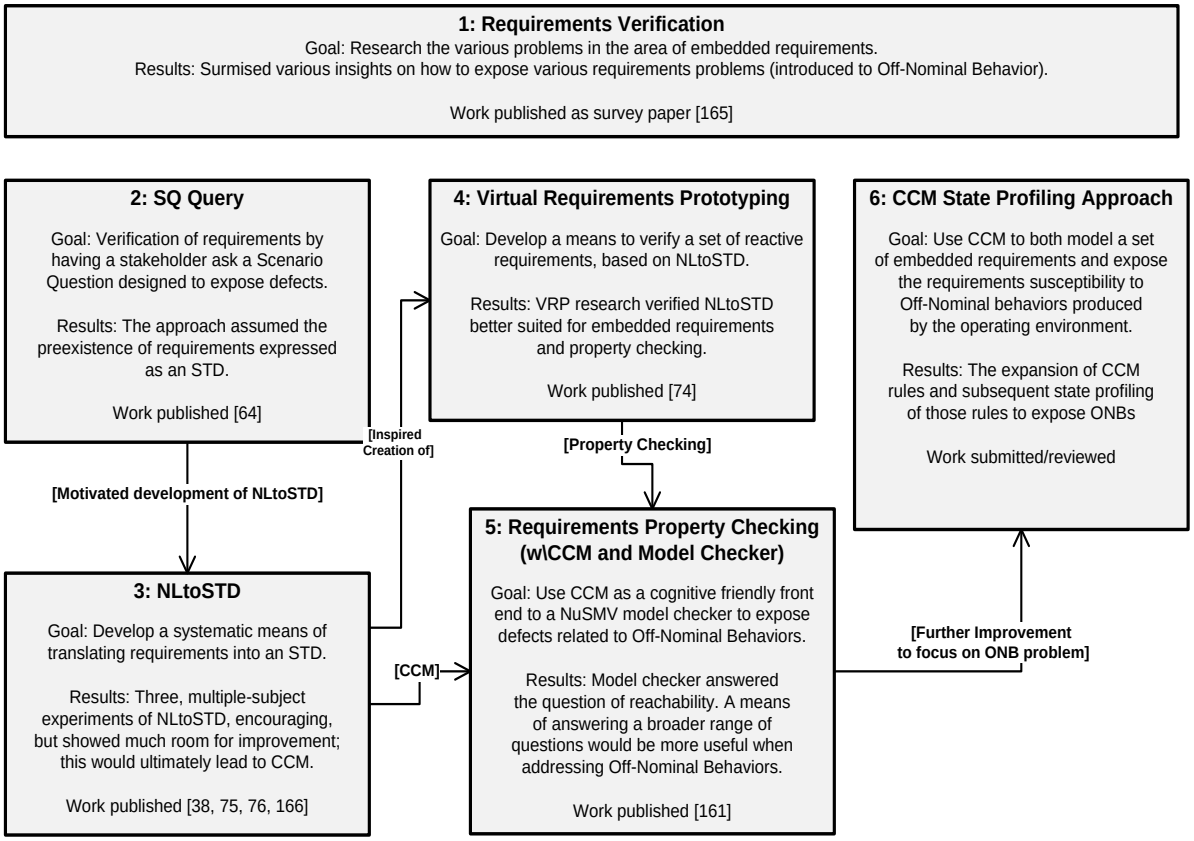


Figure 1. Research progression behind the dissertation

Each block briefly describes a goal, and at least one result of pursuing that goal. The work described in block was subsequently published. The pursuit of each goal resulted in lessons learned, which in turn often became the motivation for another goal. The logical progression from one goal to the

next, is illustrated by the arrows between the blocks of Figure 1. Together, all the blocks show the work flow that culminated in the latest version of the CCM. The workflow both tells the story of how the research progressed from concept to concept, with some publications resulting along the way.

As expressed in a previous section, the initial motivation began with the verification of requirements, which started with the concept of SQ Querying (block 2, Figure 1); the idea of posing a question to a set of requirements. In other words, what if any stakeholder could query a set of requirements in an effort to verify whether or not the behaviors expressed in those requirements were the behaviors they had intended? Such questions could represent scenarios, such as "Will the elevator cab ever move with the doors open?", or "Will the motor ever keep running beyond an overheated state?" The mere act of asking a question would be doable by anyone, with little or no prior training in formal requirements verification.

While the conception of SQ Query (block 2) addressed the key elements that would be needed to achieve its goal, it operated on the assumption that a set of requirements could be readily converted into a State Transition Diagram (STD). The irony was that while asking a question was a simple task, producing the STD upon which the question is asked would not be as simple, often requiring an ad-hoc approach. This motivated the development of a systematic means of translating a set of requirements into an STD; an approach that was named NLtoSTD (Natural Language to State Transition Diagram) (block 3). One of the principle goals in NLtoSTD was to maintain a one-to-one relationship between each requirement and a segment of the resulting STD. This could afford NLtoSTD the ability to expose defects through the mere act of translating the requirements from an informal natural language to a formalized version that would expose incompleteness and ambiguities. Gaps in the translated STD would directly map to defects in the set requirements; facilitating the exposing and correction of those defects. NLtoSTD also inspired the concept of VRP (Virtual Requirement Prototyping) (block 4), which sought to expose defects by virtually prototyping a set of requirements in a similar way that problems are exposed by physically prototyping hardware. NLtoSTD eventually became a research focus in itself

as it was further modified through a series of three multiple subject experiments (block 3). However, the lessons learned from the experiments, prompted the sacrifice of NLtoSTD's one-to-one relationship, to achieve a more consistent means of translation. The result was the development of CCM (Causal Component Model) (block 5). CCM allows the creation of a requirements model that captured not just concurrency (a common characteristic of embedded systems), but also the operating system's interaction with that system. From the research conducted with VRP came the idea of using property checking because it equated with "asking a question" of not only the system's behavior, but part of its structure as well, as in the case of a property design to expose incompleteness. CCM was, thus, combined with property checking by using a model checker as a backend to CCM (block 5). However, model checking is best suited for asking questions that are framed in the form of reach-ability, such as "Can an undesired system state be reached eventually." This is because model checking employees path traversal when analyzing a state transition system.

Ultimately, a wider range of questions could be asked if a rules-based approach was used directly on the CCM. This would help answer questions of susceptibility to off nominal behavior, whether an undesired state can be recovered, and other questions that apply more to structure than just behavior. This motivated focusing on the CCM notation, which is inherently rule-based. Upon viewing our approach as being rule-based, further observations were made.

1) A rule based system can be easily treated as an expert system, which would also allow a more effective implementation of the SQ Query concept.
2) A rule based system can be enacted (via forward chaining) in order to simulate the system's behaviors, which would also allow for the implementation of the VRP concept.
3) Rules could be possibly categorized according to certain properties, which would also allow for the implementation of the property checking concept.

Thus, a rule-based approach could potentially achieve the goals expressed in Blocks 2, 4, and 5. However there were further observations made about a rule-based system that eventually resulted in

what we will refer to as the CCM State Profiling Approach (Block 6, of Figure 1). These further observations include:

1) The rules that define the CCM were ambiguous, leading to the thought that the ambiguities in the rules, correlated to ambiguities in the initial NL requirements.
2) Rules can be selectively removed, which in turn, removes selected behaviors.
3) Each rule can be automatic translated back into a structure NL requirement.

Further research developed a means of converting the ambiguous CCM rules into more explicated rules that could, in turn, be used to produce a Label Transition System (LTS) with global states, similar to the Kripke structure associated with model checking. Eventually, it was found that the set of explicit rules describes all the behaviors possible by the CCM rules including those behaviors that were unintended due the ambiguities in the CCM rules. Block 6, shows the culmination of the research and publications produced in this dissertation. In this final approach, the CCM rules are expanded so as to address every state in the system's state space. The result is a rule for every possible intended and unintended behavior. The rules are examined using a state profiling concept to determine which rules, if any, terminate in a non-recoverable undesired state.

In summary, the ONB problem could have been addressed by SQ querying using scenario questions that describe off-nominal scenarios. Off course, this means that stakeholders would need to anticipate all the possible off-nominal scenarios in advance. As the system grows in complexity, anticipating all possible ONB scenarios because virtually un-achievable. It is more manageable to access the various undesired system states that stakeholders wish to avoid and determine if these states are reachable by actions caused by the operator environment. This was demonstrated as achievable by using a model checker as a backend to CCM. The concept was extended further by expanding the CCM rules into all possible interpretations of the requirements and removing those interpretations that can potentially result in an ONB. In this manner, we address the problem from a susceptibility standpoint.

We now examine SQ Querying, NLtoSTD, and Virtual Requirements Prototyping (VRP) in greater detail.

SQ Querying

With SQ Querying (a.k.a. SQ^2E) the goal was to develop a validation technique that analyzes a requirements model by asking a question about the model's behavior. During the SQ querying process, a Scenario Question (SQ) is first expressed in plain English, and then translated (by the user) into a set of inference constraints, which the tool uses to query a model. The tool then answers a question by generating and displaying state transition traversal path strings (in short, path strings), which represent paths through a model describing its behavior. To investigate the feasibility of this approach, we implemented a tool for SQ^2E that was written in Prolog and C#. The tool required both the inference constraints and the requirements being queried expressed as a State Transition Diagrams (STD).

We applied the tool to simple case study, and observed that SQ querying could not only catch a model error, but also prompt questions that lead to a model's improvement. The key to a SQ query's strength lies in involving users, and allowing them to verbally express a scenario question. The STD traversal paths strings, which are final results, are visually verified by the user, providing more information than what is normally obtained from a simple binary pass/fail result. The involvement of the human in the SQ querying process adds a human interpretation element; examining the various path strings for a given query can often bring to mind other questions, which ultimately leads to a more in-depth analysis.

Through our case study, we observed that our approach can be also applied to software testing process in addition to requirements validation; our SQ querying approach can also facilitate exploratory testing, which is an ad-hoc approach to testing software in which human testers test software systems in an exploratory manner [17].

In exploratory testing, test cases are not defined in advance; they are formulated and executed while testers walk through the product. One of the benefits of exploratory testing is that it provides a learning experience for the person doing the testing. Exploratory testing provides testers the flexibility to explore the software's behavior, and relies on feedback from the software system as human testers assess

the effects of their actions on the software under test. The ability to query a model and receive feedback in the form of path strings provides a similar form of interactions that is one of key characteristics of exploratory testing. Thus, it was thought that SQ querying would appeal to those who gravitate toward exploratory testing. The main drawback with SQ2E was the presumed existence of requirements expressed as an STD. This inspired the need for CCM.

Experimental Origin of CCM

CCM was developed through a progression of three separate experiments that were performed at NDSU. The experiments were designed to evaluate a proposed approach for exposing defects in a set of requirements. The approach involved creating a requirements model in such a way that NL defects, such as incompleteness, inconsistencies, and ambiguities, would carry into the model. The model would then expose visually or by automatic analysis of the requirements defects.

The rationale was that it would be much easier to expose defects in a requirements model than in the Natural language itself. This would be particularly true if the model has a direct one-to-one correspondence with the requirements, so that a defect exposed in one section of the model would immediately point to the requirement containing that defect. The idea came from visualizing the scenario illustrated in Figure 2.

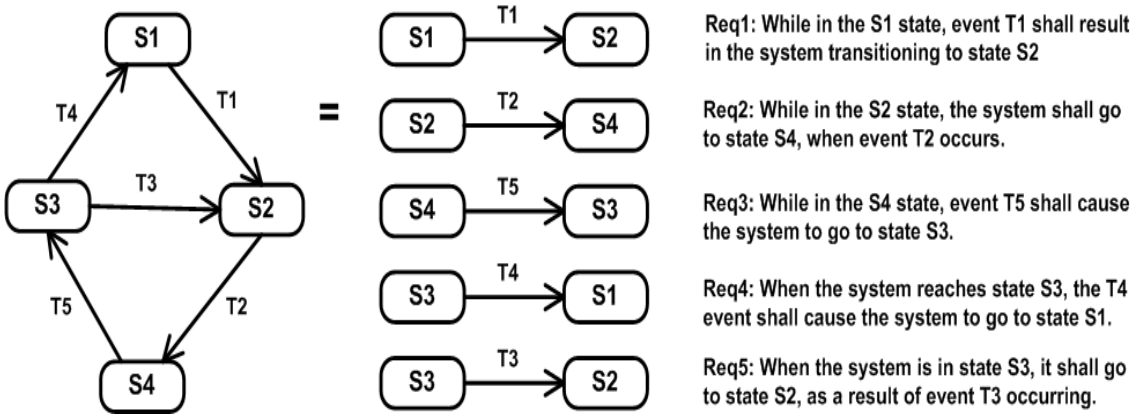


Figure 2. The initial conception that eventually led to CCM

The extreme left hand side of Figure 2 shows a State Transition Diagram (STD) consisting of four states (S1, S2, S3, and S4), and five transitions (T1, T2, T3, T4, and T5). Let us imagine that the STD models a specified system in exactly the way intended by the stakeholders. In other ways, all the desired behaviors are captured by the STD, and no unintended behaviors can occur; the STD is free of defects. Let us decompose the STD into the five segments shown in the center of Figure 2 (henceforth, these segments will be referred to as STD-Building Blocks (STD-BB)). It stands to reason that if the STD is defect free, then so are the building blocks. Let us finally translate the building block verbatim into the set of requirements shown on the right hand side of Figure 3. Again it stands to reason that if the building blocks are defect free, and they have been literally translated, then the resulting requirements are defect free as well.

The confidence that a correct STD results in a correct set of requirements is based on there being a direct one-to-one mapping between STD, its building blocks, and the requirements. This concept also applies in the opposite direction when there is a one-to-one mapping starting from requirements to a STD. Thus, the goal of NLtoSTD was to translate natural language requirements into an STD, while maintaining a one-to-one mapping; any defects in the requirements would map into the STD where the defects can be more readily exposed, and corrected. Once a correct STD is obtaining, it is converted back into a corrected set of requirements, using the same one-to-one mapping. The NLtoSTD method consists of the two steps shown in Figure 3.

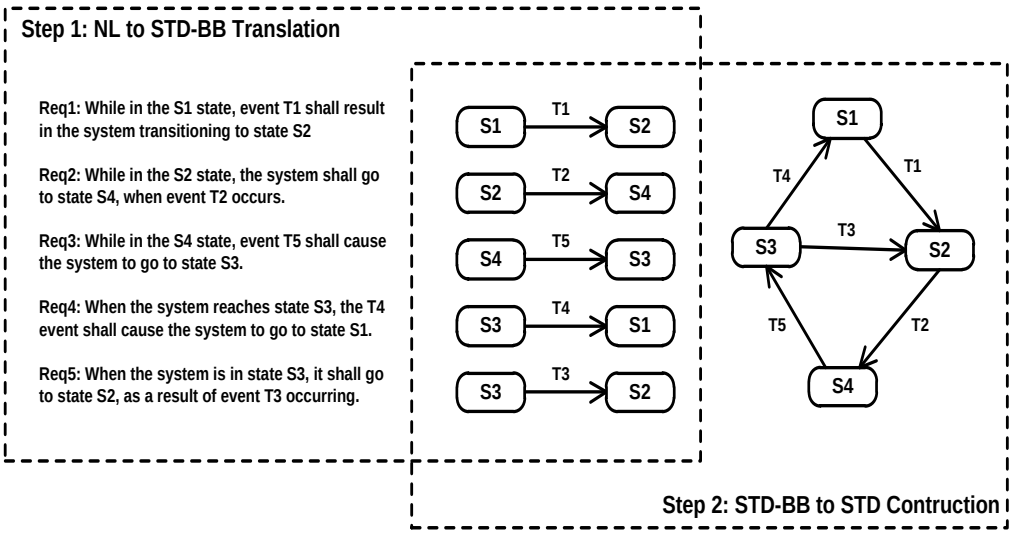


Figure 3. The two steps involved in NLtoSTD translation

Step 1 (the left side box with dotted line) involves the translation of NL requirements into STD-BBs while Step 2 (the right side box with dotted line) involves the construction of an STD using the STD-BBs. The objective in Step 1 is to convert each requirement into a STD-BB. The user is to examine each individual requirement and determines the three elements: (1) a current state (Sc), (2) a next state (Sn), and (3) a transition (T), that make up each STD-BB. If we assumption that each functional requirement describes a behavior that results in something changing state, then ideally each requirement should express a precondition in the form of a current state (Sc) and a post condition in the form of a next state (Sn). The cause of that transition (T) should also be indicated.

However, typical NL requirements do not explicitly state current and next states, thus a requirement's preconditions and post conditions are often inferred and not explicitly stated. In the ideally stated requirement, preconditions and post conditions should be explicitly expressed to minimize ambiguities and incompleteness. Similarly, the absence of the explicit transition (T) information can cause different interpretations for the same requirement by different stakeholders or inspectors.

The NLtoSTD-BB method would force the inspectors to look for these three elements in a NL requirement, thereby ensuring that the requirements are as concise and clear as possible. Figure 5

illustrates the process of converting a set of five requirements into seven STD-BBs. Note that while there are five requirements and seven STD-BBs, there is still a traceable mapping between each STD-BB and one of the five requirements. The red "???" in the STD-BBs indicate states and transitions that were not explicitly stated. This in turn results in an incomplete STD at the right-hand side of Figure 4.

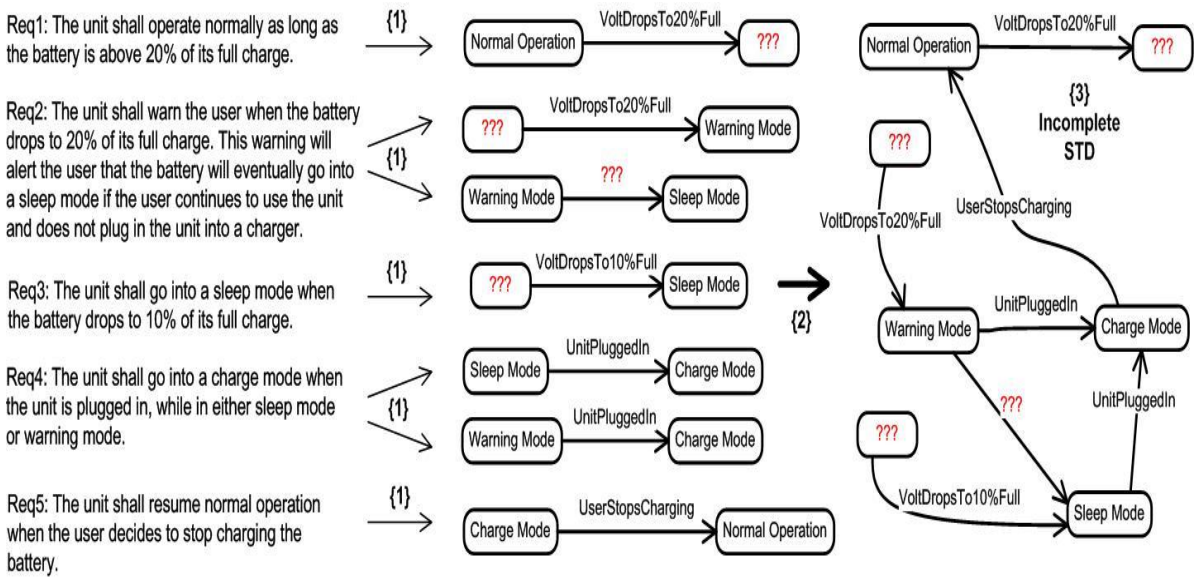


Figure 4. Example of how NLtoSTD would be used

In the case of ambiguities, the user is encouraged to enter their interpretation of states and transitions. Since the resulting STD can be analyzed for intended behaviors, an ambiguous interpretation will be exposed eventually. Inconsistencies can also be exposed through automatic analysis of the STD. The STD's behavior could be simulated by a support tool in order to expose faults that may not be evident unless the STD is enacted. Eventually, a series of the three experiments were conducted in order to determine three primary factors. 1) how effective would NLtoSTD be at exposing faults. 2) how efficient would NLtoSTD be at exposing faults. 3) How cognitive friendly would NLtoSTD be to utilize. The results of each experiment is summarized in the next three sections.

The Three NLtoSTD Experiments

The CCM was developed as the result of three experiments conducted to evaluate the feasibility and effectiveness of NLtoSTD. In the first two experiments a control was used in the form of a fault-check inspection list, the third experiment used only the NLtoSTD approach. There were three research questions (RQn) investigated during each of the three experiments.

RQ1: Is NLtoSTD-BB more effective (i.e., the number of faults) at detecting incomplete and ambiguous requirements compared to the fault-checklist based method during inspection of an NL requirement document?

RQ2: Is NLtoSTD-BB more efficient (i.e., faults per hour) at detecting incomplete and ambiguous requirements compared to the fault-checklist based method during inspection of an NL requirement document?

RQ3: Is NLtoSTD-BB viewed to be useful for improving the software quality?

As we progressed through each experiment, modifications were made to NLtoSTD. By the end of the third experiment, NLtoSTD was modified to a degree to where it became CCM. Each of the three experiments, along with the results was subsequently published, in great detail. In this dissertation we will summary the experiments beginning with experiment one. For clarity we will align each experiment with a version of NLtoSTD, labeled NLtoSTD V1.0, NLtoSTD V2.0, and NLtoSTD V3.0.

Experimental Design of Experiment One

The first experiment focused on the creation of NLtoSTD building blocks from a set of requirements. The goal of the first experiment, a repeated measures quasi-experiment, was to understand whether the original version (V1.0) of NLtoSTD-BB can be effectively used to detect faults in NL requirement documents [75]. To accomplish that goal, we compared the NLtoSTD-BB V1.0 against the traditional fault-checklist based method in the context of their ability find faults in the software requirement documents developed by student teams.

The experiment's design is shown in Figure 5. The participating subjects included sixteen (16) students enrolled in the Requirement Definition and Analysis course at North Dakota State University in Fall 2010. During the semester, the students worked in teams (of three or four subjects) to elicit and document the requirements for a different software system (that was selected by the students and agreed upon by the instructor).

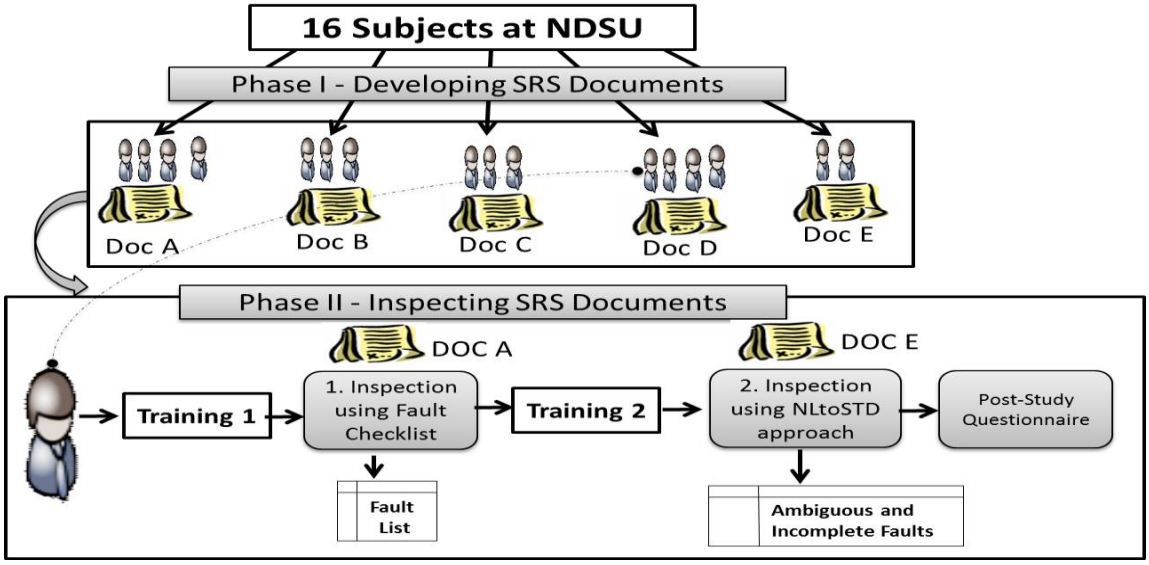


Figure 5. Design of NLtoSTD experiment one

A total of five different requirement documents (referred as Document A through E) were developed by student teams and the details about them can be found in [75], along with the size (in terms of the number of pages) of the requirement documents. After the requirements development, each subject individually inspected two different requirement documents (developed by other student teams) using two different inspection methods: the first inspection using the fault-checklist method followed by the second inspection (of a different document) using the NLtoSTD-BB method.

When using the NLtoSTD method, the subjects were asked to focus on two fault types Missing Functionalities (MF) (i.e. incompleteness) and Ambiguous Information (AI). Missing Functionalities equate to incompleteness, for example, if a Present state was not mentioned at all, the subjects were to enter a "???". Ambiguous information was information that was implied but not explicitly stated, for

example, if the requirement's wording implied a present state of "ON" then the subject was to also enter "???" (a better distinction between MF and AI was made in the second experiment.)

Results from Experiment One

Research Question 1, the effectiveness of NLtoSTD: NLtoSTD was slightly more effective compared to the fault-checklist method, when used by subjects who clearly understood the application of the NLtoSTD-BB method.

Research Question 2, Is NLtoSTD-BB more efficient (i.e., faults per hour): The result from the first experiment showed that the fault checklist was more efficient at finding faults. This was mainly because of the two reasons: (1) the subjects using the fault checklist looked for ten types of faults, whereas the subjects using the NLtoSTD-BB method focused on detecting two fault types (MF and AI); and (2) four (of eleven) subjects using the NLtoSTD method found no true faults due to their misunderstanding of the translation process. However, the results from the first experiment provided us insight to improve the translation process.

Research Question 3, Is NLtoSTD-BB viewed to be useful for improving the software quality?: The subjects' responses to the post-study survey show that, in general, the NLtoSTD-BB method is viewed favorably for most attributes. For the first experiment, some subjects reported problems that they faced while choosing the values for Sc, T, and Sn when translating NL into STD-BBs. Based on the students' responses and feedback, we revised the NLtoSTD method to make it easier to understand and apply to NL requirements.

The first experiment showed that the method was substantially better at exposing incompletenesses than ambiguities. The first experiment demonstrated that it was feasible to expose faults using an approach where a set of NL requirements is formalized into a state transition diagram building blocks. These results supported our belief that the nature of the NLtoSTD-BB translation process exposes the human tendency to not explicitly state the precondition (current state) associated with a given requirement.

Experimental Design of Experiment Two

Using the revised NLtoSTD-BB method (referred to hereafter as NLtoSTD-BB V2.0), we conducted the second experiment by replicating the first experiment with a different experimental design [76]. This study utilized a repeated-measure design (with a complete counterbalancing of the treatment order) in which the subjects inspected the same NL requirement document (developed externally) using both the NLtoSTD-BB V2.0 and the fault-checklist methods. The experiment occurred over a two-week period, and the subjects were divided into two groups. During week one, a group of subjects inspected the document using the fault-checklist method while the other group of subjects inspected the same document using NLtoSTD-BB V2.0. As shown in Figure 6, sixteen participating subjects were "randomly" divided into two groups of eight subjects each (Groups A and B). Also, each subject performed two inspections of the same requirement document (LAFS) using different inspection methods (fault checklist and NLtoSTD-BB V2.0). Finally, subjects within Group A and Group B differed in their treatment order for inspection methods during the first and the second inspection cycles.

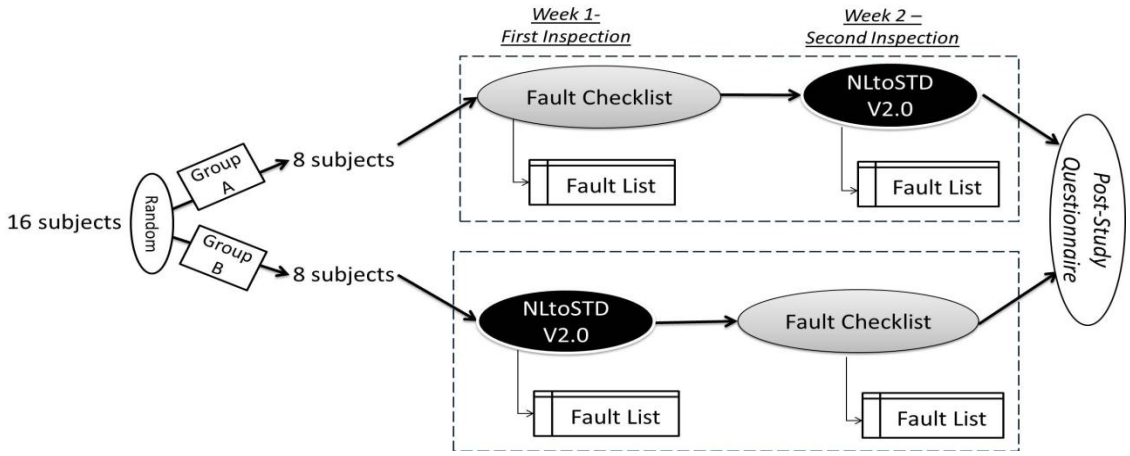


Figure 6. Design of NLtoSTD experiment two

During the second week, each subject inspected the same requirement document using the other inspection technique and reported new faults that were not detected during the first inspection. Therefore, these two inspections resulted in a list of faults for each subject using NLtoSTD-BB V2.0

and a fault checklist during the first and second inspection cycles. Sixteen Computer Science graduate students enrolled in the Software Design course at North Dakota State University during Spring 2011 participated in this experiment. These individuals were predominantly master's and Ph.D. students and had taken a requirement engineering course prior to this study.

The Software Design course focused on analyzing design decisions and implementing software designs. During this two-week experiment, the participants inspected a generic NL requirement document describing the requirements for the Loan Arranger Financial System (LAFS) that was created by professional developers at the Microsoft organization. The Loan Arranger system is responsible for bundling loans for sale based on user-specified characteristics. For use in previous studies [35], researchers seeded the artifact with realistic defects.

Results from Experiment Two

Research Question 1, the effectiveness of NLtoSTD: The results of the second experiment showed further improvement over the results from the first experiment. The results showed more consistent (across all subjects) improvement for effectiveness when using the NLtoSTD-BB V2.0 method. In particular, NLtoSTD-BB V2.0 was significantly more effective for AI fault type at $p = 0.1$ level than a traditional fault-checklist based inspection method.

Research Question 2: Is NLtoSTD-BB more efficient (i.e., faults per hour): The results showed that the NLtoSTD-BB V2.0 method was more efficient than the fault-checklist method during the first and the second inspections. During the first inspection, subjects using NLtoSTD-BB V2.0 found an average of 19 faults per hour compared to subjects using the fault-checklist method who found an average of 16 faults per hour. However, the difference in the efficiency values was not statistically significant ($p = 0.49$). During the second inspection, NLtoSTD-BB V2.0 (an average of 10 faults per hour) was slightly more efficient than the fault-checklist method (an average of 9 faults per hour). Therefore, even though the results were not statistically significant, NLtoSTD-BB V2.0 improved the efficiency of the participating subjects compared to the fault checklist for both inspection cycles.

Although the results from the second experiment was promising, they furthered the need to improve NLtoSTD-BB V2.0 in those areas that we felt were hindering the ability to use the method, as it was revealed from the third research question (RQ3).

Research Question 3, Is NLtoSTD-BB viewed to be useful for improving the software quality?: The questionnaire, which addressed RQ3, showed that, while the method used in the second experiment was easier to apply, there was still room for improvement. The subjects rated the NLtoSTD-BB V2.0 significantly positive on all the attributes in comparison the NLtoSTD-BB V1.0. So, the revisions were justified by the improvement in the responses of NLtoSTD-BB V2.0. The inspectors often reported false faults that included information they had assumed was either missing (MF) or not clearly defined (AI) in the functional requirements, but was sometimes found in the preceding sections, such as the glossary or purpose. False positives associated with NLtoSTD-BB V2.0 were mainly attributed to a lack of fully understanding how to apply the method, difficulty deciding which entity to use (especially in the case of a lengthy, non-cohesive requirement), or mistaking an operator for state. However, none of the false positives with NLtoSTD-BB V2.0 were due to the incorrect assumptions or being unable to properly comprehend the complete requirement list.

The second experiment showed that the improvement made to the method enabled it to find a greater amount of ambiguities relative to incompleteness. This was an encouragement because the second experiment suggested that the method's improvement resulted in the approach that can more equally expose incompleteness and ambiguities. Also, the results from the second experiment showed that NLtoSTD-BB helped locate precondition related problems that were otherwise undetected during the fault-checklist based inspection process. Although the results from the second experiment was promising, they furthered the need to improve NLtoSTD-BB V2.0 in those areas that we felt were hindering the ability to use the method, as it was revealed from the third research question (RQ3).

Experimental Design of Experiment Three

Experiment three was divided into two phases [162]. The first phase involved translated the requirements into NLtoSTD-BBs. The second phase involved creating a STD using the NLtoSTD-BBs derived in phase one, and analyzing the STD. A support tool was used to build the STD from building blocks. The student participants, were divided into 5 different teams of three or four participants. The students in each team individually inspected their own SRS document using the NLtoSTD method. The participants were first trained on how to map the NL requirements to STD-BBs, while documenting the building block elements in an Excel spreadsheet using an in-class example. The subjects were instructed how to load the BBs (from the spreadsheet) into the tool and then, how to construct an STD from the BBs. The subjects were then instructed to examine the constructed STD. Finally, the participants learned to record the fault type in the fault spreadsheet. To ensure that subjects understood, the subjects practiced these steps through an example system.

Results from Experiment Three Results

Research Question 1, the effectiveness of NLtoSTD: Based on the results, additional MF and AI fault types are uncovered during the examination of STD constructed from the BBs. In particular, the creation of STD aids inspectors at detecting incompleteness defects that are otherwise not apparent when looking at individual requirements. The construction of STD is useful for overall inspection effectiveness using the complete NLtoSTD method.

Research Question 2: Is NLtoSTD-BB more efficient (i.e., faults per hour): The NLtoSTD-BB method helped inspectors find inherent ambiguities and incompleteness in requirements. The comparison of the results against the previous research results [5] revealed that the subjects were able to find larger number of total faults (on average), and the distribution of faults across fault types (MF and AI) was more consistent. The results also showed that the NLtoSTD-BB method helped find the faults faster (i.e., efficiency) when compared to the results in [5].

Research Question 3: Is NLtoSTD-BB viewed to be useful for improving the software quality?: The subjects provided insights in to the use of the NLtoSTD method and improvements that can help improve the performance in future studies. The subjects mentioned that the tool should guide the NLtoSTD-BBs translation and should at least highlight parts of STD that are completely disconnected.

The results are organized around the comparison between the NLtoSTD-BB method and the traditional fault-checklist method in detecting requirement faults. The results showed that the subjects who were able to correctly extract the STD-BBs were able to find larger number of "incompleteness" in NL requirements when compared to the fault-checklist method. Using a 5-point likert scale (ranging from "very low" to "very high", the participants rated the NLtoSTD-BB method on eight different attributes related to its usability during the inspection. The results showed that only the simplicity and easy-to-understand attributes were rated significantly positive (i.e., mean response was significantly greater than 3 – the midpoint of the scale). The subjects response to post-study questionnaire revealed that, it was, sometimes, difficult to choose Sc, T, and Sn values when translating NL into STD-BBs for the requirements. Furthermore, analyzing the fault record forms provides insights that the NLtoSTD-BB method has limitations where there are multiple requirements specified for a single functionality, that causes the users to select from multiple candidate values of STD-BB elements (i.e., Sc, T, Sn) causing an incorrect transformation of NL requirements.

Virtual Requirements Prototyping (VRP)

The primary motivation behind VRP is to provide a requirements validation approach that overcomes drawbacks of current approaches we mentioned in Section I and brings stakeholders into the validation process. Validation generally addresses the question of whether the right system has been built. This question can be best answered by various stakeholders, such as customers, domain experts, and end-users, who typically are the ones requesting the system-to-be. Inspection is one of the widely accepted and used validation techniques because it is relatively easy to apply and it does not require technical knowledge for stakeholders to use it, but it is error prone due to human limitations in finding

faults in documents describing complicated embedded systems. Therefore, requirement engineers often seek to validate requirements in ways that minimize human errors, but this means they need to use techniques that extend beyond the technical expertise of non-technical stakeholders This creates a situation in which stakeholders (e.g., endusers and domain experts), who should be performing the validation, may not be able to do so, because requirement engineers could be the only qualified people who can perform the validation using such techniques. In developing the concept of VRP, we were motivated by several factors that allow for stakeholders' participation. These factors include: 1) The use of a virtual prototype that allows the stakeholders to validate through their interactions; 2) Stakeholders can incrementally create the prototype themselves through a question-based means of transforming NL requirements into a Virtual Prototype (VP); based on a concept we developed called NLtoSTD [76]. Stakeholders can evaluate the VP by using a technique we developed called SQ Querying, which is a question answer based approach of interrogating a set of requirement with Scenario Questions [64]. We facilitate the asking of questions by using matrix based user-interfaces. To answer the stakeholder's questions, we use easy-to-read path strings and requirement scenarios, which will be covered in detail. Lastly, with the emphasis of human involvement, the VRP supporting tool will incorporate automated reasoning.

VRP is an approach to validating NL functional requirements by transforming them into a virtual prototype that stakeholders can then interact with. There are three major phases to VRP. Each phase can be revisited at any time during the course of validating a set of requirements. The three phases are as shown in Figure 7, and are listed as follows.

1) Virtual Prototype Creation Phase Users transform the Natural Language (NL) requirements into a Virtual Prototype (VP). In this phase NL ambiguities can be greatly reduced, because the transformation process forces a reevaluation of the users' true intention, as worded in the requirements. The transformation process is such that a direct one-to-one trace between each NL requirement and a building block of the VP is established.

2) Virtual Prototype Correction Phase Users make any necessary initial corrections in the VP's structure. In particular, users address gaps in the VP, which are direct reflections of the incompleteness in the NL requirements. Since the first phase established a direct one-to-one trace between NL requirements and the VP, any corrections made to the VP can instantly be mapped back to the responsible NL requirement.

3) Virtual Prototype Evaluation Phase After the initial corrections have been made to the VP, it is ready for the users' evaluation. This phase exercises VP behavior, and uses STD traversal path strings and requirement scenarios to expose defects, including inconsistencies.

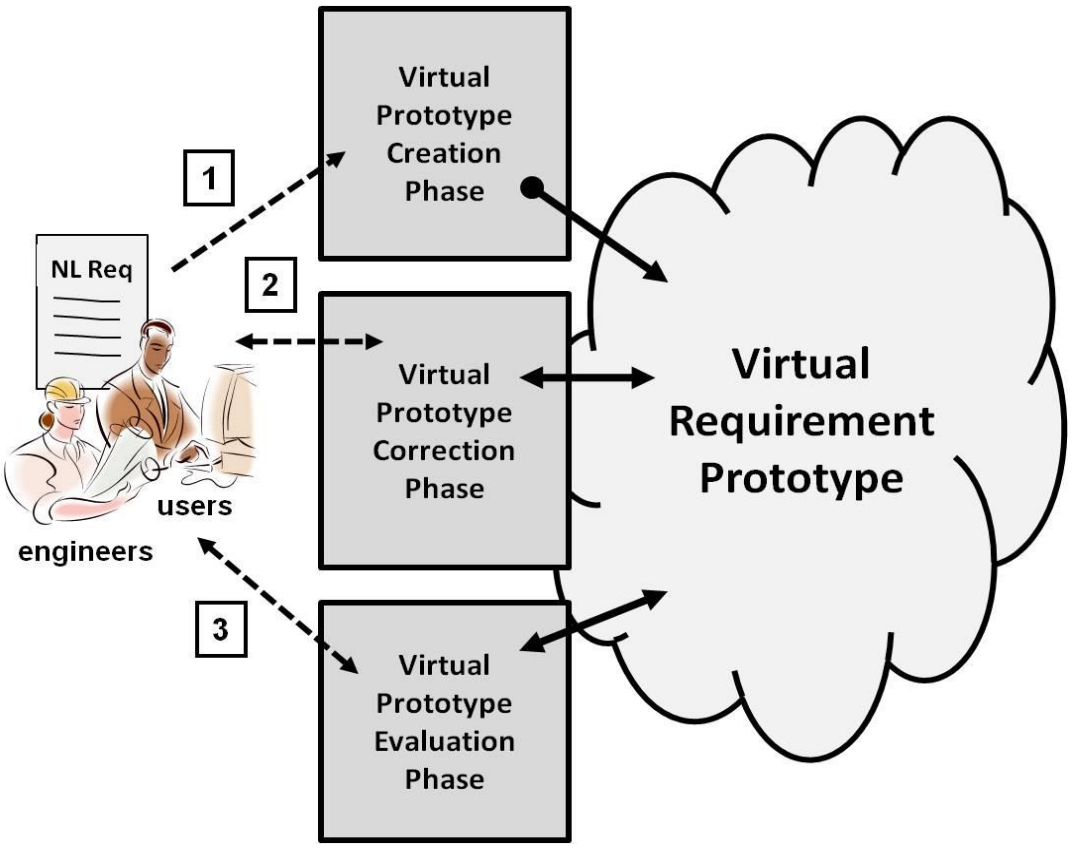


Figure 7. An overview of Virtual Requirement Prototype

THE CCM APPROACH

Overview of the Causal Component Model CCM

The Causal Component Model (CCM) has been made cognitive friendly through repeated experimentation, and can potentially be automated in the future by using a Part-Of-Speech (POS) tagger. CCM was developed to provide the following factors deemed necessary for this research.

1) The model must capture system behavior**,** since the goal is the exposure of ONBs. A common means of modeling behavior is using the concept of a state machine (a.k.a. State Transition Diagram (STD)). For example, UML models such as Activity Diagrams, State Charts, Collaboration Diagrams, and Sequence Diagrams are built around the concept of a state machine. State machines are intuitive, fairly easy to understand, and used in other aspects of the development cycle (such as the design phase); this gives them a high degree of familiarity. More importantly, for our purposes, state machines can be modeled mathematically as. graphs, which enables the automated manipulation and analysis of a state machine.

2) The model should be systematically constructible by stakeholders with varying degrees of technical training, since we are exposing ONBs during the requirements phase. This promotes stakeholder involvement, when possible. A CCM is constructed using a four step process, with each step centered on the concept of a component state. A high degree of specification detail can be sacrificed for ease of construction, since in requirements we are primarily interested in capturing *WHAT* a system is supposed to do, and not *HOW* it is supposed to do it; the "how" is left to the design and implementation phases.

3) Since the goal is to expose ONBs caused by the operating environment, the model must abstract environmental causes from system causes. CCM is designed to differentiate transitions cause by the environment from those caused by the system.

4) The preferred model should be readily converted into rules. A rule-based approach offers several advantages. Rules have an IF THEN structure which can readily define the transition between two given states. The rules can, in turn, be used to construct a state machine. Rules can be forward chained, allowing for the automatic transversal of a state machine. The IF-THEN structure of a rule can also be translated into structured English, which in turn can be read as a requirement by any stakeholder. For future research, rules can form an expert system which allows for the querying of a set of requirements.

The Causal Component Model (CCM) achieves all of the above stated objectives, and plays a central role in both ONB approaches developed in this research. Figure 8 shows the CCM of a simple motor controller, from which we make the following observations. A CCM consists of interrelated state transition diagrams (STD), with each STD modeling the behavior of a system component. The components in this example are Motor, Switch, and Temp_Sensor.

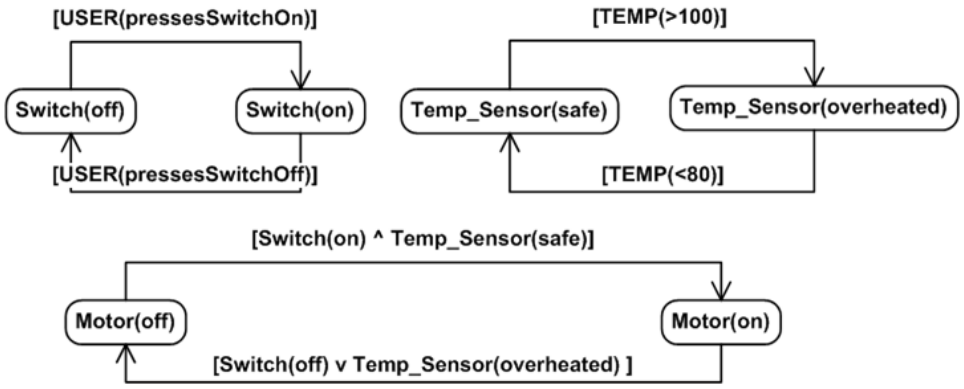


Figure 8. Example of a Causal Component Model (CCM)

Note that each STD models how that component will transition between its various component-states, designated as component(state). For example, switch(on) can transit to switch(off) and back again. The STD's are interrelated by virtue of the fact that transitions in one given STD is determined by the component state of one or more of the other STD's in the system. The CCM is designed to model a synchronous system, where each component may or may not change states concurrently, on a given

clock cycle. Thus, whether a given component in one STD changes states, depends on the state of the other components, at the time of that clock cycle. For example, in Figure 8, the motor will transit from motor(on) to motor(off) if and only if the switch is on (switch(on)) OR if the temp_sensor is sensing a safe operating temperature (temp_sensor(safe)). Note that the USER's interaction with the system is also captured; it is treated like an external component-state, and referred to as an environmental cause. By convention environmental causes are capitalized. Other environmental causes can include the ambient temperature of the system's operating environment, in this case written: TEMP(>100), and TEMP(<80).

Overall the CCM is component-state dependent, and provides a high-level view of the system. To construct a CCM the stakeholders only need to specify the state-based behavior of the system components, and how those behaviors interrelated. This keeps the system specification focused on "what" the system is supposed to do, without straying into "how" the system is supposed to achieve its behavior. The "how" is left up to the design phase, as it should be. We propose that this high level view is sufficient to expose incompleteness stemming from off-nominal behaviors, while being easy to conceive by non-technical stakeholders who should not be distracted by the details of the system's design.

Creation of a CCM

To elicit a set of requirements directly into a CCM, a stakeholder is required to follow a four step process, where each step involves the elicitation of one of the model's four artifacts: Components, Component-States, Component-State Transitions, and the Causes of Component-State Transitions. The four steps proceed as follows:

Step 1: Determine Components Elicit a set of system components, as expressed in the requirements. A component is part of a system's composition that can change states and/or cause a change in state. For example, in a motor controller system, the components can consist of {Switch, Temp_Sensor, Motor, Timer}. Note that a potential subsystem, such as the Timer, can also be

considered a component. It is up to the stakeholders to determine how they want to decompose a system into components, allowing for different abstraction levels of abstraction.

Step 2: Determine Component-States. Elicit a set of component-states for each component as expressed in the requirements. A component-state is the state of a component at a given instance of time. For example a set of component-states (expressed Component(State)) for the components of step 1 would be: {Switch(off), Switch(on), Temp_Sensor(safe), Temp_Sensor(overheated), Motor(off), Motor(on), Timer(set), Timer(expired)}

Step 3: Determine Transitions Between Component-states. Elicit the set of transitions between component-states, forming a state transition diagram (STD) for each component as expressed in the requirements. For example, some the of the transitions between the component(states) of step 2 could be Switch(off) → Switch(on), Switch(on) →Switch(off), Temp_Sensor(safe) → Temp_Sensor(overheated), Temp_Sensor(overheated) → Temp_Sensor(safe), Motor(off) → Motor(on), and Motor(on) → Motor(off).

Step 4: Determine the Component-states that cause the Transitions. To model an entire system, we must capture the way the various component(state0 transitions interact with one another. Causal relationships allow for the modeling of interactions between concurrent behaviors in a system. For example, a user pressing a switch will cause the switch to change states: USER(press) : Switch(off) → Switch(on). A change is temperature will cause the temperature sensor to change states: TEMP(>100) : Temp_Sensor(safe) → Temp_Sensor(overheated). A logical combination of switch and temperature sensor states will, in turn, cause a change in the motor's states: Switch(on) ^ Temp_Sensor(safe) : Motor(off) →Motor(on) or Switch(off) : Motor(on) → Motor(off).

Note that central to the four step process is the elicitation of component-states. Once system's component-states have been accessed, the goal is to establish how these component-states interact with one another in both a transitional and causal relationship. The intent was to have a translation process that can be performed with minimal stakeholder training. The emphasis on determining component-

states also provides the possibility of automating the translation process at some future time. This could be possibility achieved using s syntax tagger, which parses a NL sentence into nouns, verbs, adverbs, etc.

Definition of Terms and Constraints

The following is a list of the terms that will be referred to throughout this article. Included is a summary of the constraints associated with each concept express.

Components

A component is something in the system that can change states and/or cause a change in state. Typically a component can be viewed as the smallest functional unit that forms the composition of the system, but it does not necessarily need be, as it can also be a subsystem. This means that different levels of abstraction that can be modeled.

Component(State)

A component(state) is the state of a component at a given instance of time. For now we use the term state (typically nouns) versus actions (verbs), although we can treat an action as a state in the following manner. Suppose the Motor is in state of overheating. We could declare Motor(overHeating) as a Component(State). Generally there is a lot liberty in this area, and we allow whatever verbiage makes grammatical sense to the stakeholders. We view a system as a set *C* of interacting components *c*, that can each assume a component state *s*, express as *c(s)*.

System State

A system state *sys* is global state, that represents the simultaneous status of all the component states at an instance of time, and can be expressed as: $sys = (c_1(s_n), c_2(s_n), c_3(s_n), \ldots, c_n(s_n))$. To facilitate algorithmic manipulation, we can assign both component states and system states a numeric value derived from how the component states are entered into a table on the support tool. In the table, the components are entered as rows, and states are entered as columns.

The ordinal position of components and states is then used to generate a numeric value. Table 2 illustrates the numeric values (e.g. 0.2.0.) generated when entering three components and two states. From the Table 2 we derive: $c_2(s_1) = 0.1.0.$, because c_1 is in the second row (thus second position), and s1 is in the first column (thus, a value of 1). All numeric values have *N* positions, whenever there are *N* components.

Table 2. Component state entry table showing numeric value of component states

	S_1	S_2	. . . S_n
C_1	**1.0.0.**	**2.0.0.**	**. . . n.0.0.**
C_2	**0.1.0.**	**0.2.0.**	**. . . 0.n.0.**
C_3	**0.0.1.**	**0.0.2.**	**. . . 0.0.n.**

Numeric values also allows for the combining of component states, into one number, (e.g. $c_1(s_1) \wedge c_3(s_2) = 1.0.0. + 0.0.2. = 1.0.2.$) Note that the zeroes are an indication of ambiguities in the requirements, since they represent component states that have not been explicitly defined.

Component(State) Transitions

Component transitions relate two given Component(state)s (of the same component) to one another. They describe how a given component state transitions to another component state. The one hard fast rule surrounding the transitions between Component(State)s is that they only occur between the Component(State)s of the same component. One of the goal of the CCM is to model each component's behavior as it occurs in that component's domain. Thus, while ***Motor(off) → Motor(on)*** is allowed, ***StartSwitch(off) → Motor(on)*** is strictly forbidden.

System Cause

A cause for a transition that originates from within the system itself, consisting of one or more component(state)s from a component other than the component experiencing the transition. For example, the following is not allowed: Motor(stalled) : Motor(on) → Motor(off) whereas this is

allowed: Switch(off) : Motor(on) → Motor(off). Perhaps in some future iteration of the CCM we will extend its expressiveness to components being the cause of their own transitions, but for now it is restricted.

Environmental Cause

An environmental cause that originates from the system's operating environment. This can take the form of a state (or even an action) that a physical property can experience, such as TEMPERATURE(at100_degrees), or a shorthand version with a range: TEMP(>100). It can also be a USER performing a certain action, or a SENSOR reacting to a trip point. We can take more liberties with the semantics of Environmental Causes, since they need be more expressive. The convention is to capitalize environmental causes, for easy differentiating when reading the subsequent rules.

Compound Cause

A compound cause is a logical combination of two or more causes. An example is found in Figures 1, 3, and 4, namely: ***StartSwitch(on)*** ^ ***TempSensor(safe)***, which reads as ***StartSwitch(on)*** logically ANDed with ***TempSensor(safe)***. The other valid logical operator is OR. At present a NOT is not used, because the nature of CCM is the causal relationship between STD's. The absence of a specified cause is the equivalent of a logical "NOT".

Causal Relationships

To model an entire system, we must capture the way the various components interact with one another. Causal relationships allow for the modeling of interactions between concurrent behaviors in a system. At this point in its development, the CCM is used to model synchronous systems, which means that all transitions occur on a given system Transition (T). Thus, each CCM expression actually has a T term ANDed with the system cause or environmental cause, although not explicitly stated. For example the expression ***StartSwitch(on)*** ^ ***TempSensor(safe) : Motor(off)*** → ***Motor(on)*** actually has the transition term "T", and can be written as ***T*** ^ ***StartSwitch(on)*** ^ ***TempSensor(safe) : Motor(off)*** → ***Motor(on).*** However, the "T" is implied in an environmental cause.

CCM Translation Rules

CCM translation rules are the textual version of the CCM. They define the structure that a CCM would have in graphical form. They are based on the definition for a function mapping: F : X → Y, where X and Y are the same component domain. This is because each expression defines a mapping from one component state to another as a function of another component state. A transition rule can also take a symbolic or numerical form, but not combinations of the various forms. An example of a CCM transition rule is ***TempSensor(safe) : Motor(off) → Motor(on),*** which reads: the Motor will transition from off to on if the TempSensor is in a safe state.

To develop the rule notation, we built upon the function notation that specifies the mapping of elements between two domains, commonly expressed as F : X → Y . The reason for doing so is because a CCM rule must be able to represent a mapping between two component(state)s (C(s)). Furthermore, this mapping must be dependent on some transition (function) F. On the left-hand side of Figure 9, the function notion, F : X → Y , specifies that function F maps element "a" from domain X to element "b" in co-domain Y .

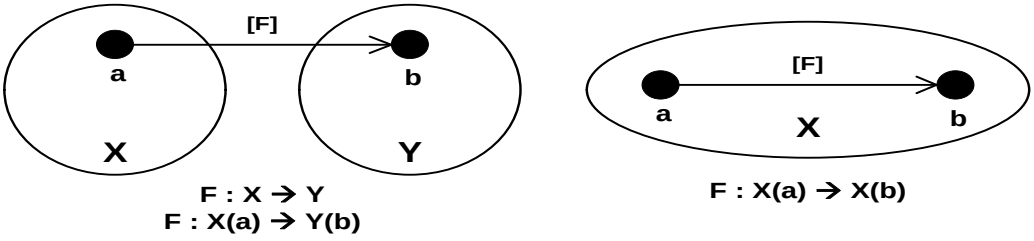


Figure 9. The CCM formal notation is based on function notation

We use the notations X(a) and Y(b) to denote that "a" and "b" are within the domain of components X and Y respectively. For our purposes, we will now assume that X = Y and restrict the function F to within one component. The reason for doing so is because we want the CCM notation to map transitions within the same component, and not between two separate components. Thus, on the right-hand side of Figure 9, we redefine the function F as a transition that maps "a" to "b" within

component X. We replace X with C, "a" with S1, and "b" with S2, which represents a component containing states S1 and S2, and replace F with inter-component transition T; this gives us T : C(S1) →C(S2). In a real world example, a component would likely have more than two states. These various states would be interconnected via a set of transition functions.

The left hand side of Figure 10 shows an example of a component C1 that has three states (S1, S2, and S3), with three transitions (T1, T2, and T3). Since the CCM is geared toward embedded systems it is important that we be able to model concurrency. To do so, we need the ability to simultaneously represent more than one component, and their transitioning states. Figure 10 shows two component (C1 and C2), and their corresponding states. In Figure 10, each component has their states interconnected in a state transition diagram (STD). Each STD can act independently and thus concurrently with the other components. Components C1 and C2 of Figure 4, each has three transitions, labeled F1, F2, and F3.

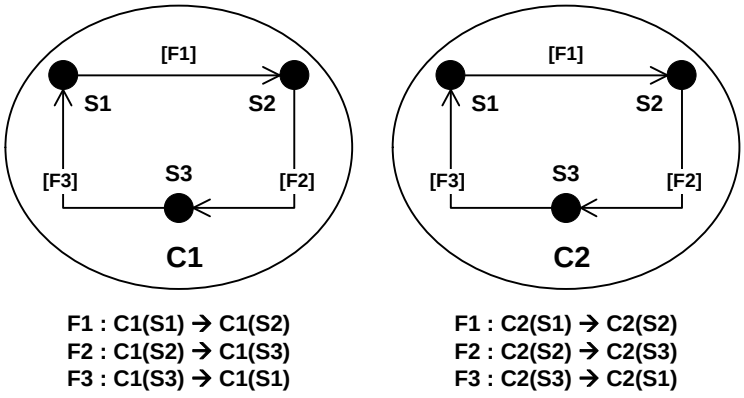


Figure 10. Three transitions within a component (C1). C2 is a second component

A transition cause can be a single component(state) or a logical combination of component(state)s. By using component(state)s the cause of a transition we establish the casual relationship between the component(state)s of two components within the system boundary. The CCM requires that we view a set of requirements as a collection of statements that describe how various system components interrelate with one another on a causation basis [161]. The CCM consists of a set of

interrelated state transition systems with each transition system modeling the behavior of a specific system component. Each component's behavior occurs concurrently with the other components, and the CCM models how one component's behavior affects another component's behavior. Figure 11 shows an example of a simple CCM where a switch turns on a motor, and 3 seconds later, the motor turns off.

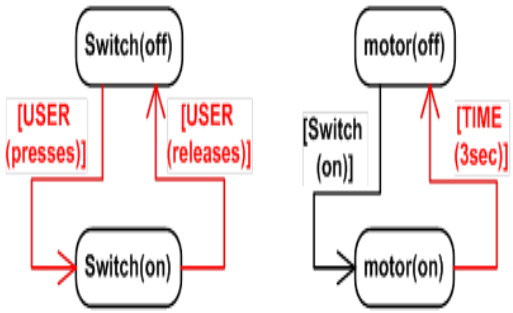


Figure 11. Simple CCM example

Our approach begins with the manual translation of requirements into translation rules, which we now define. Let EC be the set of all possible causes originating from the environment, and $ec \in EC$ be a given cause. Let CS be the set of all component states, and $cs \in CS$, a given component state. We define a set of Environmentally Caused translation Rules (ECR) : $ECR = \{ecr \mid ecr = cc : cs_p \rightarrow cs_n, cs_p \in C_n(Sp), cs_n \in C_n(S_n)\}$, where $(cc = ce, ce \in Ec)$ or $(cc = ce\ cs_2 \ldots cs_n, cs_n \in C_n(S_p), cs_n \in C_n(S_n))$. Note that the component portion of the component state must be the same for both (cs_p, cs_n). $ecr : cs_p \rightarrow cs_n$ is a mapping notation which reads $ecr \in ECR$ causes component c to transition from its present state sp to its next state sn. Recall that, component-states, CS_p and CS_n can also assume a numerical value, yielding, in a system of three components, for example: $ec : 1.0.0. \rightarrow 2.0.0.$

We define a set of System Caused translation Rules as: $SCR = \{scr \mid scr = sc : cs_p \rightarrow cs_n, cs_p \in Cn(S_p), csn \in C_n(S_n)\}$ Where $(sc = cs_x, cs_x \in C_n(S_p), cs_x \in C_n(S_n))$ or $(sc = cs_1\ cs_2 \ldots cs_n, cs_n \in C_n(S_p), cs_n \in C_n(S_n))$. Numerically, a member of ECR can have the form $ec : 0.1.0. \rightarrow 0.2.0.$, whereas a member of SCR can be $0.0.3. \wedge 1.0.0. : 0.1.0. \rightarrow 0.2.0.$, and further simplified to $1.0.3. : 0.1.0. \rightarrow 0.2.0.$

Absorption and Propagation Operations

In numerical form, a translation rule contains zeroes which results in ambiguous multiple interpretations of a given rule. Translation rules must be made explicit through what we call expansion. Before the translation rules are expanded, we require that they be subjected to a process called absorption which absorbs the cause on the rule's right-side. For members of ECR, rules in the form of *ec* $\wedge$ *cxs : cys*$_p$ $\rightarrow$ *cys*$_n$, must be converted to *ec : cxs* $\wedge$ *cys*$_p$ $\rightarrow$ *cys*$_n$. For SCR members, rules in the form of *tc* $\wedge$ *cxs : cys*$_p$ $\rightarrow$ *cys*$_n$ become *tc : cxs* $\wedge$ *cys*$_p$ $\rightarrow$ *cys*$_n$, because component--states ANDed with a cause are treated as preconditions for that given transition. As such, they are ANDed with the present states.

When the rules are in numerical form, absorption becomes a matter of shifting digits from the cause to the present state; e.g., *tc* $\wedge$ *0.0.3.* $\wedge$ *1.0.0.* : 0.1.0. $\rightarrow$ 0.2.0. becomes *tc : 1.1.3.* $\rightarrow$ *0.2.0.* The propagation process further reduces the number of zeroes by shifting digits from the present state to the next state. Thus, *tc : 1.1.3.* $\rightarrow$ *0.2.0.* becomes *tc : 1.1.3.* $\rightarrow$ *1.2.3.*, because every translation rule defines a change in only one component--state; therefore, we cannot assume that other component--states have changed at the same time.

CCM Symbolic Rules

A CCM expression can be translated into a symbolic form by using letters to represent components, and numbers to represent a component's state. The assignment of letters and numbers is based on the order in which components and states are entered into a CCM entry table.

For example, if the following components are entered in the following order: ***{StartSwitch, TempSensor, Motor}***, then the StartSwitch is represented with an A, the TempSensor with a B, and the Motor with a C. Thus, ***{StartSwitch, TempSensor, Motor} = <A, B, C>.***

Let us further assume that the three components each have two states resulting in the following list of component states: ***{StartSwitch(off), StartSwitch(on), TempSensor(safe), TempSensor(overHeat), Motor(off), Motor(on)}.*** Since each component has two states, the states are

numbered 1 and 2 according to the order that they appear in, resulting in the symbolic list ***{A(1), A(2), B(1), B(2), C(1), C(2)}***. If we apply these symbols to the following CCM expression: ***TempSensor(safe) : Motor(off) → Motor(on)***, we get the symbolic expression ***B(1) : C(1) → C(2).***

CCM Numeric Representation

A CCM symbolic rule can be translated into a numeric representation for automatic analysis and manipulation. The numeric values are derived from the ordinal values of the components and their respective states, after they are entered into the support tool entry grid. Going from symbolic to numeric the ordinal values of the components are taken from the ordinal position of the alphabet. For example, the following symbolic expression: ***T ^ B(1) : C(1) → C(2)*** translates into the numeric representation ***T : 011 → 012***. For the moment note that ***T : 011 → 012*** indicates that at transition T, we don't care how component A transitions (indicated by a 0), component B will remain at state 1, and C will transition from state 1 to state 2. As we will see in section 5, there is useful information encoded in the numeric representation. Figure 12 shows how a CCM translates to symbolic and numerical representation. Again, this translation is perform within a support tool, and is typically kept transparent to the stakeholders. The primary purpose of a numeric representation is to facilitate the automatic analysis of the CCM. A secondary use is for the creation of a Kripke structure for use with an explicit model checker. Lastly, both a symbolic and numeric represent can serve as a framework for further research into ways of analyzing requirements that have been translated into a CCM.

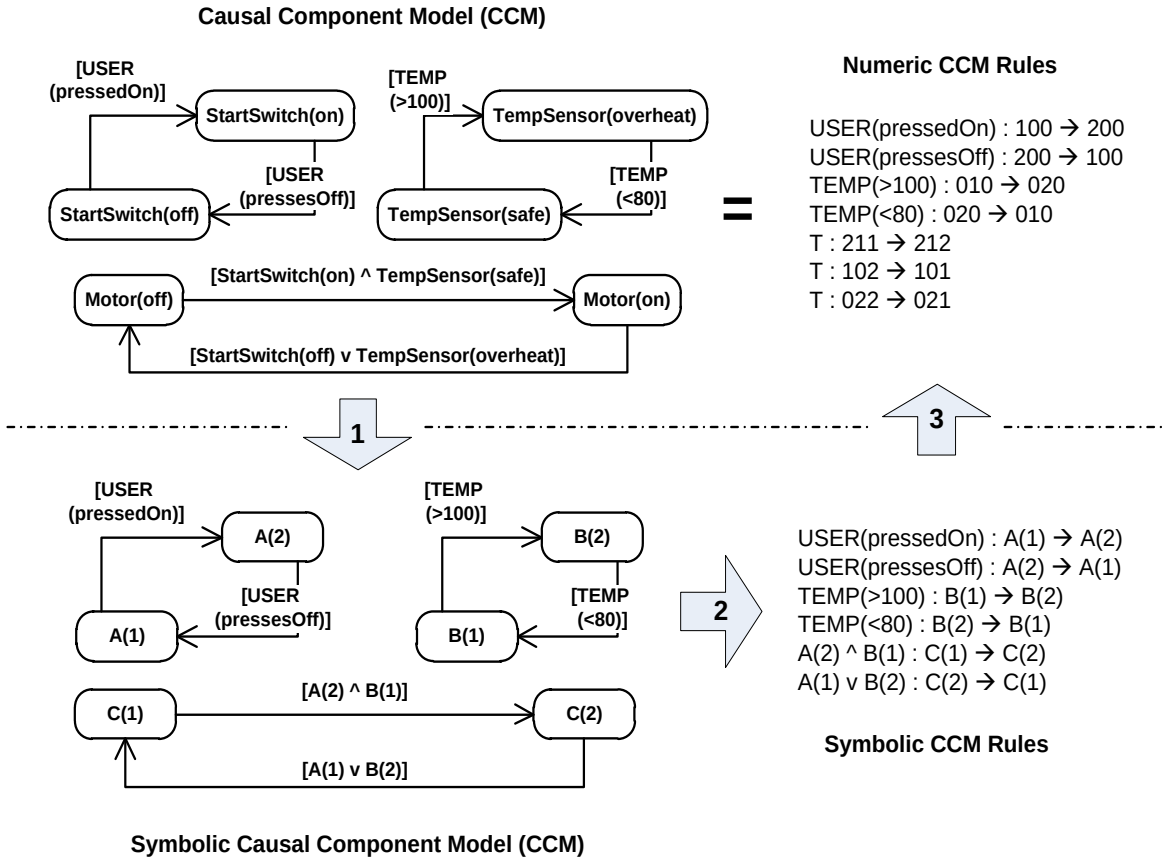


Figure 12. Symbolic and numeric representation of CCM

The upper half of Figure 12 shows the motor controller CCM (left hand) and its numeric expression (right hand). The bottom half of Figure 12 shows the three symbolic intermediary steps between the CCM and its numeric version.

Labeled Transition System (LTS)

For our purposes, We define a state transition system as a tuple *{S, R}* where S is a set of states and R is a set of binary relations $S \times S$ over *S*, known as transitions. For example, for $\{s_1, s_2\} \in S$, we say that there is a transition from s_1 to s_2 expressed as $(s_1 \rightarrow s_2)$. Figure 13 shows a LTS (labeled B) that was created from a CCM (labeled A). The CCM consists of three components *{StartSwitch,*

TempSensor, Motor}, and four component states *{StartSwitch(off), StartSwitch(on), TempSensor(safe), TempSensor(overheat), Motor(off), Motor(on)}*.

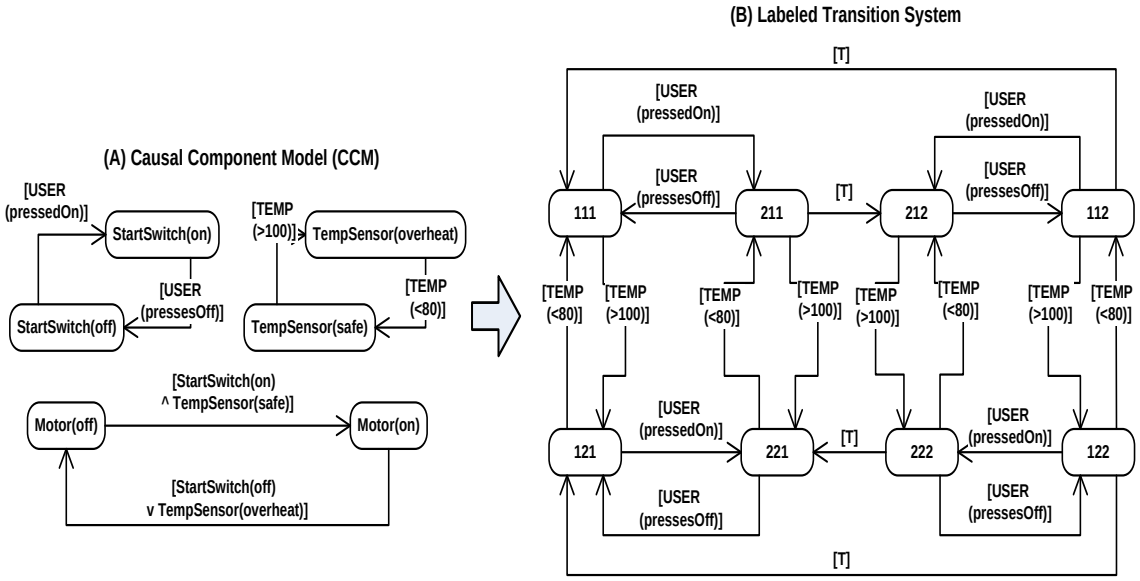


Figure 13. Comparison of CCM (A) and a Labeled Transition System (B)

Figure 13 shows that the CCM can be converted into a Labeled Transition System (LTS) in which the nodes display a snapshot of the system's global state at a given instance of time. Each transition in a LTS shows how the system goes from one global state to another. Each of the system's component-state is factored into each global system state. For example, the initial system state of our on-going motor controller example might be *{StartSwitch(off), TempSensor(safe), Motor(off)}*. Note that the component states of all three components are combined into the system state. In Figure 13B the system states are represented in numeric form, thus the initial state *{StartSwitch(off), TempSensor(safe), Motor(off)}* is shown as 111. To continue the example, the LTS of Figure 13 shows a transition from *111* → *211* when the USER pressed the StartSwitch ON. In others words, the system transitioned from *{StartSwitch(off), TempSensor(safe), Motor(off)}* → *{StartSwitch(on), TempSensor(safe), Motor(off)}*. A LTS is an effective means of analyzing a system that has two or more components behaving concurrently. For a system with N number of components, their concurrent behaviors can be analyzed in

any possible combination. For example, from the LTS of Figure 14 we can trace the various paths that go from {StartSwitch(off), _, Motor(off)} to {StartSwitch(on), _, Motor(on)}, while ignoring the behavior of the TempSensor (one such path 111 → 211 → 212, the TempSensor number (010) never changes from 1. In a typical set of requirements the LTS structure would be too large to display, but not too large to create internally within the support tool. Therefore, the analysis of the LTS occurs internally, after the tool has converted the CCM to a LTS. Note, from Figure 14, that the CCM is first converted into its numeric representation (step 1) and then the zeroes are filled in with every possible system state combination to form an expanded set of expressions that specify every possible transition path in the system (step 2). These path specifications are combined internally to create the LTS (step 3).

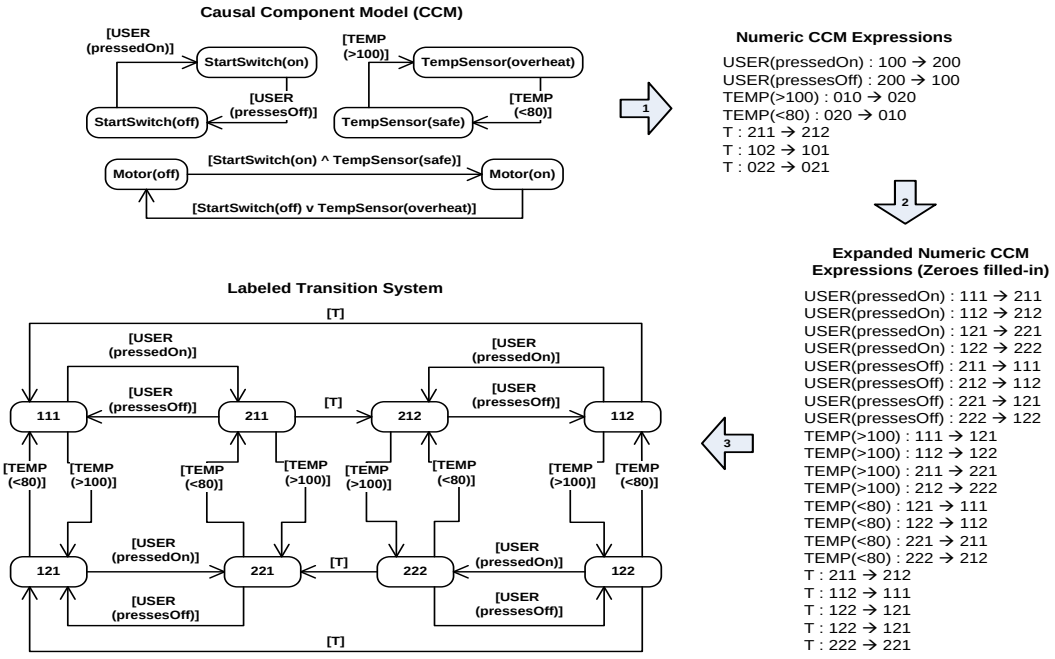
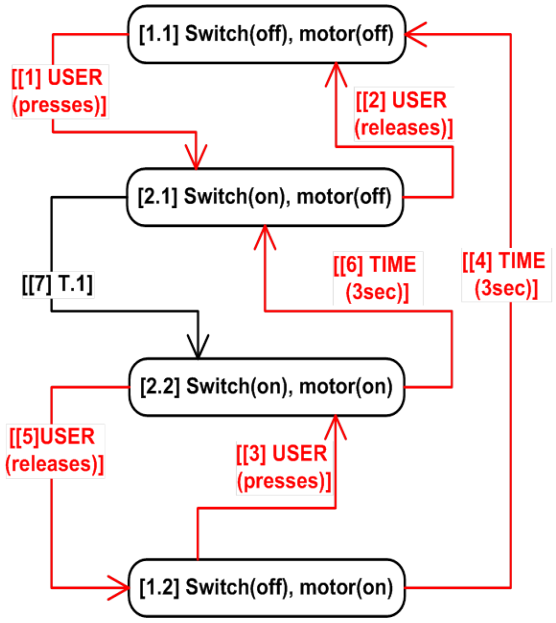


Figure 14. Conversion of CCM to Labeled Transition System

State Profiles

There are four variables that apply to a given system state. The variables are as follows: the number of in-degrees, the type of in-degree, the number of out-degrees, and the type of out-degree. The number and type of in-degrees are expressed in two sets, TID and EID. TID is defined as T transition In-

Degree; it is the set of in-degrees for a state that comes from paths labeled "T.x." EID is defined as Environment In-Degree; it is the set of in-degrees for a state that comes from paths with an environmental cause. The number and type of out-degrees are expressed in two sets, TOD and EOD. TOD is defined as T transition Out-Degree; it is the set of out-degrees from a state that comes from paths labeled "T.x." EOD is defined as Environment Out-Degree; it is the set of out-degrees for a state that comes from paths with an environmental cause. We can combine information about the four variables, as it applies to each system state, using what we call an ID-OD matrix. Figure 15 shows the ID-OD matrix for an example of a LTS.



System State ID-OD Matrix

State	EID	EOD	TID	TOD
1.1	{2, 4}	{1}	{}	{}
1.2	{5}	{3, 4}	{}	{}
2.1	{1, 6}	{2}	{}	{7}
2.2	{3}	{5, 6}	{7}	{}

EID – Environment In-Degree
EOD - Environment Out-Degree
TID – T-cycle (system) In-Degree
TOD - T-cycle (system) Out-Degree
$\{P_1, P_n\}$ – Set of labeled paths
{ } – Empty set

Figure 15. Example of state profiling an LTS

The ID-OD matrix shows the sets of in-degrees and out-degrees for all the system states. For example, state 2.2 has a set EOD of two out-degrees from the two environmental paths labeled 5 and 6. As we will see in a later section, the TID, EID, TOD, and EOD (and their cardinal numbers) are use to profile a given system and determine whether that state plays a role in an off-nominal behavior.

Example Application of CCM

Figure 16 shows an example of a Causal Component Model for a set of requirements describing a simple motor controller. The arrows in Figure 16 show the progression from NL requirements to CCM translation rules.

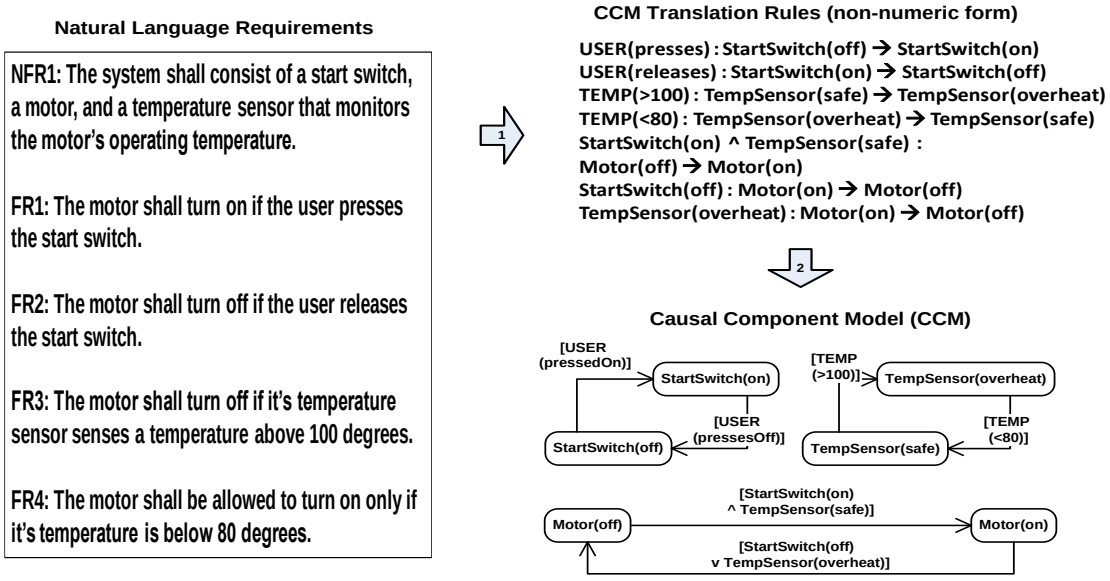


Figure 16. Example of CCM derived from a set of NL requirements

From the CCM of Figure 16, we make the following observations. Note that each system component mentioned in the requirements *{StartSwitch, TempSensor, Motor}* is represented in the CCM. Each component's possible state is also represented, using the notation Component(State).

The StartSwitch has two states "on" and "off", which is written as *{StartSwitch(off), StartSwitch(on)}*. The other two sets of component states are *{TempSensor(safe), TempSensor(overheat)}* and *{Motor(off), Motor(on)}*.

Each set of component states are used to create a State Transition Diagram (STD) that describes the behavior of that given component.

Note that in the case of the Motor's STD, the transitions are caused by the component states of both the StartSwitch and the TempSensor. This is where the concept of a causality between components comes in (thus the name Causal Component Model). A CCM consists of a set of interleaving STDs, where each STD describes a system component behavior, and each component's behavior depends on the behavior of the other components. A CCM describes a synchronous system, in which all the components can transition concurrently, as enabled by the states of the other components in the system. The state of a CCM at any given time Tn (considered a snapshot of the CCM) is the combination of all the concurrent component states at that given time. The next state the CCM will transition depending on the present CCM state. Thus the CCM models the independency between concurrent component behaviors.

The CCM is a very high level, abstract, representation of a system, which focuses on specifying what a system is supposed to do, without concern over the details of how it is supposed to do it. Note that the transitional causes of the StartSwitch and TempSensor. are capitalized by conversion to signify that these causes are from the system's operating environment. This ability of the CCM to differentiate between causes occurring from the environment (outside the system) from the causes occurring within the system allows a more complete formal examination of the CCM. Figure 16 shows that the creation of a CCM from a requirements document and/or direct elicitation involves a four step process that centers around the determination of component states and their interactions with one another.

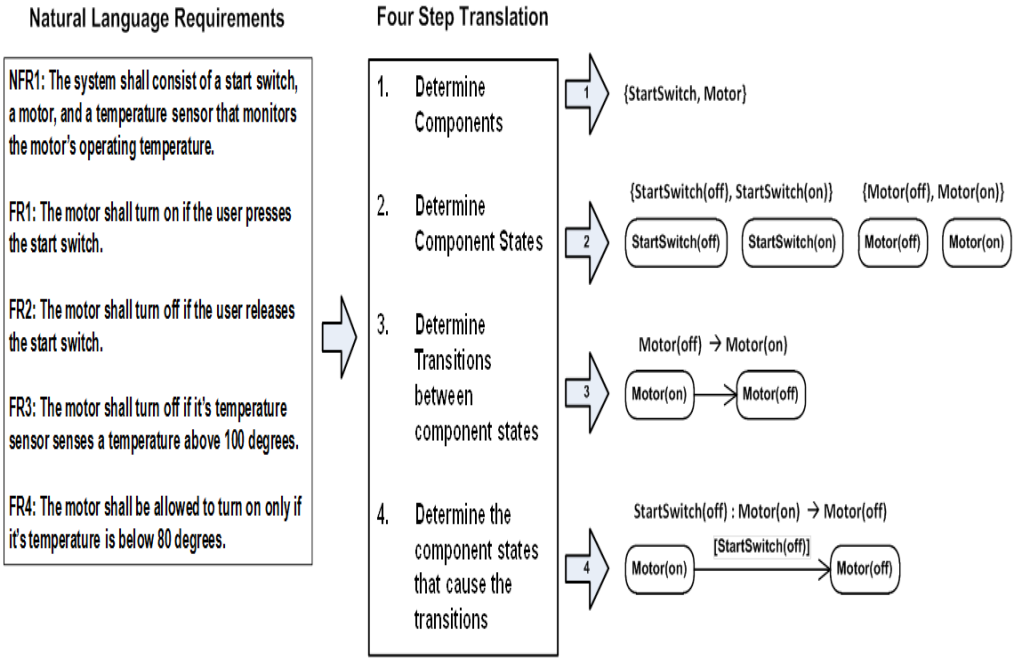


Figure 17. Example of the four step CCM creation process

The focus on component states was conceived in order to achieve a high degree of cognitive friendliness. With minimal, if any, training, determining the necessary system components, and their respective states, is a task within the ability of most stakeholders be they technical or non-technical. Domain experts should also have the ability to surmise what components would make up a system-to-be, since a component is a fundamental building block of a system, and people tend to build things from lower to higher levels of complexity. Note from Figure 17 that once a set of component states are determined (Step 2), the remaining task is determining how those component states interact with one another (Steps 3 and 4), as dictated by the desired system behaviors. The actual gathering of data for each step is performed using an entry table, as shown in Figure 18.

The CCM entry table of Figure 17 is a mockup of the table used in a support tool. Steps 2 and 3 show that the approach takes the entered data and create a CCM (step 2) or a set of CCM expressions.

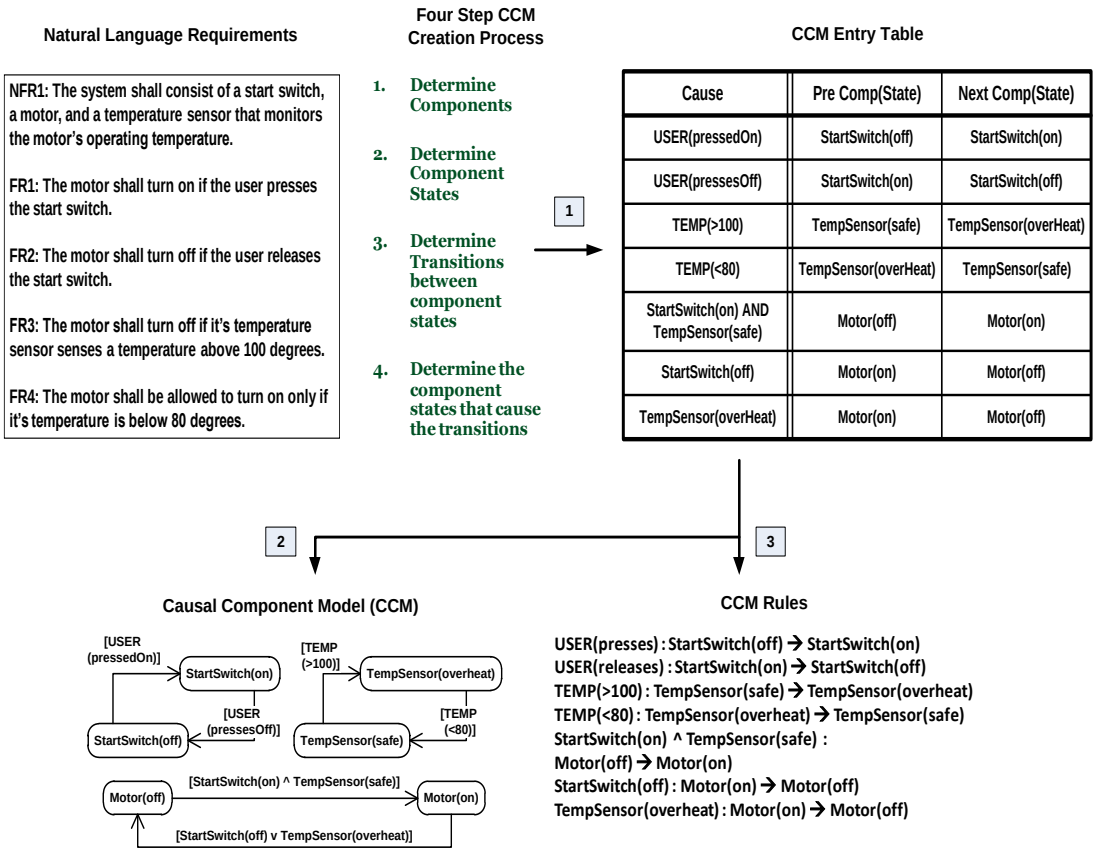


Figure 18. Example use of a CCM entry table

The conception and use of the CCM rules will be addressed in greater details in chapter 6. For now, the reader should note that CCM expressions are a notational (non-graphical) means of defining a CCM that allow for the converting of a CCM to a Labeled Transition System, a Petri-net, and a symbolic/numerical representation used for property checking. The expressions can also be used to create a NuSMV script for use with a symbolic model checker. The expressions have a IF-THEN structure that facilitates the mapping into the language used in NuSMV. Finally, the CCM expressions can also be used to covert a CCM into a set of rule-based NL requirements, for documentations purposes.

Levels of Abstraction Possible with the CCM

The translation rules, and associated CCM, represent a system at a high level of abstraction, defining what a system is supposed to do. The component--state can be used as both nodes and edges in the set of interrelated transition diagrams that form a CCM. Consequently, we can model the system at various levels of granularity by varying the number of component--states used to represent the system. This ability, in turn, helps manage the potential state-explosion problem. There are four possible (but not exclusive) levels of abstraction that can be modeled with the CCM; they are labeled A, B, C, and D in Figure 19. With all four examples, the *switch* is represented using the component(state) notation: A and B use *{Switch(off), Switch(on)}* while C and D use *{Switch(UserPresses), Switch (UserReleases)}*. A and B represent the *switch* as a node while C and D represent the switch as the cause of a transition.

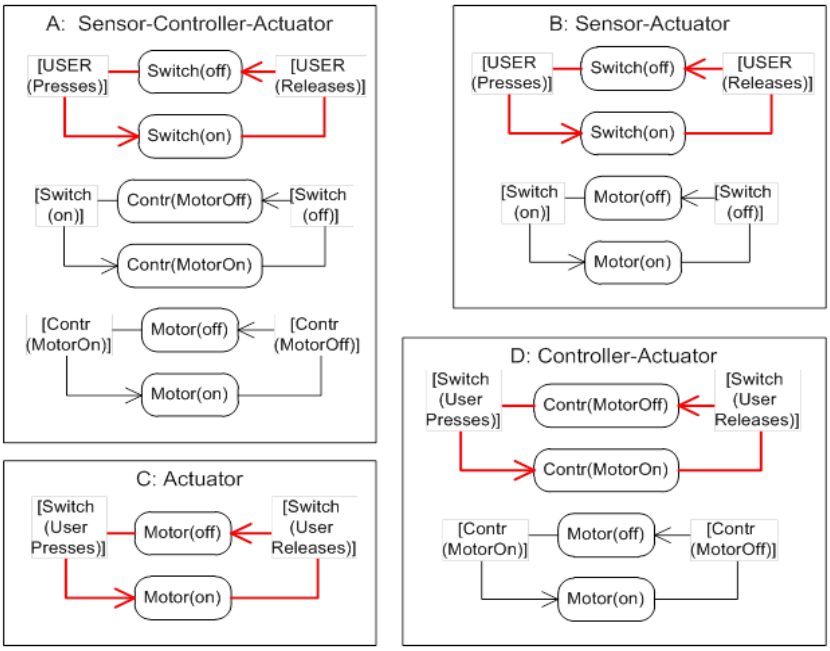


Figure 19. Levels of abstraction possible

Each example in Figure 18 shows a tradeoff between the granularity of the representation and the number of potential system states. Example A results in 6 system states (2 x 2 x 2). At the other extreme, Example C offers less detail but only produces two system states. Note also that, with Example

C (as well as D), the switch has been consolidated with the *USER* so that both the user and switch are still represented in the model. Thus, in Examples C and D, we have the ability to model the switch component as a cause and to avoid the additional system states that the *switch* would produce if it were a component. Thus, when trying to manage a possible state explosion, the preferred modeling scheme is either Example C or D, whereas input sensors are treated as a cause.

Overview of the Approaches to the ONBP

The research pertaining to this dissertation resulted in two approaches to addressing the ONBP. 1) We made model checking itself cognitive friendly by using CCM as a model checker's front-end. We refer to this as the Property Checking Approach (PCA). 2) We developed a cognitive friendly approach that achieves the end result of model checking, by extending CCM into a rules-based ONB checker; referred to as the State Profiling Approach (SPA). Since both approaches are state-based and analyze functional requirements using path traversal, they are well suited to the problem of exposing susceptibility of off-nominal behaviors in a set of embedded requirements.

Property Checking Approach

The *Property Checking Approach* (PCA) uses the CCM as a front end to a model checker. The CCM converts a set of requirements into a set of IF THEN statements that forms the model specification portion of a NuSMV model checking script. Part of the CCM information is also used with temporal logic templates, to create the script's temporal logic properties. The model checker can answer questions such as "Can a given undesired system state be avoided regardless of off-nominal behaviors from the environment." If the answer is no, the model checker provides a counter-example which shows the ONBs that result in an undesired state. The results are reported to the stakeholders, who then addresses the problem by modifying the requirements accordingly. PCA proved to be effective in exposing ONBs, however it still required a considerable amount of manual post analysis on the model checker's counter-example, in order to deduce how to correct the exposed problem.

We felt that ONB corrections could be deduced more easily if we were not restricted to counter-examples as our only source of information, since potentially useful metrics were confined to the internal operation of the model checker; metrics such as the number of global system states that contain an undesired combination of component states; all traversal paths other than just counter-examples resulting from failed properties; whether or not an undesired state is recoverable by the system. This motivated the development of a more open frame approach referred to as *State Profiling.*

State Profiling Approach

If we define an ONB as an environmentally induced behavior that results in a non-system recoverable undesired state, then exposing a system's susceptibility to ONBs requires the following items:

1) The identification of transitions between global system states.
2) Differentiating the transitions caused by the environment from those caused by the system.
3) Identifying those environmental transitions that lead to an undesired state, and the lack of those system transitions that would recover the system form the same undesired state.

Because a NuSMV model checker does not explicitly represent (or differentiate) transitions in the resulting counter-examples, the NuSMV is not ideally suited for exposing ONB susceptibility.
The reason for the lack of suitability is primarily due to the Transition System (TS) that a model checker (whether explicit or symbolic) uses internally to represent the model. The TS in question models the transitions among global system states without indicating an explicit cause of those transitions. Transitions occur on the basis that a given global state is the precondition of another global state (the transition cause is thus implicit in the global state).

Furthermore, the temporal logic properties define state transitions, without regard to the cause of those transitions. That is to say, a temporal logic property is concerned with the eventual reachability of a given state, regardless of what transitional causes may result in that state being reached.

A way to model state transitions while explicitly differentiating the type of cause that results in those transitions is needed. A rule-based representation of a system achieves this, because implicit in the rule statement is the element of transition causality, as expressed in the IF-THEN semantics of the rule.

For example, the CCM rule: "USER(setsTime) : Timer(elapsed) → Timer(set) " reads, IF the USER sets the time, THEN the timer will go from elapsed to being set. The cause of the Timer's transition from elapsed to set is explicitly stated.

Rules could then be classified as those describing environmental causes versus those describing system causes. Thus, the SPA rule-based approach achieves the following items:

1) Models system behavior as a series of transitions between global system states.

2) Explicitly identifies the transitions between global system states.

3) Differentiates the transitions caused by the environment from those caused by the system.

4) Identifies those environmental transitions that lead to an undesired state.

5) Exposes the lack of those system transitions that would recover the system from a given undesired state.

In order to make environmental causes explicit, we eliminated the model checker, and extended CCM into a dedicated, open-frame, rule-based, requirements ONB checker. The *State Profiling Approach* (SPA) starts with the CCM represented as a set of translation rules containing ambiguities from the NL requirements. The use of rules allows for the direct interpretation of results, allowing for not just the exposing of ONBs, but for information suggesting how to avoid them. Algorithmically, every possible interpretation of those rules are produced, resulting in a set of explicit rules that describe both intended and unintended behaviors. Given a set of stakeholder-defined undesired states, the rules are screened to determine if any of the rules result in an undesired state. If so, those undesired states are profiled to determine if they are non-recoverable.

Once, a set of non-recoverable undesired states has been exposed, the cause of those undesired states are determined and reported to the stakeholders, who then address the problem by modifying the requirements accordingly. The rules also allow for a more direct exposure of ONBs, as opposed to having to analyze a system's entire state space, as in the case of the model checker. This helps mitigate the state explosion problem, reduce computation time. Finally, the use of rules also allows for the exposure of incompleteness and inconsistencies in a set of requirements, without having to develop additional model checking temporal properties. To evaluate our approach we performed a case study using a commercial mini-excavator and exposed two instances of ONBs with a suggestion on how to fix them.

Fundamental Differences Between Approaches

The fundamental difference between PCA and SPA is the scope of their usefulness. PCA is a dedicated technique designed to specifically address the ONBP, whereas SPA is an analysis framework that can be applied to the ONBP as one of many possible applications. In the long term, SPA can be a more powerful analysis technique than PCA for the following reasons. Since the model checker is essentially treated as a block box, one can only work within the constraints of an input script, and a counterexample output, with all forms of analysis framed strictly as a reach-ability problem. Being a black box, the model checking algorithm is internal to the model checker, and cannot be directly analyzed using properties other than temporal. By contrast, SPA is not a black box, but rather provides direct access to all its algorithms. This makes PCA a dedicated technique, while SPA is a general purpose framework. For this reason SPA offers a greater contribution to long term ONB research, than PCA. Table 3 shows the key differences between PCA and SPA.

Table 3. Comparison between the property checking and state profiling approaches

Property Checking Approach	State Profiling Approach
Needs entire system state space	Needs only undesired state space
Produces a counter example that must be interpreted via CCM simulation.	Can produce a constraint requirement.
No access to internal LTS.	Rules are accessible; can also be used to produce a Petri net.
No information beyond a counter-example from a failed property.	Rules can be directly analyzed for additional information. Traversal information can be obtained whether or not property holds.
Internal model cannot be partitioned.	Rules can be used to model only E causes and/or T causes.
Produces no additional metrics beyond pass/fail and counter-example.	Can produce metrics such as numbers of E rules, number of T rules, number of occurrences of a given undesired state.

CCM PROPERTY CHECKING APPROACH (PCA)

Overview

The CCM can be used to address the ONBP using two different approaches: Property checking and state profiling. This section describes the former of the two. The Property Checking Approach (PCA) uses the CCM as a means of formalizing a set of requirements into a set of rules, whose properties could then be examined. Research was conducted into the possibility of directly property checking the rules derived from a CCM, however it was decided to utilize the well-established means of property checking, known as model checking.

The properties in question are related to the reach-ability question, specifically whether an undesired system state can eventually be reached. A secondary question becomes, if an undesired state can be reached, what combinations of environmental causes result in that state? Both of these questions can be answered using model checking, which is designed to address reach-ability problems, and can provide path traces, in the form of a counter-examples. Thus, in the property checking approach, CCM serves as a front end to model checker. Figure 19 shows a detailed block diagram that emphasizes the user interface portion of the property checking approach.

In Figure 19 we have highlighted in gray the CCM related elements that serve as a model checker front end. The CCM provides a cognitive friendly interface between the stakeholders and the model checker, facilitating the creation of both the model specification and the temporal properties. Model specifications for the model checker is created from the CCM Expressions {5}, and not from the Numeric Representation {6}, which is not used with the model checker.

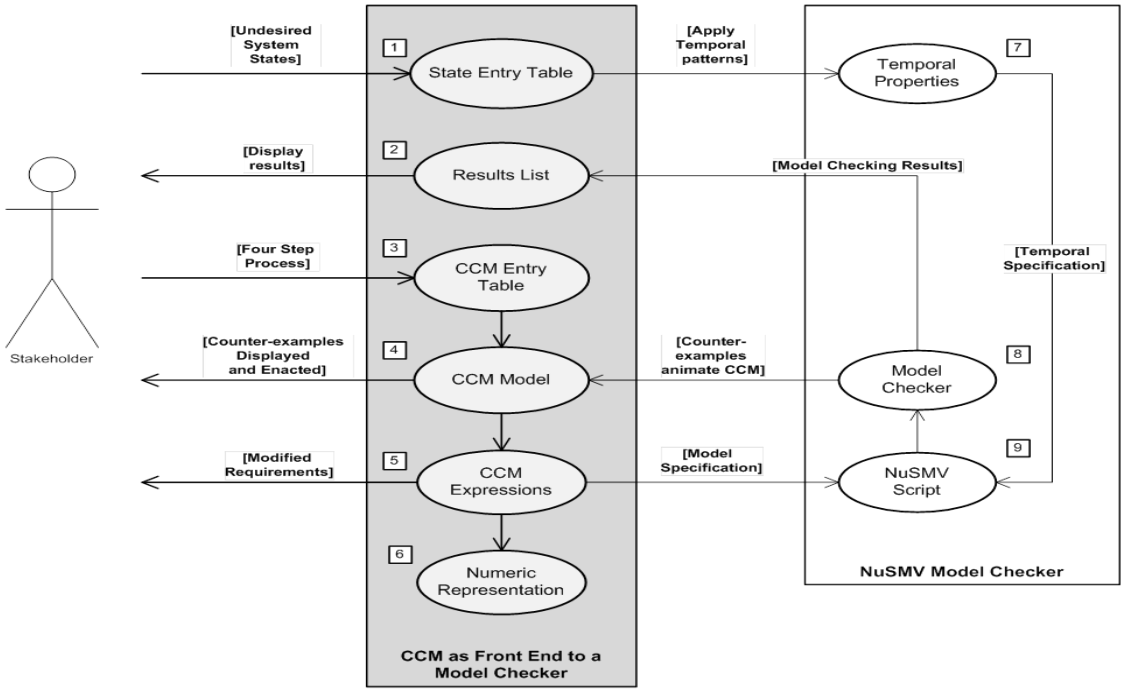


Figure 20. Overview of the Property Checking Approach (PCA)

The CCM expressions use an IF-THEN structure that maps directly to the IF-THEN verbiage used in the NuSMV script. The Temporal Properties {7} are produced using temporal property templates [Dwyer], by inserting the undesired system states in the State Entry table {1}. Other properties are automatically created by tool, by using information gathered in the CCM Entry Table {3}. Both the model specifications and the temporal specifications are used to create the NuSMV script, which in turn is used as the sole input to the NuSMV model checker {8}. The model checker determines if the model specified satisfies the temporal properties. If so, it returns a true for each property satisfied. If false, the checker returns a counter-example; a path traversal through the model that shows the reason why the property failed. The counter-example is used to animate the CCM {4}, so as to simulate the behavior revealed in the counter-example. Thus, the CCM serves as both an input interface to create the NuSMV script, and as a simulation display to help stakeholders interpret the resulting counter-examples.

PCA Preliminaries

An established means of formal verification is model checking, which is illustrated in Figure 20. Model checking is a formal method technique used for the verification of embedded systems. Its focus is primarily in the verification of hardware, but it has been used to verify software as well. Model checking can be used to verify the system-to-be requirements and/or the system's implementation.

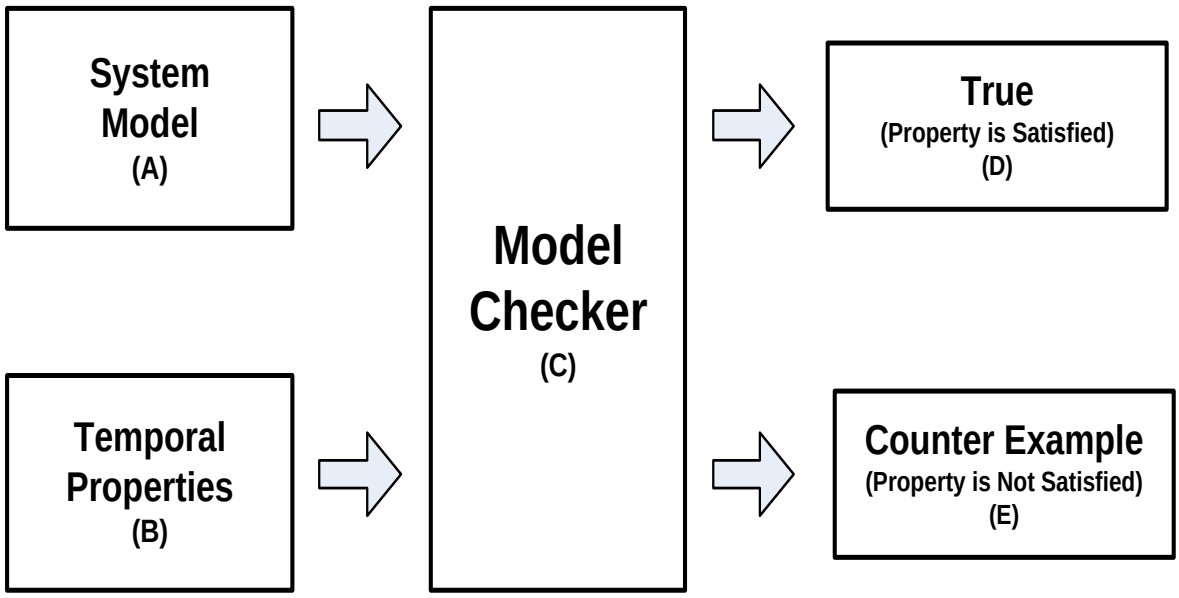


Figure 21. Overview of model checking

Figure 20 shows the key elements in either the explicit or symbolic model checking process. Referring to Figure 20, the system model (A) is examined by the model checker algorithm (C), to determine whether it satisfies a set of desired propositional temporal properties (B). The properties are specified temporal logic which allows for the representation of time, not is absolute terms, but in temporal ordering. For example, a temporal specification will specify that a certain system state must eventually be reached in the future, and in a certain order compared to another system state.

If a property is satisfied, the model checker returns “True” (D), if not satisfied, the model checker provides a counter example (E), that is, a path that shows why the property was not satisfied. Model checkers are very prone to state explosion, and several approaches, such as BDD, have been developed to reduce the number of states in a model while still retaining the behaviors being defined. Subsequently, there are various forms of model checking, each trying to manage the problem of state explosion, such as partial order reduction [144], bounded model checking [121], and symbolic model checking [127]. Among these approaches, symbolic model checking that uses a symbolic model verifier (SMV) has been a de facto model checking approach [127].

There are two major variations of model checkers, explicit and symbolic. In an explicit model checker the system is modeled by a type of finite state machine (FSM) known as a Kripke structure.
A Kripke Structure represents a concurrent system’s behavior as a graph, with each node represents a global system state, which is a snapshot of all the system’s concurrent states at any given time. An explicit model checker examine one state at a time, performing depth-first traversal of the Kripke from state to state. A symbolic model checker can avoid building an entire Kripke, and instead create a reduced model, using proportional logic, that captures the system’s behavior, without having to model every state variable combination in the system; using a symbolic algorithm, sets of states are examines at once. Symbolic checker, such as NuSMV use a rule-based script to capture both the model and the temporal properties.

Explicit model checking is said to be better suited for hardware verification, whereas symbolic is considered better fit for verifying software. However, model checking is geared toward using Boolean predicates, which can represent some limitation with checking software, that uses a use a wider variety of data types. The same can be said about model checking requirements that describe behavior in a non-Boolean manner. The CCM based approach proposed here will address this issue by translating requirements into a Boolean representation that can be symbolically manipulated using propositional logic.

Temporal quantifiers can then be added to the propositional logic in order to specify temporal properties using the same CCM notation. A detailed examination of model checking is beyond the scope of this dissertation. For our purposes we are only concern with the fact that model checking can be used as a form of property checking a set of requirements that has been initial modeled as a CCM. The CCM rules maps directly to the IF THEN structure of a symbolic model checking script, such as NuSMV. The model checker is treated as a block box, to which we provide the model and temporal properties and the model checker confirms whether those properties hold.

Support Tool Implementation

The PCA approach was implemented into a support tool, called CCMChecker, allowing us to conduct case studies with various sets of requirements. The tool is a Windows application that was implemented in Visual C# and uses a third-party graphing library called Graphviz to generate a graphical representation of the CCM. Among the various features the tool offers, it enables the entering of component--states, generates translation rules, and automatically expands those rules. The tool can even display a color-coded Labeled Transition System (LTS) from the expanded rules. The colors differentiate the environmental causes (as red transitions) from the system causes (in black). Stakeholders can manually simulate the CCM behavior by clicking the environmental causes associated with that CCM. Other displays include a LTS that only has the states transitioned by the environmental causes and those states transitioned by the system. The tool generates the results of each step in the form of an unformatted report.

The tool is designed in such a way that allows for future expansion and modifications. This is achieved by partitioning functionalities into reusable components (objects) and global dedicated data structures (structs and lists). The tool's architecture uses several data structures to contain the data in its various forms as it is processed from step to step. These include, but not limited to, separate data structures for translation rules, the NuSMV script statements, and the temporal logic properties.

The data structures are shared, on a read/write basis, between data processing classes, which are also self-contained objects. One such object is the NuSMV script statement generator, which serves as an interface to the model checker via the model checker's script. Thus, another variation of an SMV model checker can be used by exchanging the script generator object, for an object that translates the CCM rules into the appropriate SMV script language. The tool does not only serve as an entry and translator of requirements, but it also executes a third party command line model checker exe, and reads its counter-examples via the executable's standard output stream. This combined with the changeable script generator facilitates the use of various open source model checkers. This makes the tool both an implementation to the property checking approach, and a development platform for future approach improvements.

Skid Loader Case Study

Using CCMChecker, we conducted a study using a real world embedded system, a commercial skid loader, to evaluate our approach. In particular, we focused on the method's ability to expose off-nominal behaviors of the system, which is the major concern for safety critical systems because off nominal behaviors that are not handled properly can result in safety-related accidents. Figure 21: A typical skid loader with the load bucket up (dashed lines) and down (solid line). The safety function in question is the automatic disabling of the skid loader bucketBoom when the safety seatBar is lifted. A skid loader, shown in Figure 21, is typically usable by anyone without special training or certification. Thus, it can potentially be missused. The manufacturer's objective is to make it as fool-proof as possible. In a typical skid loader design, the bucketBoom moves up and down directly in front of the cab, creating a potential problem for someone not confined within the cab during the bucket's use.

Figure 21 shows the relative proximity of the bucket's up/down movement and the operator cab. For safety reasons, the driver should be confined within the skid loader cab during the bucket's movement.

So, when the driver climbs into the cab, there is a seat-Bar that comes down over the driver, similar to the passenger restrain used in roller-coaster. For our case, we do not want the bucketBoom to move up or down if the seatBar is up. A seatBar that has been raised could lead to the driver protruding part of their body outside the cab, within the bucket's envelope of movement. This would create a potential safety hazard. We analyzed a total of 76 requirements focusing on those requirements that pertain to the controlling of the bucket. Of the 76 requirements we found an issue that revolved around four requirements; we focus our case study on those four requirements. The portion of the requirements 1 that pertains to that feature is:

REQ1: The bucket boom shall move upward when the left pedal is tilt forward with the driver's toe.

REQ2: The bucket boom shall move downward when the left pedal is tilt back with the driver's heel.

REQ3: The bucket boom shall stop moving when the left pedal is allowed in the default position by the driver's foot.

REQ4: All bucket boom operations are allowed only if the seat bar is lowered.

For confidentiality reasons, we have paraphrased the wording of the original requirements that were written for the embedded system company while maintaining the intended meaning. Note that in the initial set of requirements, REQ4 does state that "All bucket boom operations are allowed only if the seat bar is lowered." This would appear to address the desired safety requirement. However, to the stakeholders stating REQ4, there may be certain assumptions based on what they perceive as the driver's normal behavior.

For example, a typical normal behavior would be that the driver climbs into the cab, lowers the seatBar, and then operates the bucket. Upon existing, the driver would then stop operating the bucket (placing it in a down position), and then raise the seatBar when he is departing the cab. The following subsections describe how our approach is applied to the skid loader's requirements and detects possible safety-critical defects. The safety function in question is the automatic disabling of the skid loader bucketBoom when the safety seatBar is lifted.

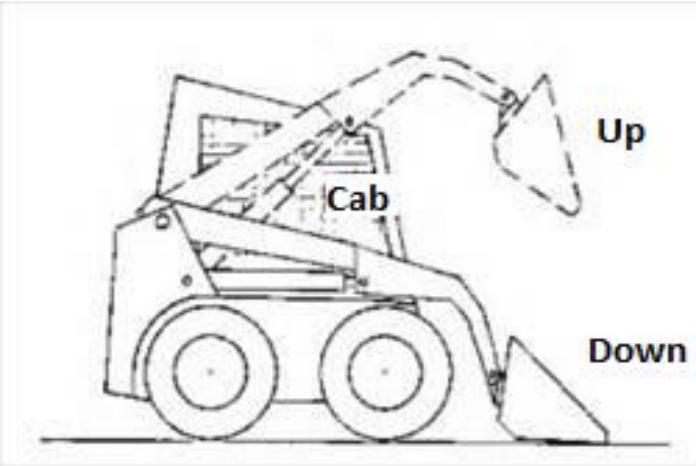


Figure 22. A typical skid loader with the load bucket up

Creating the CCM

To create the CCM, and subsequent NuSMV script, a user must enact the four steps used to construct a CCM. We explain how these steps are applied to the skid loader's requirements.

Step 1: Elicit a set of system Components (C), as expressed in the requirements. From the set of requirements above, we can readily identify three components: bucketBoom, seatBar, and leftPedal, which we have abbreviated. Note that, in this case, the three components are quickly identified because they are system artifacts that are either controlled by the system, or provide an interface to the driver. Three components will produce a CCM with three STDs. One advantage to our method is the ability to model the interactions between the components we wish to focus on, without having to model the entire system; this results in a compositional approach [15]. In this case, we are focusing on the interaction between the bucketBoom and the seatBar, so we can either model the leftPedal as a third STD or treat it as part of a transition. Having the option to abstract additional STDs as part of a transition helps manage the complexity and size of the resulting CCM. For our purposes, we will only model the bucketBoom and the seatBar.

Step 2: Elicit a set of Component(States)) as expressed in the requirements. Next, CCM creation requires that we assess what possible states the two components will be in at various times during the desired system operation. This gives us the following list of component states: seatBar(up), seatBar(down), bucketBoom(notMoving), bucketBoom (movingUpward), and bucketBoom(movingDownward).

Step 3: Elicit a set of inter-component Transitions (T), forming a State Transition Diagram (STD) for each component as expressed in the requirements. Having determined the required Component(State)s, we will now establish the inter-component transitions to build a STD for each component as expressed in the requirements. We can start with the seatBar, which only has two possible states (up or down), and consequently two possible transitions (T1 and T2). Using the CCM translation rule notation we construct the following two expressions from REQ4:

T1: seatBar(down) → seatBar(up)

T2: seatBar(up) → seatBar(down)

We can do the same with the bucketBoom, which three states and

six possible transitions (T3 - T8) between them:

T3: bucketBoom(notMoving) → bucketBoom(movingUpward)

T4: bucketBoom(movingUpward) → bucketBoom(notMoving)

T5: bucketBoom(notMoving) → bucketBoom(movingDownward)

T6: bucketBoom(movingDownward) → bucketBoom(notMoving)

T7: bucketBoom(movingDownward) → bucketBoom(movingUpward)

T8: bucketBoom(movingUpward) → bucketBoom(movingDownward)

Step 4: Elicit a set of Causal Relationships. The final step is for the stakeholders to establish the casual relationships between the different component STDs. The seatBar transition function is defined solely by the driver's act of raising or lowering the seatBar. Thus, we have:

[1] DRIVER(raisesSeatBar) : seatBar(down) →seatBar(up)

[2] DRIVER(lowersSeatBar) : seatBar(up) → seatBar(down)

According to the requirements, the bucketBoom's behavior is controlled by two components, the liftPedal and the seatBar. We decided earlier to only represent the seatBar's STD in the CCM, and specify the liftPedal's causal relation to bucketBoom within the bucketBoom's transition functions. Thus, we obtain the following expressions for the bucketBoom:

[3] DRIVER(tiltLPForward) ^ seatBar(down) :

bucketBoom(notMoving) → bucketBoom(movingUpward)

[4] DRIVER(LPDefault) : bucketBoom(movingUpward)

→ bucketBoom(notMoving)

[5] DRIVER(tiltLPBack) ^ seatBar(down) :

bucketBoom(notMoving) → bucketBoom(movingDownward)

[6] DRIVER(LPDefault) : bucketBoom(movingDownward)

→ bucketBoom(notMoving)

[7] DRIVER(tiltLPForward) ^ seatBar(down) :

bucketBoom(movingDownward) → bucketBoom(movingUpward)

[8] DRIVER(tiltLPBack) ^ seatBar(down) :

bucketBoom(movingUpward) → bucketBoom(movingDownward)

Figure 23 shows the CCM model derived from the requirements by following the four steps. Note that the transitions for the bucketBoom incorporate the behavior of the DRIVER as well as that of the seatBar.

The DRIVER is written in uppercase because it is considered a transition from the system's operating environment; a cause that originates external to the system. As to the seatBar's transitions, they are solely external, since their direct cause is the DRIVER; the CCM we have specified is heavily affected by external actions to the system. The DRIVER is a direct cause of behavior in both the bucketBoom and the seatBar.

This means that there are potentially many ways in which the DRIVER can cause an unexpected (off-nominal) behavior in the system. With the nominal behavior in mind, the most intuitive way to model REQ4 (Figure 22) would be by forming a logical conjunction of the seatBar component state with the bucketBoom, which enables the bucketBoom to move only when the seatBar is down. However, the real question is whether there is off-nominal behavior by the driver that would defeat the desired safety feature. In other words, can the DRIVER somehow create a scenario in which the bucketBoom is moving while the seatBar is up?

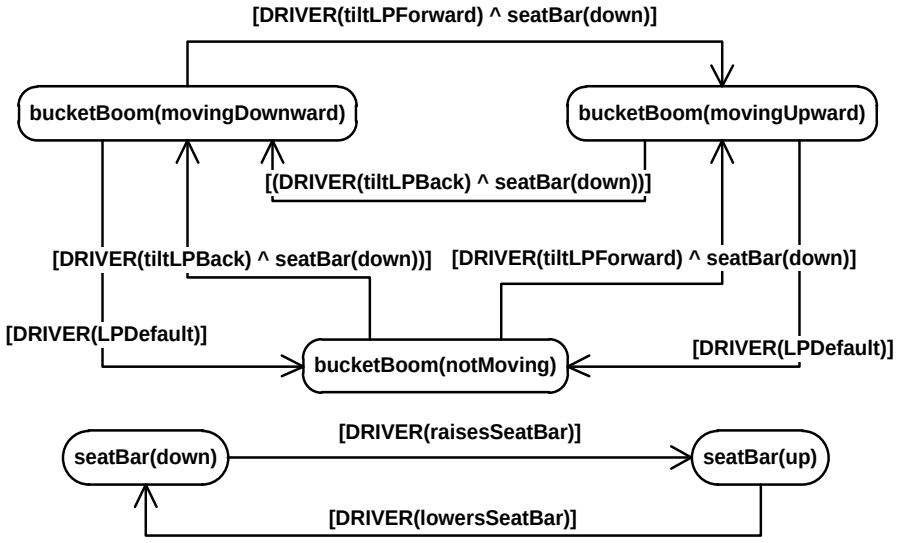


Figure 23. The partial CCM of the Skid Loader

Creating the NuSMV Script

The CCM is used as a stakeholder friendly front-end to a NuSMV model checker. This requires the CCM's translation into the script notation used in NuSMV. To achieve this we use a mapping algorithm that leverages the fact that both CCM and NuSMV notations have an IF-THEN structure. Another similarity is that both CCM and NuSMV notations represents components, present states, next states and transitions. Thus, both share similar artifacts and semantics, while expressing those semantics using different syntax. Figure 24 shows a typical NuSMV script. Note that Figure 24 is a partial listing of the complete script.

```
MODULE main
VAR
   bucketBoom : {movDown, movUp, notMov};
   seatBar: {down, up};
   driver : {tiltLPBack, tiltLPForw, lpDefault, raiseSeatBar, lowSeatBar};

ASSIGN
     init(seatBar) := up;
     init(bucketBoom) := notMov;

ASSIGN
     next(seatBar) := case
          seatBar = up & driver = lowSeatBar : down;
          seatBar = down & driver = raiseSeatBar : up;
          TRUE:seatBar;
          esac;
ASSIGN
     next(bucketBoom) := case
          --One
          bucketBoom = movDown & (seatBar = down & driver = tiltLPForw) : movUp;
          bucketBoom = movUp & (seatBar = down & driver = tiltLPBack) : movDown;
          --Two
          bucketBoom = notMov & (driver = tiltLPBack & seatBar = down) : movDown;
          bucketBoom = movDown & driver = lpDefault : notMov;
          --Three
          bucketBoom = notMov & (driver = tiltLPForw & seatBar = down) : movUp;
          bucketBoom = movUp & driver = lpDefault : notMov;
          TRUE:bucketBoom;
          esac;
```

Figure 24. An example of partial NuSMV script

Note that the ASSIGN sections of Figure 24 use an IF-THEN structure to describe a component's behavior. For example, The statement, seatBar = up & driver = lowSeatBar : down, reads IF seatBar is up and the driver lowers the seatBar, THEN the seatBar will transit to a down state. Expressed in CCM notation, the same behavior is written as: driver(lowSeatBar) : seatBar(up) ! seatBar(down). In the general form, *T : C(S1) → C(S2)*, Transition (T) can be a disjunctive and/or conjunctive expression of component states. In general terms, given a set of CCM expressions, the form is shown in line (1):

(1) Cn(S2) : Cp(S1) → Cp(S2),

Where Cn and Cp are different components, and Cp transitions from states S1 to S2 as caused by Cn being in state S2. The algorithm used to create the ASSIGN sections iterates through each CCM expression, performing the following steps:

Convert Cp(S1) to Cp = (S1)

Convert Cp(S2) to : S2;

Convert Cn(S2) to Cn = S2

Convert Cp(S1) to nextCp

Convert Cp(S1) to 1 : Cp

Concatenate ASSIGN, nextCp, := case, Cp = (S1), &, Cn =

S2, : S2;, 1:Cp, and esac;

The concatenated terms form the ASSIGN section (2):

(2) ASSIGN

next(Cp) := case

Cp = S1&Cn = S2 : S2;

1:Cp;

esac;

The VAR section of the NuSMV script is created using a similar parse-convert-concatenate algorithm resulting in the two CCM expressions (3) and (4) translated into the VAR section starting at (5).

(3) Cn(S2) : Cm(S1) → Cm(S2)

(4) Cp(S1) : Cn(S1) → Cn(S2)

(5) VAR

Cm : fS1; S2g;

Cn : fS1; S2g;

Creating the NuSMV Temporal Properties

It is not sufficient to convert the CCM into the model portion of the NuSMV script. To complete the NuSMV script, we still need to specify the temporal logic properties that will be used to expose any off-nominal scenarios that may be produced in the CCM. While the creation of temporal logic properties is traditionally the most knowledge intensive endeavor in model checking, we will facilitate the automatic creation of the properties by leveraging two areas of research that has been conducted in the pass. The first is the development of temporal logic patterns as a means to simplify the specification process [130, 136]. The second area of research, which directly relates to the first area, is the fact that the most useful temporal properties can be achieved with a handful of patterns [130, 131]. For our purposes, we have leveraged this prior research to achieve the desired defect coverage with the following four temporal logic patterns (written in the NuSMV script language).

A: AG(!(Cn = S1) → EF(Cn = S1))

B: AG((Cn = S1) → EF(!(Cn = S1)))

C: AG(!(Cn = c(s)1 & Cm = c(s)2) → EF(Cn = c(s)1 & Cm = c(s)2))

D: !EF(Cn = c(s)1 & Cm = c(s)1)

In each pattern, Cn represents component, and Sn is that component's state. For example, C1 = S1 means State number 1 of component number 1. Patterns A and B are used to expose incompleteness in the CCM, which would translate into an undesirable behavior in the system. Pattern A verifies whether a given component(state) can be entered. This is a measure of the state's in-degree, of which there should be at least one in-degree per component(state). Pattern B verifies whether a given component(state) can be exited. This is a measure of the state's out-degree, of which there should be at least one per component(state). Together, patterns A and B verify whether a round-trip traversal is possible in a given component's STD. Embedded systems typically do not terminate, while running, and are in a given state at any given time. Thus, every component(state) in the CCM should have at least one in-degree and one out-degree. Patterns A and B are generated automatically for each component(state) by the tool. Pattern C also verifies round-trip traversal but on a system level. In this case, the question is whether the initial system state can be reentered. The component(states) that comprise the initial system state is selected by the user, while the temporal property is created by the tool. PatternD, is the strongest property used to exposed off nominal behavior. It determines whether an undesired system state is eventually reachable or not.

The component(states) that comprise the undesired system state is selected by the user, while the temporal property is created by the tool. Referring back to listing one, we see one example of a temporal logic property, in the SPEC (last line) section of the script: !EF(seatBar = up& bucketBoom = movUp). Note that this property uses pattern D, which determines whether an unsafe system state is eventually reachable or not. To create the temporal properties the support tool uses a mapping algorithm similar to the one used for the script model. We now turn to a case study, derived from the design of a real world skid loader presently available on the market. The Skid Loader's manufacturer will be kept confidential. In particular, we focused on the method's ability to expose off-nominal behaviors of the system, which is the major concern for safety critical systems because off nominal behaviors that are not handled properly can result in safety related accidents. The CCM of Figure 23, is automatically

converted into the model portion of the NuSMV script as shown in Figure 24. The temporal logic properties are also generated, and placed in the NuSMV script. The followings are the properties generated for the skid loader example: !EF(seatBar = up&bucketBoom = movUp) : This verifies that an undesired system state cannot occur. It is using pattern D as listed in Section 6.4

AG(!(seatBar = up&bucketBoom = notMov)

! EF(seatBar = up&bucketBoom = notMov))

This verifies that you can always get back to initial states. It is using pattern C as listed in Figure 19. The following properties verify that there is at least one in-degree for each of the seatBar states. Similar in-degree properties using the same pattern below were created for the bucketBoom (not shown). They are using pattern A as listed in Figure 19.

AG(!(seatBar = up) ! EF(seatBar = up))

AG(!(seatBar = down) ! EF(seatBar = down))

The following properties verify that there is at least one out-degree for each of the seatBar states. Similar out-degree properties using the same pattern below were created for the bucketBoom (not shown). They are using pattern B as listed in Figure 19.

AG((seatBar = up) ! EF(!(seatBar = up)))

AG((seatBar = down) ! EF(!(seatBar = down)))

SPEC --Verifying that an undesired system state cannot occur

!EF(seatBar = up & bucketBoom = movUp)

SPEC --Verifying that you can always get back to initial states (Completeness)

AG(!(seatBar = up & bucketBoom = notMov) -> EF(seatBar = up & bucketBoom = notMov))

SPEC --Verifying that you can always exit the initial states (Completeness)

AG((seatBar = up & bucketBoom = notMov) -> EF(!(seatBar = up & bucketBoom = notMov)))

SPEC --Verify that there is at least one indegree (Completeness)

AG(!(seatBar = up) -> EF(seatBar = up))

SPEC --Verify that there is at least one outdegree (Completeness)

AG((seatBar = up) -> EF(!(seatBar = up)))

SPEC --Verify that in B(1):A(1)->A(2), B(1) precedes A(2)

!E[!(seatBar = down) U (bucketBoom = movUp & !(seatBar = down))]

Model Checking Results

After the model and temporal properties are combined into a NuSMV script, the script is executed by a NuSMV command line executable. The standard out is read from the executable, and any and all counter-examples that represent a property that did not hold is parsed out. The counter example can then be represented as either a traversal path consisting of only those states and transitions that lead up to the undesired state, or as a sequence of CCM simulation snapshots, where each snapshot highlights a system state. Both representations will be demonstrated later in this section. Figure 24 shows the results of running the skid loader NuSMV script through the model checker; only the single resulting counter example is shown, using a traversal path representation. The requirements as stated will result in an undesired system state, that of the seatBar being up, while the bucketBoom is moving. Furthermore, the model checker was able to reveal this because the NuSMV script did not purposely assign any rules or initial values to the DRIVER component states. This allowed for an unconstrained enactment of the DRIVER's behavior upon the model. Subsequently, all possible behavioral combinations by the DRIVER were simulated. This includes any off-nominal behaviors that stakeholders may have assume the driver would never enact under normal operation of the skid loader.

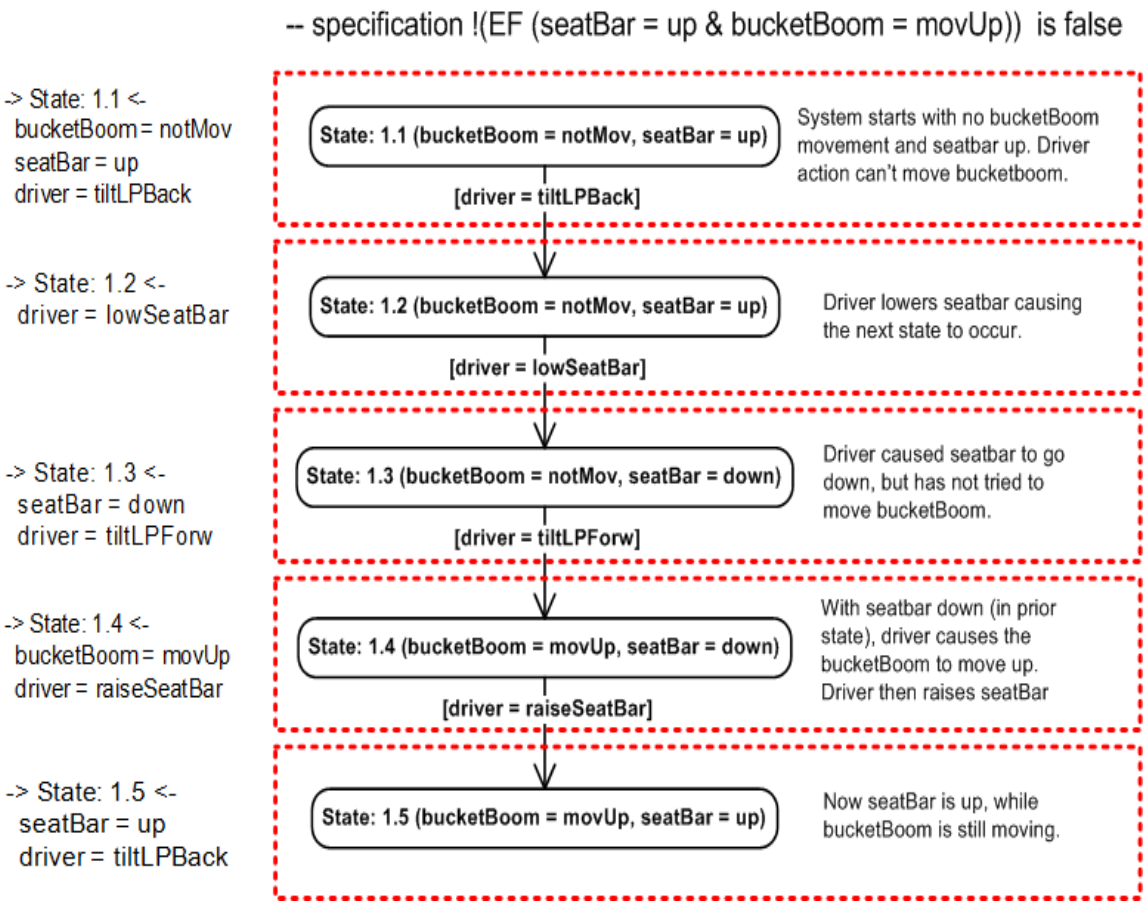


Figure 25. Counterexample from first run through model checker

In this case, from the counter-example, we see that the undesired system state is reached when the driver decides to lift the seatBar while the bucketBoom is in motion; a trivial action on the driver's but one that was unaccounted for in the requirements. REQ4 says: "All bucket boom operations are allowed only if the seatBar is lowered." This requirement is satisfied in the case where the bucket-Boom is not allowed to move until the seatBar is lowered, but no requirement covers the scenario where the seatBar is raised, once the bucketBoom is moving. This is an example of a "trivial" yet potentially dangerous oversight on the part of stakeholders. In response to the exposed missing requirement, we could make the adjustment shown in Figure 25.

Since the undesired state occurs as a result of the driver's unconstrained behavior with the seatBar, we have added a seatBar lock in order to constraint the seatBar and limit the results of driver's behavior. The addition of the seatBar lock represents one or more requirements that were missing and could have sooner or later been elicited or conceived in order to address the unconstrained behavior of the driver. If not caught during the requirements phase, this issue would have likely been caught down-stream during the system's development at a higher cost.

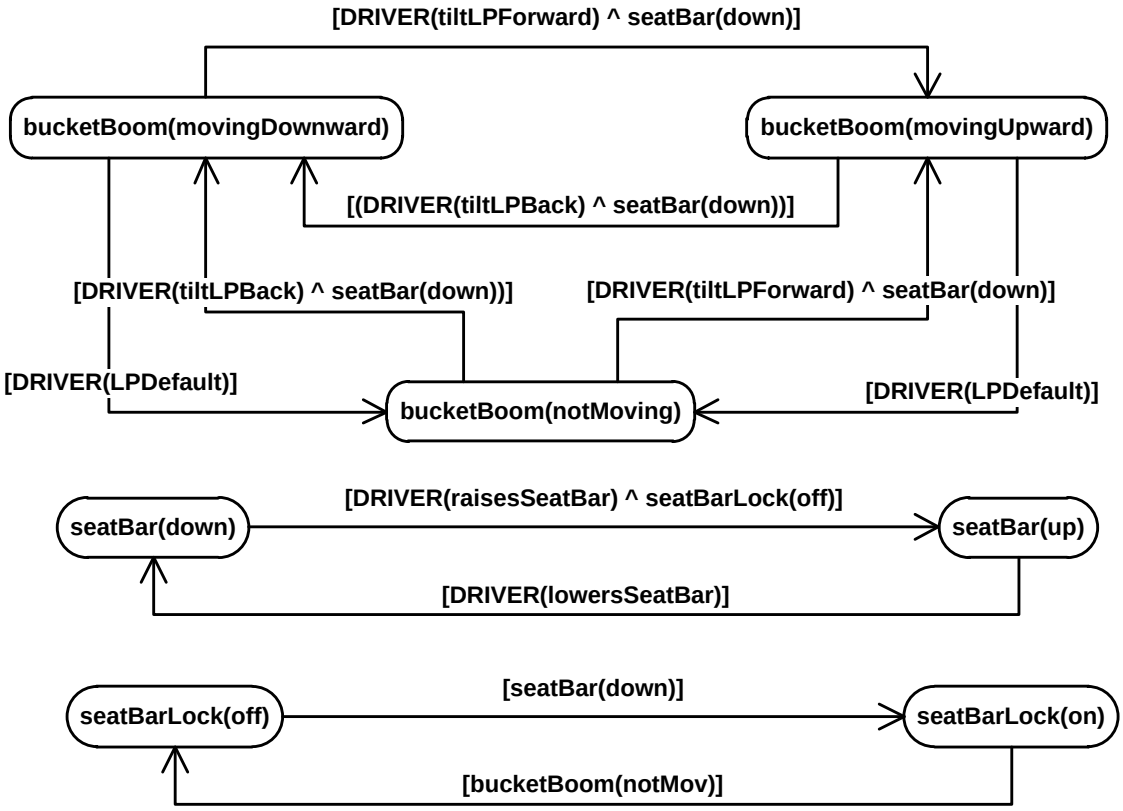


Figure 26. The modified CCM in response to the counterexample of Figure 24

However, using our approach, it would have likely been caught very quickly, as the requirements are elicited, since CCMChecker allows for continuous simulation of the requirements, on an incremental basis.

The additional requirements describing the seatBar lock could be worded and added to the set of requirements at the stakeholder's discretion. Also, the wording can be taken directly from the CCM by explicitly describing the STD pertaining to the added seatBar lock. For example, additional requirements could be expressed as "REQ5: If the seatBarLock is off, and the seatBar is pulled down, the seat-BarLock shall activate," and "REQ6: If the seat bar is down, it can only be raised up, if the driver tries to raises the seatbar and the seatBarLock is off." Notice that the wording is taken from the modified CCM in Figure 25.

This new CCM was automatically described in a new NuSMV script, run again in the NuSMV executable, and produced the result as shown in Figure 26. The new counter example in Figure 10 shows that the addition of a seatBarLock requirement was not enough to avoid the undesired system state. To obtain a better understanding of how to address the remaining problem, we analyzed the new counter-example, using two sequence snapshots from the CCM simulations enacted by approach. Figure 27 shows the succession of two CCM snapshots leading up to the undesired state of the seatBar being up while the bucketBoom is moving. The two snapshots in Figure 27 are created by the approach, as it runs a simulation of the counter-example using the same CCM diagram generated during elicitation. Generating a CCM-based simulation helps the stakeholders interpret the counter-example produced by the Model Checker.

The two CCM snapshots in Figure 27 are labeled "State 1.5" (top) and "State 1.6" (bottom), which corresponds to the same two states of the counter example in Figure 26. Starting from the top of Figure 27, the states with the bold line represent the states the three components are in during State 1.5. The transitions in bold will allow a state transition on the next clock cycle, resulting in State 1.6.

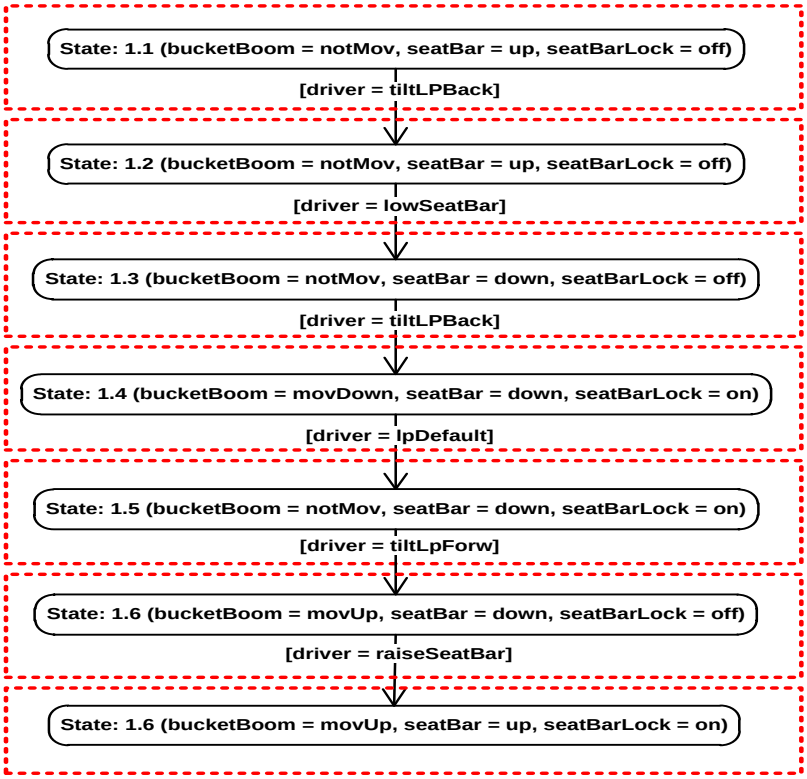


Figure 27. Counterexample from running the first corrected set of requirements

In State 1.5 of Figure 27, the system is poised for the bucketBoom to start moving upward, and the seatBarLock to unlock during the next transition cycle. With the bucketBoom moving and the seatBar unlock (State 1.6), there is nothing to prevent the unsafe condition if the driver decides to lift up the seatBar. Therefore, the only apparent solution is to further constraint how State 1.6 occurs, by enabling the seatBar to unlock solely on the basis of a non-moving bucketBoom. By examining the transitions in bold in State 1.5 of Figure 27, the stakeholders might decide that it is a bad idea to allow the unlocking of the seatBarLock only when the bucketBoom is not moving. Therefore, they may try placing a further restriction on how the seatBar is unlocked. One additional restriction would be to logically AND the driver's attempt to raise the seatBar (driver (raiseSeatBar)), with bucketBoom(notMov), which would be the only safe bucketBoom state for the driver to raise the seatBar. We see this modification in Figure 28. While it may not be obvious that this additional

restriction would finally solve the problem, the fact is that CCMChecker's model checker will take very little time to verify whether it is indeed a solution.

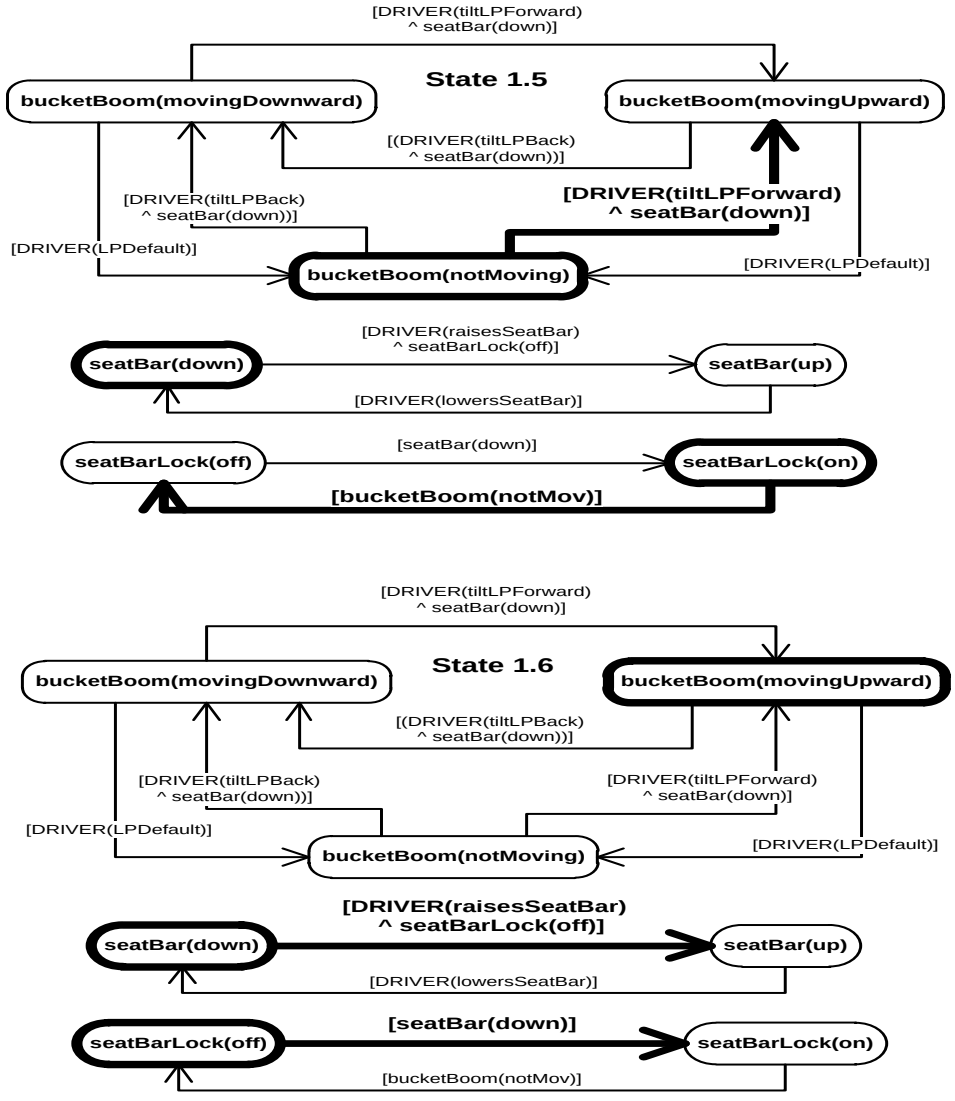


Figure 28. Two snapshots in CCM-based counterexample simulation

As it turns out is, when running the model checker on the CCM in Figure 28, all properties hold (they all return true). The additional seatBarLock and associated restrictions, can be expressed as additional requirements at the stakeholders discretion.

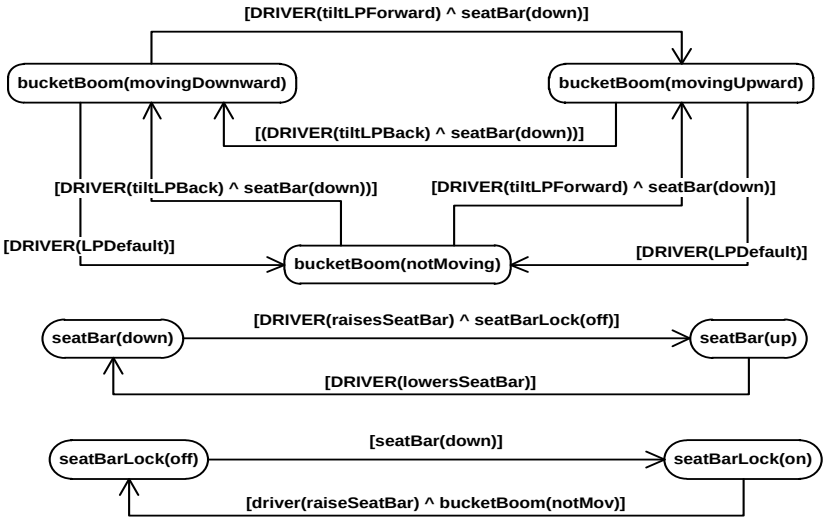


Figure 29. Additional restriction is placed on the unlocking of the seatBarLock

Discussion and Limitations of PCA

The skid loader case study stresses some important observations about the proposed method. To add an exposed missing requirement, the stakeholders can suggest a solution, based on how they have assessed the problem, and the method/tool provides feedback on the solution's success. This greatly accelerates the process of finding missing requirements that address off-nominal scenarios. Of course, for the sake of space, we have only specified one undesired system state. In an actual elicitation process, more than one undesired state could be entered, increasing the chances of catching any further problems that may have been produced by using the final solution in this study.

We are encouraged by the results and plan on applying the technique to a greater set of requirements, particularly where the system is intended to interact with a high percentage of external stimuli, up and beyond that of merely user interaction. This would include sensor data, and interrupts. While our focus has been on exposing off-nominal behaviors, we think that other anomalies can be potentially exposed as well, such as inconsistency between concurrent processes, and possible deadlock and starvation scenarios. Some of the remaining questions

pertaining to our approach, include the question of scalability. CCM's ability to focus on the interaction between a subset of a system's components allows for the analysis of subsystems. This in turn, enables us to divide and conquer the system in such a way that modeling a complete system may not be necessary in most cases. Partial modeling would keep the number of states manageable.

CCM STATE PROFILING APPROACH (SPA)

In embedded systems where there are considerable interactions with the system's operating environment, it is not unusual for the environment to behave in a manner unaccounted for by the system specification. Of particular concern are those interactions that come from the system's human operators. Human behavior can be unpredictable and off-nominal, contrary to what the designers of a system consider normal behavior for the operator. Stakeholders typically make assumptions when specifying a set of system requirements. Key to these assumptions is the way in which the system and its operating environment are intended to behave; these intended behaviors are treated as nominal. Quite often, unintended, unusual behaviors are not accounted for; therefore, not every contingency the system might have to address is included in the requirements [138]. We refer to these unintended behaviors as Off-Nominal Behaviors (ONBs) [10, 151]. ONBs can produced in an undesired system state, which can result in a safety hazard or damage to the equipment being controlled. At the heart of our research and subsequent approach is the question: What undesired states can the system's operating environment cause, due to lack of explicitness in the requirements? Or what have the stakeholders miss from not stating things clearly enough? We want to address these questions in a way that allows non-technical stakeholders, particularly domain expects, to participate in addressing the ONB problem.

Even a simple set of requirements can result in a system that contains more susceptibility to ONBs than anticipated, primarily due to requirement ambiguities. To illustrate this claim, consider the following simple requirement: A system shall consist of a push-button switch and a motor. When an operator presses the switch, the motor shall be turned on, and it shall stay on for 3 seconds before automatically turning off. These requirements assume that the human operator will press the switch, wait for the motor to start, and then immediately release the switch. The system will then start counting the seconds and will turn the motor off after 3 seconds. The system's correct operation assumes that the operator behaves in the way implied by the requirements; this action is considered the operator's nominal behavior.

However, what if the operator releases the switch before the motor starts or decides to turn the motor off before the 3 seconds expire? How will the system react to an operator who acts in an off normal manner? Will the system be placed in an undesired state from which it cannot recover? Is the system specified with enough contingencies to avoid an undesired state from an operator or from the environment in general? Our experience with embedded requirements suggests an increase in potential ONB-induced undesired states with an increase in requirement ambiguities.

To date, some researchers have tried to address the ONB problem [147, 148, 149, 133], but most attempts to resolve this problem have focused on how the human operator reacts to off-nominal situations within the operating environment [151]. Our approach addresses the ONB problem from the system's perspective by exposing any undesired states unintentionally produced by the requirements. We use a stakeholder-friendly modeling approach that we developed called the Causal Component Model (CCM) and algorithmically model all possible transition paths in the system model. These paths represent both intended and unintended system behaviors in response to the operating environment. We then determine whether non-recoverable undesired states can be reached, and if so, our approach determines which ONB results in that state.

Overview

Our approach operates on the premise that ambiguities in a set of requirements can allow ONBs to result in a non-recoverable, undesired system state. "Undesirable" means a system state that should be avoided while "non-recoverable" means that the system cannot exit that state automatically; both "undesired" and "non-recoverable" are necessary terms in our definition. For example, in a microwave oven, an undesired state would be the oven cooking while the door is open. We will designate that situation as {Oven(cooking), Door(open)}, where both Oven and Door are components of the microwave oven system. However, we simply cannot avoid all occurrences of {Oven(cooking), Door(open)}.

We want the microwave oven to detect when {Oven(cooking), Door(open)} occurs because there is typically nothing to keep a person from opening the door while something is cooking. Therefore, an undesired state is tolerable if the system is specified to recover from it. (In this case, when {Oven(cooking), Door(open)} occurs, the oven should automatically be turned off.) However, if {Oven(cooking), Door(open)} occurs and no means to recover from it has been specified, then {Oven(cooking), Door(open)} becomes a non-recoverable, undesired state that must be avoided. Our approach would determine if an undesired state can occur and if it is non-recoverable, and then indicate what environmental action caused its occurrence. We now describe, in detail, the major steps involved with the CCM state profiling approach as expressed in Figure 30. The squares in Figure 30 refer to the artifacts produced while the numbered ovals describe the processes producing the artifacts.

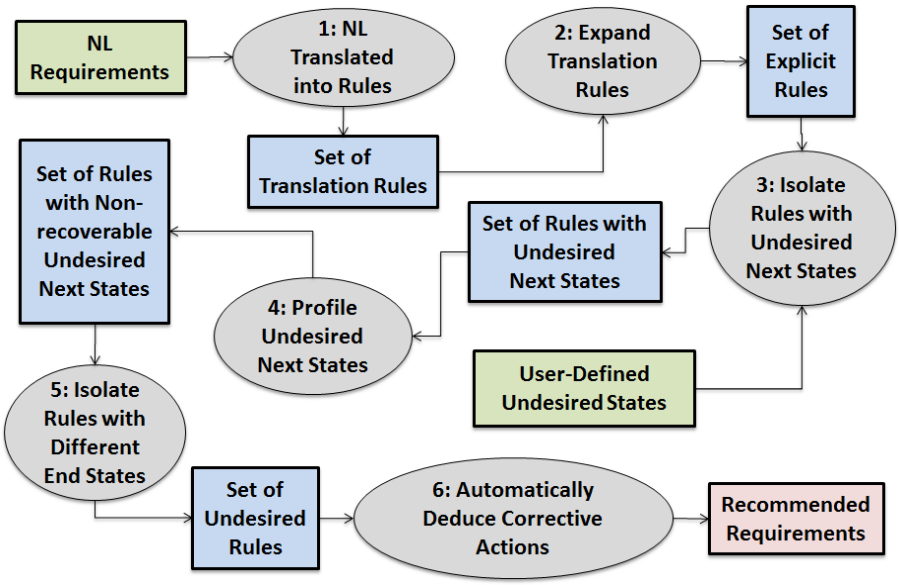


Figure 30. Overview of CCM state profiling approach

The green squares show input artifacts; the red square is the final output for the process. The following list is written to correspond with each oval.

1) Natural Language (NL) Translated into Rules: NL requirements for an embedded system are translated into a set of translation rules that can be modeled as a CCM. The translation is achieved by using the four-step CCM creation process. A translation rule specifies the transition between two given system states and is expressed in the form of the mapping function: Transition_Cause : Present_State → Next_State. The Transition_Cause can be directly induced by the system's operating environment or by the system itself. At this stage, not every possible system state is accounted for by the translation rules because not all states are explicitly expressed in the requirements, thus we perform the next step.

2) Expand Translation Rules: To account for every possible system state, we must expand the translation rules to a set of explicit rules that define the transitions between as many system states within the system's entire state space as possible. Expanding the rules accounts for all transitions that are both explicitly stated and implied by the requirements.

3) Isolate Rules with Undesired Next States: Each rule specifies a transition from a given present state to the next state. Using a set of stakeholder-defined, undesired states, we determine which rules transition into the next state that is also a user-defined, undesired state and then create a set of these rules.

4) Profile Undesired Next States: Typically, but not always, a rule's next state is matched to the present state of another rule, making that next state recoverable by another rule. "State profiling" determines if a state is recoverable by a rule specifying a Transition_Cause by the system. If so, we call that next state "system recoverable." In this step, we are looking for undesired next states that are non-recoverable by the system. Such an undesired state can be a safety hazard, or it can cause the system to lock up. A rule with an undesired, non-recoverable next state is called an "undesired rule."

5) Isolate Undesired Rules with Different End States: We define "end states" as a rule's present and next states. If a given rule has end states that are both undesired, then we ignore that rule because the transition into an undesired state does not originate with that rule.

If a rule has an acceptable present state and an undesired next state, then that rule is considered the cause of an undesired state; the rule is isolated for further analysis. We focus on isolating rules that specify an environmental cause because we are interested in how an ONB from the environment can cause an undesired state.

6) Stakeholders Address ONB Problem: From the final set of undesired rules, we create a list describing which environmental causes result in an undesired state from a given acceptable state. Knowing how the environment causes a transition from an acceptable state to an undesired state helps the stakeholder determine how to address the potential ONB.

SPA Preliminaries

In our approach, we utilize the concept of a component. A component is part of a system's composition that can change states and/or cause a change in state. For example, in a microwave oven system, the components can consist of {startButton, ovenDoor, cookTimer}. Note that a potential subsystem, such as the cookTimer, can also be considered a component. It is up to the stakeholders to determine how they want to decompose a system into components, meaning that different abstraction levels can be modeled. We assign states to components and refer to them as component--states. We combine component--states to form (global) system states.

These system states are mapped to one another using rules, derived from the requirements, which define the cause of the transition between two given system states. Finally, there is the goal of exposing non-recoverable, undesired system states and the off-nominal behaviors that may cause them. We now formally define these concepts and operations. An example of these concepts and operations follows.

The Expanded Rules Algorithm

The expanded rules are the explicit rules derived from the ECR and SCR translation rules. We expand the translation rules to eliminate the zeroes, resulting in each possible interpretation for the translation rules.

The algorithm begins by generating the system's entire state space. Let CSn be a set of component--states for a given component and CARD = { |CS1|, |CS2|, . . . , |CSn| } be the set of each component--state's cardinality. The system state space, SSP, is the Cartesian product SSP = CARD x CARD. We can parse the digits of a numerical system state into a set of component--states as shown in this example: 2.0.1 ≡ $\{cs_1, cs_2, cs_3\}$, where $cs_1 = 2$, $cs_2 = 0$, $cs_3 = 1$. Similarly, we numerically represent and parse the state-space members into the form $\{ss_1, ss_2, \ldots, ss_n\}$. We expand the translation rules by applying the following algorithm to the translation rules.

1. FOR EACH ordered pair ‹csn, ssn› derived from $\{cs_1, cs_2, \ldots, cs_n\}$ x $\{ss_1, ss_2, \ldots, ss_n\}$
2. IF $cs_n = ss_n$, THEN $cs_n \rightarrow cs_n$, and we compare the next digit at n + 1.
3. IF $cs_n \neq ss_n$, THEN we stop comparing digits and go to the next member of SSP.
4. IF ($cs_n = 0$ $ss_n \neq 0$), THEN $ss_n \rightarrow cs_n$, and we compare the next digit at n + 1.

The algorithm stops when all the ordered pairs, ‹cs_n, ss_n›, have been processed. The expanded rules are contained in two separate sets based on whether the rule causes are from the environmental or the system.

The State Profiling Algorithm

State profiling is used to determine the number of in-degrees (ID) and out-degrees (OD) for a given system state. Figure 30 shows a given state with its in/out-degrees.

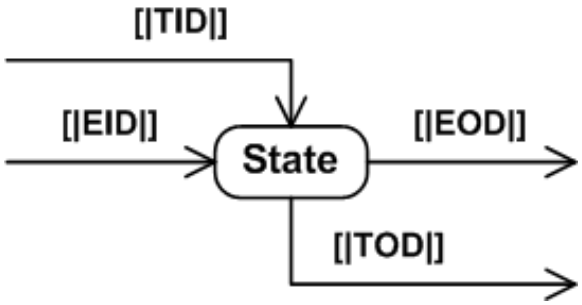


Figure 31. System state with labeled in/out degrees

The degrees are further designated by the type of transition cause, for which there are two: a system cause (T) and an environmental (E) cause. "TID" is the set of all in-degrees caused by the system, "EID" is the set of all in-degrees caused by the environment, "TOD" is the set of all out-degrees caused by the system, and "EOD" is the set of all out-degrees caused by the environment. We use the set cardinality symbol, "| |", to indicate the number of in/out-degrees for a given set. For example, |TOD| = 0 means that there are no system out-degrees. State profiling establishes a cardinal number for each set which, in turn, indicates how that given state is transitioned by either the system or the environment.

A key question is whether a state is system recoverable. We want to isolate states that are non-recoverable by the system, which translates into states with |TOD| = 0. The first step in state profiling is to determine the cardinal number for each of the four degree sets: EID, TID, EOD, and TOD. The state--profiling algorithm compares every system state with the present and next state of each rule to determine the number of in-degrees and out-degrees according to the type of rule (environmental cause or system cause). State-profiling is achieved with the following algorithm.

1. LET |EID| = 0, |EOD| = 0
2. LET |TID| = 0, |TOD| =0
3. LET EC be the set of all rules: $ec : se_p \rightarrow se_n$, that have an environmental cause.
4. LET TC be the set of all rules: $tc : st_p \rightarrow st_n$ that have a transition cause.
5. LET S be the set of all system states, *s*
6. FOR EACH system state, $s \in S$
7. COMPARE s to each rule state {sep, sen, stp, stn}
8. IF s ≡ sep, THEN increment |EOD| by 1
9. IF s ≡ sen, THEN increment |EID| by 1
10. IF s ≡ stp, THEN increment |TOD| by 1
11. IF s ≡ stn, THEN |increment |TID| by 1

We use the following convention to represent a state's profile: State {|EOD|, |EID|, |TOD|, |TID|}. For example, if state 1.2.1. has |EOD| = 1, |EID| = 2, |TOD| = 1, and |TID| = 3, we write 1.2.1.{1, 2, 1, 3}. Due to requirement ambiguities, even a simple set of requirements can result in a system that contains more susceptibility to ONBs than anticipated. To illustrate this claim, we will use the following simple requirements and define one undesired state:

REQ1: A system shall consist of a push-button switch, a motor, and a temperature sensor.

REQ2: The motor shall be turned on and off when an operator presses the switch on and off, respectively.

REQ3: The motor shall be allowed to turn on if the temperature is below 80 but not when overheated (i.e., above 100 degrees).

UNDESIRED STATE: It is any system state when the motor is on while overheated.

NOTE that this example is similar to the prior motor controller, except this time we have purposely seeded the requirements with a defect. We will apply the process outlined in Figure 1 to the requirements we just stated. For clarity, we will used the same process headinECR labeled in Figure 1.

1) Rules Creation: Our approach begins with translating the requirements into translation rules. Translation is performed using a four-step process:

Step 1: Determine components:

{Switch, Temp_Sensor, Motor}

Step 2: Determine component--states:

{Switch(off), Switch(on), Temp_Sensor(safe), Temp_Sensor(overheated), Motor(off), Motor(on)}

Step 3: Determine component--state transitions:

Switch(off) → Switch(on)

Switch(on) → Switch(off)

Temp_Sensor(safe) → Temp_Sensor(overheated)

Temp_Sensor(overheated) → Temp_Sensor(safe)

Motor(off) → Motor(on)

Motor(on) → Motor(off)

Step 4: Determine transition causes (resulting in the following translation rules, 1 to 6):

[1] USER(press) : Switch(off) → Switch(on)

[2] USER(releases) : Switch(on) → Switch(off)

[3] TEMP(>100) : Temp_Sensor(safe) → Temp_Sensor(overheated)

[4] TEMP(<80) : Temp_Sensor(overheated) → Temp_Sensor(safe)

[5] Switch(on) Temp_Sensor(safe) : Motor(off) → Motor(on)

[6] Switch(off) : Motor(on) → Motor(off)

The translation rules can also be used to create a Causal Component Model (CCM) [74, 76, 161]. Figure 32 shows a CCM that models the translation rules. The interrelationships among the three STDs occur on the basis of causation.

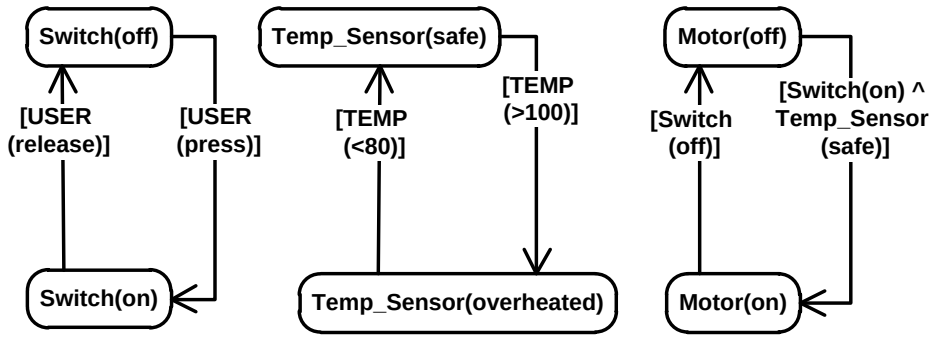


Figure 32. A Causal Component Model, of the requirements

For example, the switch has a causal relationship with the motor because the switch causes a transition between the two motor states. Transitions for the three STDs occur concurrently on a synchronous basis. Thus, when the USER presses the switch, its transition from “off” to “on” occurs on the following Transition (T) cycle. The switch's “on” state then allows the motor to transition from “off”

to “on” during the next T cycle. We next create a numerical representation of component(states) by using the following component order {Switch, Temp_Sensor, Motor}, where off = 1, on = 2, safe = 1, and overheated = 2.

Switch(off), 1.0.0.

Switch(on), 2.0.0.

Temp_Sensor(safe), 0.1.0.

Temp_Sensor(overheated), 0.2.0.

Motor(off), 0.0.1.

Motor(on), 0.0.2.

We use the numerical Component(States) to create a numerical representation of the translation rules, yielding:

[1] USER(press) : 1.0.0. → 2.0.0.

[2] USER(releases) : 2.0.0. → 1.0.0.

[3] TEMP(>100) : 0.1.0. → 0.2.0.

[4] TEMP(<80) : 0.2.0. → 0.1.0.

[5] 2.0.0. ^ 0.1.0. : 0.0.1 → 0.0.2.

[6] 1.0.0. : 0.0.2. → 0.0.1.

We apply absorption and propagation to numeric rules 5 and 6. Absorption results in [5] T.1 : 2.1.1. → 0.0.2. and [6] T.2 : 1.0.2. → 0.0.1. and propagation results in [5] T.1 : 2.1.1. → 2.1.2. and [6] T.2 : 1.0.2. → 1.0.1. producing the final version of the translation rules:

[1] USER(press) : 1.0.0. → 2.0.0.

[2] USER(releases) : 2.0.0. → 1.0.0.

[3] TEMP(>100) : 0.1.0. → 0.2.0.

[4] TEMP(<80) : 0.2.0. → 0.1.0.

[5] T.1 : 2.1.1. → 2.1.2.

[6] T.2 : 1.0.2. → 1.0.1.

2) Expand Translation Rules: The translation rules are expanded according to Expanded Rules Algorithm, resulting in the following explicit rules:

[1] USER(press): 1.1.1. → 2.1.1.

[2] TEMP(>100): 1.1.1. → 1.2.1.

[3] USER(releases): 2.1.1. → 1.1.1.

[4] TEMP(>100): 2.1.1. → 2.2.1.

[5] USER(press): 1.2.1. → 2.2.1.

[6] TEMP(<80): 1.2.1. → 1.1.1.

[7] USER(releases): 2.2.1. → 1.2.1.

[8] TEMP(<80): 2.2.1. → 2.1.1.

[9] USER(press): 1.1.2. → 2.1.2.

[10] TEMP(>100): 1.1.2. → 1.2.2.

[11] USER(releases): 2.1.2. → 1.1.2.

[12] TEMP(>100): 2.1.2. → 2.2.2.

[13] USER(press): 1.2.2. → 2.2.2.

[14] TEMP(<80): 1.2.2. → 1.1.2.

[15] USER(releases): 2.2.2. → 1.2.2.

[16] TEMP(<80): 2.2.2. → 2.1.2.

[17] T.1: 2.1.1. → 2.1.2.

[18] T.2: 1.1.2. → 1.1.1.

[19] T.2: 1.2.2. → 1.2.1.

3) Isolate Rules with Undesired Next States: Recall that, in our example, an undesired state is any system state where the motor is on (0.0.2.) while overheated (0.2.0.), resulting in a mask (0.2.2.).

When the zeroes are filled by the system's state space, we get the two undesired system states: {1.2.2., 2.2.2.}. The rules with the next states that are undesired are as follows:

[10] TEMP(>100): 1.1.2. → 1.2.2.

[12] TEMP(>100): 2.1.2. → 2.2.2.

[13] USER(press): 1.2.2. → 2.2.2.

[15] USER(releases): 2.2.2. → 1.2.2.

4) Profile Undesired Next States: The undesired next states are profiled, producing the following two profiles:

State {EOD, EID, TOD, TID}

1.2.2.{ 2, 2, 1, 0 }

2.2.2.{ 2, 2, 0, 0 }

Note that state 2.2.2. is non-recoverable because |TOD| = 0. Thus, the rules with non-recoverable, undesired next states are as follows:

[12] TEMP(>100): 2.1.2. → 2.2.2.

[13] USER(press): 1.2.2. → 2.2.2.

5) Isolate Undesired Rules with Different End States: Of the two rules, 12 and 13, with non-recoverable, undesired next states, it is rule 12 that transitions from an acceptable state (2.1.2.) to an undesired state (2.2.2.), and the cause is the temperature exceeding 100 degrees. We consider 2.1.2. acceptable because it does not have (2.2.2.), but rather, (2.1.2.) = {Switch(on), Temp_Sensor(safe), Motor(on)}.

6) Stakeholders Address ONB Problem: Translating rule 12 back to English, we have [12] TEMP(>100): {Switch(on), Temp_Sensor(safe), Motor(on)} → {Switch(on), Temp_Sensor(overheated), Motor(on)}. This rule indicates, to the stakeholders, that no provision has been specified for what happens if the motor overheats WHILE it is in an "on" state.

As an example, we will correct the requirements to address the ONB that results in the undesired state by adding a fourth requirement: RQ4: The motor shall turn off if the temperature exceeds 100 degrees. Without retracing the entire process with the addition of RQ4, we will only show the key differences, beginning with the new set of translation rules:

[1] USER(press) : Switch(off) → Switch(on)

[2] USER(releases) : Switch(on) → Switch(off)

[3] TEMP(>100) : Temp_Sensor(safe) → Temp_Sensor(overheated)

[4] TEMP(<80) : Temp_Sensor(overheated) → Temp_Sensor(safe)

[5] Switch(on) ^ Temp_Sensor(safe) : Motor(off) → Motor(on)

[6] Switch(off) : Motor(on) → Motor(off)

[7] Temp_Sensor(overheated) : Motor(on) → Motor(off)

Translation rules are converted into numeric rules (note the extra [7] T.3 rule, defining the new constraint from RQ4):

[1] USER(press) : 1.0.0. → 2.0.0.

[2] USER(releases) : 2.0.0. → 1.0.0.

[3] TEMP(>100) : 0.1.0. → 0.2.0.

[4] TEMP(<80) : 0.2.0. → 0.1.0.

[5] T.1 : 2.1.1. → 2.1.2.

[6] T.2 : 1.0.2. → 1.0.1.

[7] T.3 : 0.2.2. → 0.0.1.

When expanded and sorted for undesired next states, we obtain:

[10] TEMP(>100): 1.1.2. → 1.2.2.

[12] TEMP(>100): 2.1.2. → 2.2.2.

[13] USER(press): 1.2.2. → 2.2.2.

[15] USER(releases): 2.2.2. → 1.2.2.

State profiling on the next states produces:

State {EOD, EID, TOD, TID}

1.2.2.{ 2, 2, 2, 0 }

2.2.2.{ 2, 2, 1, 0 }

Note that state 2.2.2. is now recoverable (|TOD| = 1). There is now a system rule that recovers 2.2.2., [21] T.3: 2.2.2. → 2.2.1., which states that if the motor overheats while it is on, it will automatically turn off. This rule did not exist before because there was no requirement to address this scenario.

Support Tool Implementation

A support tool was developed to implement the approach, allowing us to conduct case studies with various sets of requirements. The tool is a Windows application that was implemented in Visual C# and uses a third-party graphing library called Graphviz to generate a graphical representation of the CCM. Among the various features the tool offers, it enables the entering of component--states, generates tanslation rules, and automatically expands those rules. The tool can even display a color-coded Labeled Transition System (LTS) from the expanded rules. The colors differentiate the environmental causes (as red transitions) from the system causes (in black). Stakeholders can manually simulate the CCM behavior by clicking the environmental causes associated with that CCM. Other displays include an LTS that only has the states transitioned by the environmental causes and those states transitioned by the system. The tool generates the results of each step in the form of an unformatted report. The tool is designed in such a way that allows for future expansion and modifications. This is achieved by partitioning functionalities into reusable components (objects) and global dedicated data structures (structs and lists). The tool's architecture uses several data structures to contain the data in its various forms as it is processed from step to step.

These include, but not limited to, separate data structures for translation rules, rules with undesired next states, and rules non-recoverable undesired next states. The data structures are shared, on a read/write basis, between data processing classes, which are also self-contained objects. This facilitates expandability since no data is tied specifically to a given process. Thus if, in the future, there is a new process to perform on the translation rules, that new process would simply access the translation rule data structure without the need to modify another process. This makes the tool both an implementation to the state profiling approach, and a development platform for future approach improvements.

Excavator Case Study

We will now show how the approach can be applied to a real-world embedded system. We also analyze the subsystem of a small excavator that is commercially sold and rented to consumers for residential use with landscaping projects. The subsystem in question is part of the excavator's operator-safety feature. The following subset of requirements is taken from an actual, commercially available excavator. For proprietary reasons, we will forgo the name of the product and rephrase the requirements in generic statements. We begin with some of the Non-Functional Requirements (NFR), followed by the Functional Requirements (FR) which specify the desired behavior of the safety feature. The operator-safety feature will consist of the following:

NFR1: A seat-bar which keeps the operator restricted to the seat. The seat-bar operates similarly to the bar restraint on a rollercoaster.

NFR2: A seat switch which is turned on when the operator is seated.

NFR3: A cab door switch which is turned on when the cab door is closed.

NFR4: A means to activate/deactivate the drive system which enables the machine to move.

NFR5: A means to activate/deactivate the bucket system which controls the excavator's earth-moving bucket. Figure 33 shows the relative position of the components described by the requirements.

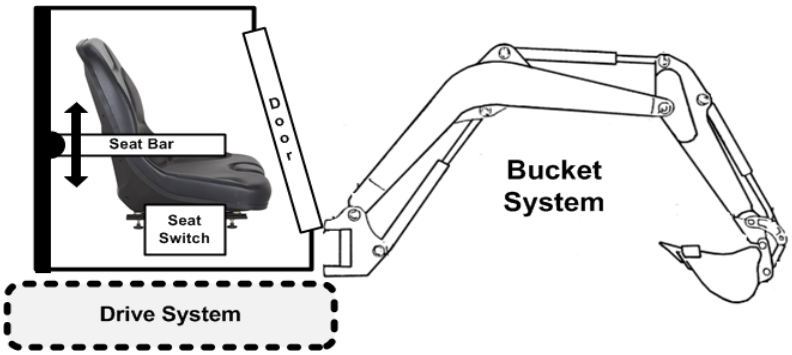


Figure 33. The components used in the excavator case study

The feature shall behave as follows:

FR1: When the operator is sitting in the seat, the machine's drive system shall be active, allowing the excavator to be driven between destinations.

FR2: If the door is closed and the seat bar is down, the machine's bucket system shall be active, allowing operation of the earth-moving bucket.

FR3: If someone outside the cab opens the door, the bucket system shall be inactive, preventing bucket operation.

From the requirements, we determine and enter the components and their respective states, as shown in Figure 34.

Components	State	State
SeatSwitch	off	on
SeatBar	up	down
Door	open	closed
Drive	inactive	active
Bucket	inactive	active

Figure 34. Component State Grid (CSG), used to enter the component states entry table

Note from Figure 34 that we refer to the entire drive system as "Drive." The scope of the requirements involve the activation and deactivation of the drive system. Therefore, for our purposes, it is sufficient to black-box the drive system without concerning ourselves with its operational details. The same applies to the bucket system. This black-boxing helps manage state explosion. A numerical value is assigned to each component--state based on the ordinal values of the component rows and the state columns. The next two steps in the translation-rule creation are achieved in a separate grid, shown in Figure 35. The creation of these rules is achieved by clicking and dragging the desired component--state from the Component State Grid to the Translation Rules Grid.

Cause	Pre Comp(state)	Next Comp(state)
OPERATOR(inSeat)	SeatSwitch(off)	SeatSwitch(on)
OPERATOR(outOfSeat)	SeatSwitch(on)	SeatSwitch(off)
OPERATOR(pullsSBDown)	SeatBar(up)	SeatBar(down)
OPERATOR(putsSBUp)	SeatBar(down)	SeatBar(up)
OPERATOR(closesDoor)	Door(open)	Door(closed)
OPERATOR(opensDoor)	Door(closed)	Door(open)
SeatSwitch(on)	Drive(inactive)	Drive(active)
Door(open)	Bucket(active)	Bucket(inactive)
Door(closed) AND SeatBar(down)	Bucket(inactive)	Bucket(active)

Figure 35. The Translation Rules Grid (TRG), used to create the translation rules by the stakeholders

The rules are formatted according to the column headings in Figure 35: Cause : Pre Comp(state) → Next Comp(state). Numerical representation of the translation rules becomes

Rules caused by the environment (the operator)

[1] OPERATOR(inSeat) : 1.0.0.0.0. → 2.0.0.0.0.

[2] OPERATOR(outOfSeat) : 2.0.0.0.0. → 1.0.0.0.0.

[3] OPERATOR(pullsSBDown) : 0.1.0.0.0. → 0.2.0.0.0.

[4] OPERATOR(putsSBUp) : 0.2.0.0.0. → 0.1.0.0.0.

[5] OPERATOR(closesDoor) : 0.0.1.0.0. → 0.0.2.0.0.

[6] OPERATOR(opensDoor) : 0.0.2.0.0. → 0.0.1.0.0.

[7] T.1 : 2.0.0.1.0. → 0.0.0.2.0.

[8] T.2 : 0.0.1.0.2. → 0.0.0.0.1.

[9] T.3 : 0.2.2.0.1. → 0.0.0.0.2.

With five components, each having two possible states, the complete system state space consists of 2 x 2 x 2 x 2 x 2 = 32. When expanded, the 9 rules above become 96 rules caused by the environment and 20 rules caused by the system, for a total of 116 expanded rules. Like the translation rules above, each expanded rule is numbered. Space prevents listing the 116 expanded rules. Each rule maps a given present state into the next state. For example, expanded rule number 61: OPERATOR(inSeat) : 1.1.2.1.2. → 2.1.2.1.2. specifies the transition from present state 1.1.2.1.2. to next state 2.1.2.1.2. The question is whether there is a rule that terminates into an undesired state. Therefore, we analyze every next state that happens to be an undesired state using state profiling to determine if that undesired state is system recoverable. There are three undesired states with which we are concerned:

{SeatSwitch(off), Drive(active)} Mask: 1.0.0.2.0.

{Door(open), Bucket(active)} Mask: 0.0.1.0.2.

{SeatBar(up), Bucket(active)} Mask: 0.1.0.0.2.

Note that, in each case, we are only stating the two component--states that define an undesired state. We want to verify that the Drive is not active unless someone is in the seat, that the Bucket is not active if the Door is open, and that the Bucket is not active if the SeatBar is up. The mask for each state is shown; again, the 0 represents a don't-care for the corresponding component--state. Using the masks and screening for a |TOD| = 0 state profile, we obtain the following list of non-recoverable, undesired states.

State Profiles: {EOD, EID, TOD, TID}

1.1.2.2.1.{3, 3, 0, 0} {SeatSwitch(off), Drive(active)}

1.2.1.2.1.{3, 3, 0, 1} {SeatSwitch(off), Drive(active)}

1.1.1.2.1.{3, 3, 0, 1} {SeatSwitch(off), Drive(active)}

1.2.2.2.2.{3, 3, 0, 1} {SeatSwitch(off), Drive(active)}

1.1.2.2.2.{3, 3, 0, 0} {SeatSwitch(off), Drive(active)}

State Profiles: {EOD, EID, TOD, TID}

1.1.2.1.2.{3, 3, 0, 0} {SeatBar(up), Bucket(active)}

2.1.2.2.2.{3, 3, 0, 1} {SeatBar(up), Bucket(active)}

State Profiles: {EOD, EID, TOD, TID}

1.2.1.2.2.{3, 3, 1, 0} {Door(open), Bucket(active)}

1.1.1.2.2.{3, 3, 1, 0} {Door(open), Bucket(active)}

1.1.1.1.2.{3, 3, 1, 0} {Door(open), Bucket(active)}

2.1.1.1.2.{3, 3, 2, 0} {Door(open), Bucket(active)}

1.2.1.1.2.{3, 3, 1, 0} {Door(open), Bucket(active)}

2.2.1.1.2.{3, 3, 2, 0} {Door(open), Bucket(active)}

2.1.1.2.2.{3, 3, 1, 1} {Door(open), Bucket(active)}

2.2.1.2.2.{3, 3, 1, 1} {Door(open), Bucket(active)}

Note that the undesired states {Door(open), Bucket(active)} are recoverable as indicated by their state profiles (|TOD| = 1 or 2), because the contiguity for recovering from that undesired state is addressed in the requirements, namely FR3: If someone outside the cab opens the door, the bucket system shall be inactive, preventing the bucket's operation. As we will see, no such contingencies are addressed for the other two undesired states. We next determine the rules that terminate at the non-recoverable, undesired states.

We focus on rules with next states of 1.1.2.2.1.{3, 3, 0, 0} {SeatSwitch(off), Drive(active)}, which consist of rules 27, 40, and 44.

[27] OPERATOR(closesDoor): 1.1.1.2.1. → 1.1.2.2.1.

[40] OPERATOR(outOfSeat): 2.1.2.2.1. → 1.1.2.2.1.

[44] OPERATOR(putsSBUp): 1.2.2.2.1. → 1.1.2.2.1.

Note that rule 40 is the only one of the three rules where the SeatSwitch changes from SeatSwitch(on) to SeatSwitch(off). If we translate rule 40 into a NL, we get OPERATOR(outOfSeat): {SeatSwitch(on), SeatBar(up), Door(closed), Drive(active), Bucket(inactive)} → {SeatSwitch(off), SeatBar(up), Door(closed), Drive(active), Bucket(inactive)} Note that rule 40 defines what causes the transition from an acceptable state {SeatSwitch(on), Drive(active)} to an undesired state {SeatSwitch(off), Drive(active)}. This transition is the one we are interested in preventing. Specifically, rule 40 reveals that the undesired state occurs when the driver gets up from the seat while the drive is active. A review of the requirements reveals that none of them state what happens if the driver gets up from the seat AFTER the drive has been activated.

The requirements only specify that the driver needs to sit down to activate the drive, however, they do not specify what should happen if the driver were to stand up. Perhaps, this contingency was assumed (ambiguous) or simply missed (incompleteness). If we determine all the other rules that terminate at a non-recoverable {SeatSwitch(off), Drive(active)} state while transitioning from an acceptable state, we find that there are three other rules that reveal a similar scenario with the same cause as rule 40, the operator moving from the seat once drive has been activated. The reason that four rules reveal the same cause for an undesired state is due to the fact that there is more than one transition path into which the undesired state can be reached. The stakeholders should respond to this information by correcting the requirements so that the requirements explicitly state what should happen if the driver moves from the seat once the drive has been activated. If we determine the rules that terminate in an

{SeatBar(up), Bucket(active)} state, we find two rules that define a change from {SeatBar(dwon), Bucket(active)} to {SeatBar(up), Bucket(active)}. They are rules 68 and 95:

[68] OPERATOR(putsSBUp): {SeatSwitch(off), SeatBar(down), Door(closed), Drive(inactive), Bucket(active)} → {SeatSwitch(off), SeatBar(up), Door(closed), Drive(inactive), Bucket(active)}

[95] OPERATOR(putsSBUp): {SeatSwitch(on), SeatBar(down), Door(closed), Drive(inactive), Bucket(active)} → {SeatSwitch(on), SeatBar(up), Door(closed), Drive(inactive), Bucket(active)}

Rules 68 and 95 reveal a similar problem as the rules surrounding the SeatSwitch: the requirements do not specify what happens if the operator lifts the SeatBar after the Bucket has been made active. Again, the stakeholders would correct this situation by explicitly specifying what should happen if the SeatBar is raised.

Observations from a Follow-up Study

This case study uses the requirements for an elevator system [163]. The requirements was prepared as a 78 page document, which includes functional and non-functional requirements, and some degree of analysis/design as expressed in UML Sequence, State, and Collaboration diagrams. For our purposes, we focused on the 25 natural language functional requirements, listed in the document. Space does not allow a complete listing of the requirements, both before and after the CCM approach was applied. In summary, the requirements describe a system that takes cab requests from buttons within the elevator cab and on each floor being serviced. The system is to deliver a cab to a floor where a request is made and then deliver the cab to the floor where the operator desires to go. Other artifacts managed include the doors, and an emergency switch in the cab. We modeled the requirements in a CCM, and analyzed them according to the steps used in the excavator case study. We will give an overview of how the CCM approach was utilized and what the approach exposed, from an off-nominal standpoint.

The elevator system consists of 18 components, as explicitly listed in the document. Each component has a detail description, from which we deduced two states for 8 components, while ten components had only the "on" state listed. For example, the components "Emergency Bell Button" and "Emergency Bell" had only the states "Pressed", and "Ring" respectively. There was no mention of the "Emergency Bell Button" not-pressed, or the "Emergency Bell" not-ringing. One inherent benefit of the CCM is that since each component has its own STD, and each STD should be completed, we are forced to account for at least two component states (e.g. {EmergencyBell(Ringing), EmergencyBell(NotRinging)}), instead of the one component state specified in the document. It can be argued that having to account for components states that are not explicitly listed, is one way to address incompleteness, and ambiguities in a requirements document. Accounting for these unspecified component states, gave us a total of 18 components with 2 component states each, giving us a total of 2^{18} (262,144) system states. Considering that each of those system states has at least two transition edges (at least one in-degree and one out-degree), this gives us a lot or transition paths to examine, both automatically.

As previously mentioned, the CCM approach allows us to reduce a large number of system states, by treating components states as edges, rather than nodes. Thus, input sensors, of which there were 9, are treated as causes, and not component states. This reduced the number of system states to 92 (512). We were able to reduce the number of states further by considering that there are 6 user indicators listed as components (e.g. Load Bell, Emergency Bell, Floor Number Display, etc.). Indicators are open-ended, meaning that they do not cause a transition in the system. Since we are interested in transition paths that can facilitate an ONB, we can ignore open-end indicators that will not contribute to the number of paths. This left us with 3 components {Controller, ElevatorEngine, DoorOpenDevice}. Note that this means the system will consist of one controller and two actuators, similar to example D of Figure 19. The Controller ended having 5 states, while the two actuators ended up with two states each.

This gave us a total of (2 x 2 x 5) 20 system states, compared to the initial 262,144 states we would have had without taking advantage of CCM's flexibility in representing different levels of abstraction. It should be noted that while it seems that the large discrepancy between 262,144 states and 20 states, would result in a loss of valuable information, the 20 states still capture the interaction between the controller and the key actuators in the system, where ONBs will tend to be a problem.

In our analysis of the Elevator case study, we looked at how the requirements addressed undesired states. The CCM approach generates all 20 possible system states, a subset of which would be considered non-acceptable (undesired states). We define an undesired system state as one that represents an undesired combination of component states. For example, the system state {DoorOpenDevice(DoorsOpened), ElevatorEngine(MoveCab), Contr(reachedRequestedFloor)}, has both the door opened while the cab is moving. This state should, ideally, not be reachable, or if reachable, it should at least be recoverable by the system. After applying the CCM approach to the elevator requirements, we divided the resulting analysis into two categories; 1) Potential ONB problems related to undesired states, 2) Potential ONB problems related to system behavior. The following list summarizes the ONB problems related to undesired states.

- Number of undesired states exposed - 7
- Number of undesired states that are reachable – 4
- Number of undesired states that can potentially be caused exclusively by the environment –3
- Number of undesired states that the system could not recover from – 5

Note that in spite of only 20 possible states, there is still about 55% that would classify as potentially problematic. This is because, during the expansion of requirements, the CCM approach will produce all the possible combinatorial system states; not just those system states implied in the requirements. The following summarizes two of the five potential ONBs related to system behaviors exposed.

An unintended behavior is that pressing an emergency stop button stops the cab movement monetarily, only to have the system start the cab moving again. This is because the requirements specifies that the elevator engine stops upon pressing the emergency button, but says nothing about halting the "send cab request", which starts the engine cab in the first. A person reading the requirement may assume that that both the engine and its request to send a cab are both tied to the emergency button. Our approach raised this question, forcing the stakeholders to explicitly state the exact result of pressing the emergency button.

Another unintended behavior is that the doors may get stuck in a 3 second cycle of opening and closing. One requirement, labeled F6 states: "A cab's door opening device opens its doors when it reaches a floor that has an active summon." While another requirement (F10) states: "After the doors of a cab have been open for 3 seconds, its door opening device closes them." No requirement explicitly states that the opening and closing occurs once, because the active summon state should be satisfied, and no longer active when the floor is reached.

The elevator requirements seem to rely substantially on common knowledge about elevator operation. Perhaps the stakeholders behind the requirements did not feel the need to fill the requirements with "trivial" knowledge. This is a common occurrence, and results in requirements lacking explicitness. We realize that the process of translating NL requirements into CCM can compound ambiguities, due to how the user of CCM interprets the requirements. However, the important thing to note is that the mere act of trying to formalize a set of NL requirements into a model, and then analyzing the behavior of that model, will raise questions that otherwise would not come to mind. Such questions were raised in the elevator case study.

Discussion and Limitations of SPA

We have introduced a new requirement verification method that can detect and help correct the off-nominal behavior susceptibility. Our case study shows that the proposed approach is able to expose

off-nominal behaviors for further correction. This section discusses further observations and lessons learned from our study as well as the limitations of our approach.

1) The approach can typically expose ONBs that are not obvious from reading the requirements and therefore beneficial to domain expects who may have a tendency to assume common knowledge about their domain of expertise while describing a system's desired behavior.

2) The distinction between environmental and system rules allows for isolating behaviors produced by a human operator, proving to be beneficial because humans are the biggest perpetrators of ONBs.

3) By taking a rules-based approach, we are able to readily translate a set of natural language requirements into a format that can be modeled, formally analyzed, and readily interpreted without prior training in a formal language.

4) The approach allows for simulation of the CCM. A user can make all the corrections via interaction with the CCM, as opposed to interacting with a large LTS representation, which helps manage visual-state explosion.

5) There are also lessons learned about the question type that should be asked during the elicitation phase. These questions include "When specifying the behavior of a given component, what should the other components be doing; does it matter?" "When specifying a component's state, should the opposite state be specified?" In many cases, the answer is yes because addressing these questions can initially reduce the number of zeroes in the translation rules which, in turn, reduces the number of ambiguities.

While our case study showed promising results, there are some limitations with our approach in terms of knowledge representation and reasoning. For knowledge representation, the CCM model itself could be improved by broadening the definition of component--state to include actions. We could follow the same component(state) format but expand it to component(state/action).

To a degree, we already include actions in practice because we can express a component in the state of performing an action. For knowledge reasoning, we can also broaden the direction of research and place more emphasis on exposing ambiguities outside the context of those related to ONBs. Another perceived limitation is that the approach does not allow for the specification of transition guards. However, the goal has been to maintain a level of abstraction that is high enough for requirements, and there is already the ability to logically AND causes together which can be used as a logical equivalence of a guard. Through further case studies with our approach, we will have a clearer understanding about if provisions for guards are necessary.

In the area of application, we demonstrate that the approach has the ability to raise specific questions about how a system will behave as specified. There is still some degree of stakeholder interaction required, however, because we are operating in the requirement phase, where stakeholder involvement is expected. We do not think that the approach needs to be completely automated, nor do we think that the approach is limited to smaller requirement sets. Finally, we address how state explosion could be minimized by reducing the number of system states and treating some components strictly as causes instead of states.

CONCLUSIONS AND FUTURE DIRECTIONS

This dissertation is the result of endeavoring to automate requirements verification. The various approaches conceived and explored can be categorized under the research heading of knowledge representation and reasoning. Overall, this research has produced a requirements representation and reasoning framework that can be used as either a front–end to an existing formal approach such as model checking, or as a standalone techniques that can be extended to expose various defects. Thus, the major contributions and merits occur in both how requirements can be represented and how that representation can be automatically reasoned upon. We now summarize the merits in these areas.

Merits in Requirements Representation

In CCM, we have a requirements model that can explicitly represent separate concerns. These include:

- The concurrent behaviors of individual systems components.
- The state transition causes originating from the system's operating environment.
- The state transition causes originating from the systems components.
- The set of rules defining how the system behaves in response to the environment.
- The set of rules defining how the system reacts to the environment.

Merits in Requirements Automated Reasoning

CCM is a rule-based approach with an IF-THEN structure that is both intuitive to most users. This allows for both the manual and automated manipulation of the rules. This includes:

- The rules can be humanly read or written, without prior training in logic and set theory.
- The rules can be cascaded and/or combined to form Kripke structures, LTSs, or Petri-nets.
- The rules can be traversed through forward chaining to allow for path traversal analysis.
- The rules can be used to create an expert system, which in turn can be queried.

- The rules themselves can be represented numerically, which allows for the limited mathematical manipulation of the rules.
- The numerical representation of the rules also allows for the direct property checking of those rules.
- There can be a direct one-to-one mapping of rules to requirements, which facilitates the translation of rules back to requirements.
- State Profiling allows for the direct property checking of individual states.

Merits in Modeling Separation of Concerns

Maintaining the separation of environmental and system causes greatly facilitates the ability to isolate one or more of these concerns for analysis purposes. For example, if desired, we can readily determine what sequence of events from the environment will result in a given state. Or we can readily determine which component will have the highest impact on the system if its behavior were to change. Or we can readily determine conflicting behaviors based on whether or not a given reaction state is associated with more than one system rule.

Merits in Addressing System ONB Susceptibility

There is merit in trying to address the off-nominal behaviors from a system's standpoint. In this research the emphasis has been on the exposure and elimination of system ONB susceptibility. This takes a different approach to the more common endeavor of preventing ONBs from arising in the operating environment. While the prevention of ONBs is equally important, the system susceptibility question is tied to the question of system robustness. Thus, in eliminating ONB susceptibility, the system is made more robust and foolproof to the actions of human operators. In applications where an embedded system is expected to greatly interact with a human operator, making the system foolproof can be critical.

Merits as a Model Checker Front End

There is merit in providing a front end to a model checker that is specifically geared toward model checking requirements. This is valuable since model checking is primarily used in formally verifying hardware, with lesser applications in software verification, and even less use in requirements engineering. CCM represents an effort toward making model checking accessible to non-technical users such as stakeholders who are domain expects but not engineers.

Impact of this Research

There are two immediate areas for potential impact resulting from this research. They are stakeholder involvement, and the addressing of ONBs. We will now address the two immediate areas individually.

Stakeholder Involvement

It is hoped that this research will have an impact on stakeholder involvement during the requirements engineering phase of a project. Quite often there is a division between stakeholder and requirements engineer that coincides with the division between informal and formal representations of the system being specified. Non-technical stakeholders tend to communicate their desires using natural language, whereas requirements engineers try to formalize those desires, in an effort to prevent them for the more formal design phase. There is also the question of verification, which is preferably done using a more automated approach. To achieve this automation, the requirements must be formalized. Furthermore, there is the question of communicating the formal verification results back to the non-technical stakeholder. Ideally, the gap between stakeholder and engineer should be eliminated, and at least greatly reduced.

Addressing ONBs

Scanning the available literature on the ONB problem reveals that not enough research is being performed to specifying systems foolproof to their operating environment.

The fool proofing often comes as an afterthought, typically addressed at the time of implementation with off nominal testing [133]. Sometimes a potential ONB problem is not caught and addressed in-house, before the product is released. The desired impact from this research is that the ONBP be addressed as early as the requirements phase. This would not only reduce the cost with addressing it downstream, but raise awareness of an ONB problem when the stakeholders are still involved in the development process.

Future Directions

Each of the above mentioned merits can be developed further by others in the requirements engineering field. To build upon this research, others can perform improvements upon the approaches as presented here and/or extend the research into various directions.

Potential Improvements

Right now the intent is maintaining the representation at a very high level of abstraction where primarily the causal relationships between components are captured and represented. This has proven to be sufficient when representing requirements, where the emphasis is on what the system is support to do. This also keeps the translation from NL requirements to CCM rules as cognitive friendly as possible. However, there may be a future need to obtain greater expressiveness within the CCM rules, at the potential expense of ease-of-use. A move in that direction would trend the CCM approach toward a rule-based design modeling language. This could be facilitated by moving away from a stand-alone support tool, to a plugin for a Simulink toolbox such as state-flow. The CCM could then in turn be converted directly into code, using Simulink's capability to automatically build implementations from models. Another potential improvement is in the change from the manual to the automatic translation from NL to CCM rules. A complete turnkey translation would require extensive work in integrating Natural Language Processing techniques.

With the present state of NLP technology, there would likely still be a need to utilize a human in the loop to some degree.

Therefore, a sufficient compromise may be to utilize a technique called Part-Of-Speech Tagging (POS Tagging) [166], in which a NL sentence is parsed and tagged according to its grammatical components. For example, Part-Of-Speech Tagger will take a sentence such as:

The grand jury commented on a number of other topics.

And tag the grammatical parts of the sentence (see the key below):

The/DT grand/JJ jury/NN commented/VBD on/IN a/DT number/NN of/IN other/JJ topics/NNS./.

The CCM translation process centers around Component(States), which in many cases can take the form of Noun(Verb). POS tagging could determine the nouns and verbs of a sentence, wherein they could then be formatted into all the combinations of Noun(Verbs) and stuffed into a drop down list of Component(States) that a stakeholder can choose from. The result, while not completely turn-key, would still aid the stakeholder in the sentence parsing portion of the translation process. For example, Given the following set of NL Requirements:

R1: The bucket boom shall raise upward when the left pedal is pushed down with the driver's toe.

R2: The bucket boom shall lower downward when the left pedal is pushed down with the driver's heel.

R3: The bucket shall tip forward when the right pedal is pushed down with the driver's toe.

R4: The bucket shall tip backward when the right pedal is pushed down with the driver's heel.

R5: The bucket boom shall become immovable if the seat switch is off.

When Part-Of-Speech Tagging is applied to those requirements, we get the following set of grammatically tagged requirements (note the tags).

R1 : DT/ The NN/ bucket NN/ boom MD/ shall VB/ raise RB/ upward WRB/ when DT/ the VBD/ left VB/ pedal VBZ/ is VBN/ pushed RP/ down IN/ with DT/ the NN/ driver NN/ ' VBZ/ s NN/ toe

R2 : DT/ The NN/ bucket NN/ boom MD/ shall VB/ lower RB/ downward WRB/ when DT/ the VBD/ left VB/ pedal VBZ/ is VBN/ pushed RP/ down IN/ with DT/ the NN/ driver NN/ ' VBZ/ s NN/ heel ./ .

R3 :DT/ The NN/ bucket MD/ shall VB/ tip RB/ forward WRB/ when DT/ the RB/ right VB/ pedal VBZ/ is VBN/ pushed RP/ down IN/ with DT/ the NN/ driver NN/ ' VBZ/ s NN/ toe ./ .

R4 : DT/ The NN/ bucket MD/ shall VB/ tip RB/ backward WRB/ when DT/ the RB/ right VB/ pedal VBZ/ is VBN/ pushed RP/ down IN/ with DT/ the NN/ driver NN/ ' VBZ/ s NN/ heel ./ .

R5 : DT/ The NN/ bucket NN/ boom MD/ shall VB/ become JJ/ immovable IN/ if DT/ the NN/ seat NN/ switch VBZ/ is RP/ off ./ .

Note in the table below that we can separate the nouns (NN) from the verbs (VB). We could further combine instances of verb-adverb (VB – RB) into one term.

Table 4. The nouns and verbs from the skid loader requirements

Nouns (NN)	Verbs (VB) - Adverbs (RB)
bucket	Raise - upward
boom	Lower - downward
driver	Tip - forward
seat	Tip - backward

From the table we could provide drop downs for nouns and verbs and let the user select value, or when further research perhaps even create the Component(State) to a high percentage of accuracy. For example, we may be able to produce: boom(raiseUpward), boom(lowerDownward), bucket(tipForward), bucket(tipBackward).

Other Directions

Since CCM is rule based, one future direction is the creation of an expert system that would allow for the querying of requirements. Such querying could occur by either forward chaining or backward chaining the rules. Forward chaining could handle questions such as "What state will the system be in if the operator performs a given sequence of events?" Whereas backward chaining could handle questions such as "Given a system state what sequence of events would produce it?" Querying requirements would enable the use of Scenario Questions (one of the published concepts that contributed to CCM).

Another direction could be to perform property checking on the CCM rules themselves, rather than using a model checker. This could be achieved by using a combination of state profiles and the numerical representation of the rules. State profiles could be used when property checking states, whereas the numerical representation would be conducive to property checking rules.

Overall, this research has resulted in a framework for the representation and analysis of requirements, with an emphasis on addressing the ONB problem from a system's standpoint. The hope is that this framework can be further enhanced into other directions and applications.

REFERENCES

[1] D. Berry, "Ambiguity in natural language requirements documents," *Lecture Notes in Computer Science*, vol. 5320, 2008.

[2] L. Mich, M. Franch, and P. Inverardi, "Market research for requirements analysis using linguistic tools," *RE Journal*, vol. 9(1), 2004

[3] B. Cheng and J. Atlee, "Research directions in requirements engineering," *Future of Software Engineering (FOSE 07)*, 2007, pp. 285-303.

[4] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A roadmap," *International Conference on Requirements Engineering*, Tutorial T7, 2000, pp. 35-41.

[5] H. P. Afshar, H. Shojai, and Z. Navabi, "A new method for checking FSM correctness," *In Proceedings of the International Symposium on Telecommunications,* August 2003.

[6] C. Damas, B. Lambeau, and A. Lamsweerde, "Scenarios, goals, and state machines: A win-win partnership for model synthesis," *In Proceedings of the International ACM SIGSOFT Symposium on the Foundations of Software Engineering FSE* , 2006, pp. 197-207.

[7] E. Letier. J. Kramer, J. Magee, and S. Uchitel, "Monitoring and control in scenario-based requirements Analysis," *In Proceedings the 27th international conference on Software engineering ICSE*, 2005, pp. 382 – 391.

[8] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts," *International Conference on Software Engineering ICSE*, 2000, pp. 304-313.

[9] D.K. Deeptimahanti and M.A. Babar, "An automated tool for generating UML models from natural language requirements," *Automated Software Engineering ASE*, 2009, pp. 680-682.

[10] B. Boehm and V. Basili, "Software defect reduction top 10 list," *IEEE Computer*, vol. 34 (1), 2001, pp. 135-137.

[11] F. Shull et al., "What we have learned about fighting defects," *In Proceedings of the International Software Metrics Symposium*, 2002, pp. 249-258.

[12] K. Wiegers, *Creating a Software Engineering Culture*, Dorset House Publishing Company, Aug. 1996.

[13] K. Wiegers, *Software Requirements, 2nd ed.*, Microsoft Press, ISBN 0735646066, 2003.

[14] H. Hofmann and F. Lehner, "Requirements engineering as a success factor in software projects," *IEEE Software*, vol. 18(4), 2001, pp. 58-66.

[15] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*, Wiley, ISBN 0471974447, 2006.

[16] I. Bray, *An Introduction to Requirements Engineering*, Addison Wesley, ISBN 0201767929, 2002.

[17] J. Ryser and M. Glinz, "SCENT: A method employing scenarios to systematically derive test cases for system test," University of Zurich, Technical Report, 1999.

[18] H. EL-Gendy and N. EL-Kadhi, "New method for testing fsm-based systems," *In proceedings of the 11th IEEE International Conference on Software, Telecommunications, and Computer Networks* ," 2003

[19] N. Maiden, "SAVRE: Scenarios for Acquiring and Validating Requirements," *Journal of Automated Software Engineering*, vol. 5, pp. 419-446, Aug 1998.

[20] T. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering,* vol. 4(3), May 1978.

[21] I. Bratko, *Prolog Programming for Artificial intelligence, 3rd ed*. Addison Wesley, 2008.

[22] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario based analysis of software architecture," *IEEE Computer Society Press*, vol. 13(6), pp. 47-55, November 1996.

[23] J. Ryser, S. Berner, and M. Glinz, "A scenario-based approach to validating and testing software systems using statecharts," *In Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA)*, 1999, p 7.

[24] J. Whittaker, *Exploratory Software Testing*, Addison Wesley, 2010.

[25] A. Sutcliffe and M. Ryan, "Experience with SCRAM, a Scenario Requirements Analysis Method," *International Conference on Requirements Engineering*, 1998, pp. 164-171.

[26] C. Damas, B. Lambeau, P. Dupont, and A. Lamsweerde, "Generating annotated behavior models from end-user scenarios," *IEEE Transactions on Software Engineering, TSE,* vol. 31(12), pp. 1056-1073, December 2005.

[27] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Monitoring and control in scenario-based requirements analysis." *In Proceedings the 27th international conference on Software engineering*, 2005, pp. 382-391.

[28] A. Sinha, S. Sutton, and A. Paradkar. "Text2Test: Automated inspection of natural language use cases," *In Proceedings of the International Conference on Software Testing (ICST),* 2010, pp. 155–164.

[29] A. Sutcliffe, N. Maiden, S. Minocha, and D. Manuel, "Supporting scenario-based requirements engineering," *IEEE Transactions on Software Engineering,* vol. 24(12), pp 1072-1088, December 1998.

[30] P. Gough, F. Fodemski, S. Higgins, and S. Ray, "Scenarios—An industrial case study and hypermedia enhancements," *In Proceedings of the second IEEE Symposium on Requirements Engineering,* 1995, pp. 10–17.

[31] T. Royer, "Using scenario-based designs to review user interface changes and enhancements," *In Proceedings of DIS95: Designing Interactive Systems*, 1995, pp. 237–246.

[32] Q.A. Malik, J. Lilius, and L. Laibinis, "Scenario-based test case generation using event-B models," *In Proceedings of the International Conference on Advances in System Testing and Validation Lifecycle*, 2009, pp. 31-37.

[33] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Transactions on Software Engineering*, vol. 29(2), pp 99-115, February 2003.

[34] A.A. Porter, L.G.Votta, V.R. Basili, "Comparing detection methods for software requirements inspections: A replicated experiment," *IEEE Transactions on Software Engineering* TSE, vol. 21(6): 563-575, June 1995.

[35] D. Gause, User DRIVEN design - The luxury that has become a necessity, *International Conference on Requirements Engineering*, 2000.

[36] D. Deeptimahanti and M. Babar, "An automated tool for generationg UML models from natural language requirements," *Automated Software Engineering*, 2009, pp. 680-682.

[37] L. Kof, "Scenarios: Identifying missing objects and actions by means of computational linguistics, "*In Proceedings of 15th International Requirements Engineering Conference*, 2007, pp. 121–130.

[38] D. Aceituna, H. Do, and S. Lee, "A human interactive approach to building requirements models," *International Symposium on Software Reliability Engineering*, fast abstract, 2010.

[39] D. Popescu, S. Rugaber, N. Medvidovic, and D. Berry, "Reducing ambiguities in requirements specifications via automatically created object-oriented models," *Monterey Workshop on Computer Packaging*, 2007, pp. 103–124.

[40] L. Kof, R. Gacitua, M. Rouncefield, and P. Sawyer, "Ontology and model alignment as a means for requirements validation," *International Conference on Software Engineering*, 2010, pp. 46–51.

[41] L. Kof, "From Requirements documents to system models: A tool for interactive semi-automatic translation," *International Conference on Requirements Engineering*, 2010, pp. 391 – 392.

[42] B. Brykczynski, "A survey of software inspection checklists," *ACM SIGSOFT Software Engineering Notes*, vol. 24(1):82, January 1999.

[43] M. Fagan. "Advances in software inspections," *IEEE Transactions on Software Engineering*, vol. 12(7):744–751, July 1986.

[44] D. Berry and E. Kamsties. *Perspectives on Software Requirements*, Kluwer Academic Publishers, 2004.

[45] F. Chantree, B. Nuseibeh, A. de Roeck, and A. Willis. "Identifying nocuous ambiguities in natural language requirements." *International Conference of Requirements Engineering*, 2006, pp. 59–68.

[46] C. Heitmeyer, R. Jeffords, and B. Labaw. "Automated consistency checking of requirements specifications," *ACM Transactions on Software Engineering and Methodology TOSEM*, vol. 5(3), pp. 231-261, July 1996.

[47] S. Sakthivel. “Survey of requirements verification techniques,” *Journal of Information Technology*, vol. 6, pp. 68-79, June 1991.

[48] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong. “Orthogonal defect classification - A concept for in-process measurements,” *IEEE Transactions on Software Engineering TSE*, vol. 18(11), pp.943–956, November 1992.

[49] J. Chaar, M. Halliday, I. Bhandari, and R. Chillarege. “Inprocess evaluation for software inspection and test,” *IEEE Transactions on Software Engineering*, TSE, vol. 19(11), pp. 1055-1070, November 1993.

[50] M. Lezak, D. Perry, and D. Stoll. “A case study in root cause defect analysis,” *International Conference of Software Engineering ICSE*, 2000, pp. 428-437.

[51] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslen. *Experimentation in Software Engineering An Introduction*, Kluwer Academic Publishers, 2000.

[52] D. Gause, “User DRIVEN Design-The luxury that has become a necessity,” *International Conference on Requirements Engineering*, 2000.

[53] D. Deeptimahanti and M. Babar. “An automated tool for generating UML models from natural language requirements,” *Automated Software Engineering*, 2009, pp. 680-682.

[54] I. Sommerville, *Software Engineering, 8th Edition,* Addison- Wesley, ISBN-10: 0137035152 2007

[55] A. Sinha. S. Sutton. and A. Paradkar. “Text2Test: Automated inspection of natural language use cases.” *In Proceedings of the International Conference on Software Testing*, 2010, pp. 155– 164.

[56] R.V. Buskirk and B.W. Moroney. “Extending prototyping,” *IBM Systems Journal*, vol. 42(4), pp. 613-623, October 2003.

[57] L.A. Cortés, P. Eles, Z. Peng. “Verification of embedded systems using a petri net based representation,” *In Proceedings. International Symposium on System Synthesis*, 2000, pp. 149-155.

[58] R. Alur, C. Courcoubetis, and D. Dill. “Model checking for real-time systems,” *In Proceedings of 5th Symposium on Logic in Computer Science*, 1990, pp. 414-425.

[59] A. Homrighausen, H. Six, and M. Winter. “Round-trip prototyping for the validation of requirements specifications,” *Conference Draft for Requirements Engineering Foundation for Software Quality (REFSQ)*, vol. 1, 2001

[60] A. Ravid and D. Berry. “A method for extracting and stating software requirements that a user interface prototype contains,” *Requirements Engineering*, Springer-Verlog, 2000, pp. 225-241.

[61] D. Zowghi, and V. Gervasi. “On the interplay between consistency, completeness, and correctness in requirements evolution,” *Information and Software Technology*, vol.45(14) pp. 993- 1009, November 2003.

[62] P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goosens. “Embedded software in real-time signal processing systems: Application and Architecture Trends,” *In Proceedings of the IEEE*, vol. 85, pp. 419–435, March 1997.

[63] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni- Vincentelli. “Design of embedded systems: Formal Models, Validation, and Synthesis,” *In Proceedings of the IEEE*, vol. 85, pp. 366–390, March 1997.

[64] D. Aceituna, H. Do, and S. Lee. “SQ2E: An approach to requirements validation with scenario Question,” *In Proceedings of the 17th Asia-Pacific Software Engineering Conference, 2010,* pp. 33-42.

[65] F. Schneider, S.M. Easterbrook, J.R. Callahan and G.J. Holzmann. “Validating Requirements for Fault Tolerant Systems using Model Checking,” *IEEE Conference on Requirements Engineering*, 1998, pp. 4-13.

[66] Nuseibeh, B., and Easterbrook, S., “Requirements engineering: A roadmap,” *International Conference on Software Engineering*, 2000, pp. 35-41.

[67] C. Jones. *Software Assessments, Benchmarks, and Best Practices*, Addison-Wesley, ISBN-10: 0201485427, 2000.

[68] H.F. Hofmann, and F. Lehner. “Requirements Engineering as a Success Factor in Software Projects,” *IEEE Software* vol. 18(4), pp. 58-66, July/August 2001.

[69] T. F. K Cynthia and M. Helweg-Larsen. “Perceived Control and the Optimistic Bias: A Meta-analytic Review,” *Psychology and Health*, vol. 17 (4), pp. 437–446, April 2002.

[70] F. P. Brooks. *Mystical Man Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)*, Addison-Wesley, ISBN-10: 0201835959, 1995.

[71] F. P. Brooks. *Design of Design: Essays from a Computer Scientist*, Pearson Education, ISBN-10: 0201362988, 2010.

[72] R.V. Buskirk and B.W.Moroney. “Extending Prototyping,” *IBM Systems Journal*, vol. 42(4), pp. 613-623, October 2003.

[73] S. Beydeda, M. Book, and V. Gruhn, editors. *Model Driven Software Development*, Springer, Berlin, IBN- 3540285547, 2005.

[74] D. Aceituna, H. Do, and S.W. Lee, “Interactive Requirements Validation for Reactive Systems through Virtual Requirements Prototype,” *Requirements Engineering Conference workshop, MoDRE* 2011, pp. 1–10.

[75] D. Aceituna, H. Do, G. Walia, and S.W. Lee, “Evaluating the Use of Model-based Requirements Verification Method: A Feasibility Study,” *Requirements Engineering Conference workshop, EmpiRE*, 2011, pp. 13-20.

[76] D. Aceituna, G. Walia, H. Do and S.W. Lee. “Model-based Requirements Verification Method: Conclusions from Two Controlled Experiments,” *Journal of Information and Software Technology,* Vol. 56, Issue 3, pp. 321-334. March 2014.

[77] D. Berry, and E. Kamsties, *Perspectives on Software Requirements*, Kluwer Academic Publishers, 2004.

[78] B. Boehm, "Get Ready for Agile Methods,With Care " *Computer*, vol. 35, pp. 64-69, January 2002.

[79] B. Boehm, W.J. Hansen, W. J. "The Spiral Model as a Tool for Evolutionary Acquisition," *The Journal of Defense Software Engineering*, May 2001.

[80] D.A. Norman. "Cognition in the Head and in the World: An Introduction to the Special Issue on Situated Action," *Cognitive Science*, vol. 17(1), pp. 1-6, October–December 1993.

[81] K. Ryan. "The Role of Natural Language in Requirements Engineering," *In Proceedings of the IEEE Int. Symposium on Requirements Engineering*, 1993, pp. 240-242.

[82] V. Lamsweerde. "Formal Specification: a Roadmap," *International Conference on Software Engineering*, 2000, pp. 147–159.

[83] M. Giese, and R. Heldal. "From Informal to Formal Specifications in UML," In Proceedings of UML2004, vol. 3273 of LNCS, Springer, 2004, pp. 197—211.

[84] Y. Ishihara, "A Translation Method from Natural Language Specifications into Formal Specifications Using Contextual Dependencies," *In Proceedings of IEEE International Symposium on Requirements Engineering*, 1993, pp. 4-6.

[85] FAA, "Requirements Engineering Management Handbook," *Office of research and technology development*, Washington DC, June 2009.

[86] S. Edwards, L. Lavagno, E. Lee and A. Sangiovanni-Vincentelli. "Design of embedded systems: Formal Models, Validation, and Synthesis," *In Proceedings of the IEEE*, vol. 85, pp. 366–390, March 1997.

[87] A. W. Brown, J. Conallen and D. Tropeano. "Models, Modeling, and Model Driven Development," In Model-Driven Software Development, 2005, pp. 1-16.

[88] M. Book, B. Sami, and V. Gruhn, editors, *Model-Driven Software Development*, Springer, Berlin, 2005, p. 464.

[89] P. Paulin, C. Liem, M. Cornero, F. Nacabal and G. Goosens. "Embedded software in real-time signal processing systems: Application and Architecture Trends," *In Proceedings of the IEEE*, vol. 85, 1997, pp. 419–435.

[90] E.M. Clarke, E.A. Emerson, A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 2, pp. 244-263, April 1986.

[91] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*, The MIT Press, Cambridge, MA, USA, ISBN-10: 0262032708, 1999.

[92] W. Sitou and B. Spanfelner. “Towards requirements engineering for context adaptive systems,” *Computer Software and Applications Conference, COMP-SAC*, vol. 2, pp. 593-600, February 2007.

[93] S. Mallick and S. Krishna., “Requirements Engineering: Problem Domain Knowledge Capture and the Deliberation Process Support,” *In Proceedings of the 10th International Workshop on Database & Expert Systems Applications, DEXA '99*, 1999, pp. 392-397.

[94] N. Ubayashi, Y. Kamei, M. Hirayama and A. Tamai. “Context Analysis Method for Embedded Systems-Exploring a Requirement Boundary between a System and Its Context*,” IEEE 19th International Requirements Engineering Conference*, 2011, pp. 143-152.

[95] A. Post, J. Hoenicke and A. Podelski. “Vacuous Real-time Requirements,” *IEEE 19th International requirements Engineering Conference*, 2011, pp. 153-162.

[96] J.C. Leite. “Requirements Validation through Viewpoint Resolution,” *IEEE Transactions on Software Engineering*, vol. 17, pp.1253-1269, December 1991.

[97] M. Sabetzadeh. *View merging in the presence of incompleteness and inconsistency*, Requirements Eng, Springer-Verlag, vol. 11, pp. 174-193, November 2006.

[98] C. A. Furia, D. Mandrioli, A. Morzenti and M. Rossi. “Modeling Time in Computing: A Taxonomy and a Comparative Survey,” *ACM Computing Surveys (CSUR)*, vol. 42(2), Article No. 6, February 2010.

[99] W. Zhao, P. Kulkarni, D. Whalley, C. Healy, F. Mueller and U Gang-ryung. “Timing the wcet of embedded applications,” *In 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004, pp. 472-481.

[100] N. Juristo, A.M. Moreno and A. Silva. “Is the European Industry Moving toward Solving Requirements Engineering Problems?” *IEEE Software*, vol. 19(6), p70, Nov /Dec2002.

[101] C.W. Johnson. “The Utility Space of Requirements Engineering,” doi:10.1.1.38.6227

[102] R. Alur and A. Chandrashekharapuram. “Dispatch sequences for embedded control models.” *11th IEEE Real-Time and Embedded Technology and Applications Symposium*, vol.11, pp.508–518, November 2005.

[103] B. Brykczynski. “A survey of software inspection checklists,” *ACM Software Engineering Notes*, vol. 24(1), pp. 82–89, January 1999.

[104] L. A. Cortes, L. Alej, R. Corts, P. Eles, and Z. Peng. “Verification of embedded systems using a petri net based representation,” *In Proceedings of 13th International Symposium on System Synthesis*, 2000, pp. 149–155.

[106] D. Deeptimahanti and M. Babar. “An automated tool for generating UML models from natural language requirements,” *Automated Software Engineering*, 2009, pp. 680–682.

[107] M. Fagan. “Advances in software inspections,” *IEEE Transactions on Software Engineering*, vol. 12(7), pp.744–751, July 1986.

[108] D. Gause. “User DRIVEN design - The luxury that has become a necessity,” *In Proceedings of 4th International Conference on Requirements Engineering*, 2000.

[109] C. Heitmeyer, R. Jeffords, and B. Labaw. “Automated consistency checking of requirements specifications,” *ACM Transactions on Software Engineering and Method*, vol. 5(3), pp. 231–261, March 1996.

[110] M. Leszak, D. E. Perry, and D. Stoll. “A case study in root cause defect analysis,” *In Proceedings of the 22nd International Conference on Software Engineering*, 2000, pp. 428–437.

[111] A. Porter and L. Votta. “Comparing detection methods for software requirements inspections: A replication using professional subjects.” *IEEE Transactions on Software Engineering*, vol. 21, pp. 563–575, June 1995.

[112] S. Sakthivel. “Survey of requirements verification techniques,” *ACM Transactions on Software Engineering and Method*, 1991, pp. 68–79.

[113] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, 2000.

[114] A. A. Porter and L. G. Votta. “An experiment to assess different defect detection methods for software requirements inspections,” *International Conference on Software Engineering (ICSE)*, 1994, pp. 103-112.

[115] G. M. Schneider, J. Martin, and W. T. Tsai. “An experimental study of fault detection in user requirements documents,” *ACM Transactions on Software Engineering and Method*, vol. 1(2), pp.563-575, April 1992.

[116] X. Franch, A. Maté, J. Trujillo, and C. Cares. “On the joint use of i* with other modelling frameworks: A vision paper,” *IEEE 19th International Requirements Engineering Conference*, 2011, pp. 133-142.

[117] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. “Goal and scenario validation: A fluent combination,” *Requirements Engineering Journal* (Springer), vol. 11(2), pp. 23-137, December 2005.

[118] J. Carver, N. Nagappan, and A. Page. “The impact of educational background on the effectiveness of requirements inspections: An empirical study,” *IEEE Transactions on Software Engineering*, vol. 34(6), pp. 800-812, June 2006.

[119] B. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[120] J. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *Transaction on Software Engineering (TSE)*, vol. 19, pp. 24–40, January 1993.

[121] A. Bierel, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. “Bounded model checking,” *Advances in Computers*, vol. 58, pp. 117–148, December 2003.

[122] B. Boehm. “Anchoring the software process,” *IEEE Software*, pp.73-82, July 1996.

[123] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. "Model checking large software specifications," *Transaction on Software Engineering (TSE)*, vol. 24(7), pp. 156–166, July 1998.

[124] K. Cheung. "Verify SCR requirements using XSPIN model checking to elevator case study," 2001, CiteSeer.psu:10.1.1.212.8694.

[125] Y. Choi and M. Heimdahl. "Model checking RSML-e requirements," *IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2002, pp. 109–118.

[126] Y. Choi, S. Rayadurgam, and M. Heimdahl. Toward automation for model-checking requirements specifications with numeric constraints. *Requirements Engineering Journal*, vol. 7(4), pp. 225–242, September 2002.

[127] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. "NuSMV: A new symbolic model verifier," *In Proceeding of International Conference on Computer-Aided Verification (CAV'99)*, 1999, pp. 495–499, 1999.

[128] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. "Verification of the Futurebus+ Cache Coherence Protocol," *Formal Methods in System Design (FMSD)*, vol. 6(2), pp. 217–232, March 1995.

[129] E. Clarke, D. Long, and K. Mcmillan. *Compositional model checking*. MIT Press, 1999.

[130] M. Dwyer, G. Avrunin, and J. Corbett. "Property specification patterns for finite-state verification," *In Proceedings of the second workshop on Formal methods (FMSP)*, 1998, pp. 7–15.

[131] M. Dwyer, G. Avrunin, and J. Corbett. "Patterns in property specifications for finite-state verification," *In Proceedings of the International Conference in Software Engineering (ICSE)*, 1999, pp. 411–420.

[132] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. "Assisting requirement formalization by means of natural language translation" *Formal Methods in System Design (FMSD)*, vol. 4(3), pp. 243–263, March 1994.

[133] D. Foyle and B. Hooey. "Improving evaluation and system design through the use of off-nominal testing: A methodology for scenario development," *In Proceedings of the 12th International Symposium on Aviation Psychology*, 2003, pp. 397–402.

[134] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. "Model checking early requirements specifications in tropos," *In proceedings of International Conference on Requirements Engineering*, 2001, pp. 174–181.

[135] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. "Using abstraction and model checking to detect safety violations in requirements specifications," *Transaction on Software Engineering (TSE)*, vol. 24(11), pp. 927–948, November 1998.

[136] 1S. Konrad and B. Cheng. "Facilitating the construction of specification pattern-based properties," *In proceedings of International Conference on Requirements Engineering*, 2005, pp. 329–338.

[137] N. Leveson. “The role of software in recent aerospace accidents,” *In Proceedings of the 19th International System Safety Conference*, 2001.

[138] N. Leveson, G. Leveson. “Systemic factors in software-related spacecraft accidents,” *AIAA Space 2001 Conference and Exposition*. 2001.

[139] S. Mallick and S. Krishna. “Requirements engineering: Problem domain knowledge capture and the deliberation process support.” *In proceedings of Tenth International Workshop on Database and Expert Systems Applications (DEXA)*, 1999, pp. 392–397.

[140] T. Sreemani and J. Atlee. “Feasibility of model checking software requirements: A case study,” *In proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96)*, 1996, pp. 77–88.

[141] U. Stern and D. Dill. “Automatic verification of the SCI cache coherence protocol,” *In Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995, pp. 21–34.

[142] K. Suda and N. Rani. “The importance of risk radar in software risk management: A case of a Malaysian company,” *International Journal of Business and Social Science (IJBSS)*, vol. 1(3), pp. 262-272, March 2010.

[143] N. Ubayashi, Y. Kamei, M. Hirayama, and T. Tamai. “A context analysis method for embedded systems - Exploring a requirement boundary between a system and its context.” *In proceedings of International Conference on Requirements Engineering*, 2011, pp. 143–152.

[144] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. “Peephole partial order reduction,” *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'08)*, Springer 2008. LNCS 4963, pp. 382-396.

[145] E. Yu. “Social modeling and I*,” *In Conceptual Modeling: Foundations and Applications Lecture Notes in Computer Science*, vol. 5600, pp. 99–121, 2009.

[146] H. Giese, I. Kruger, “A Summary of the ICSE 2004 Workshop on Scenarios and State Machines: Models, Algorithms, and Tools.” *SIGSOFT Software Engineering Notes*. vol. 30(1), p. 2, January 2005.

[147] S. Verma, S. Lozito, K. Thomas, D. Ballinger, “Procedures for Off-Nominal Cases: Very Closely Spaced Parallel Runway Operations.” *Digital Avionics Systems Conference*, 2008, pp. 2.C.4-1–2.C.4-11.

[148] G.C. Fraccone, V. Volovoi, A.E. Colón, M. Blake, “Novel Air Traffic Procedures: Investigation of Off-Nominal Scenarios and Potential Hazards.” *Journal of Aircraft*, vol. 48(1), pp. 127-140, January-February 2011

[149] M.J. Armstrong, “Identification of Emergent Off-Nominal Operational Requirements During Conceptual Architecting of the More Electric Aircraft.” *BiblioLabsii*. August 2012.

[150] E.M. Clarke, E.A. Emerson, A.P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.” *ACM Transactions on Programming Languages and Systems.*, vol. 8(2), pp. 244-263. April 1986.

[151] J.C. Day, K. Donahue, A. Kadesch, A.K. Kennedy, E. Post, "Modeling Off-Nominal Behavior in SysML." *The American Institute of Aeronautics and Astronautics (AIAA) Infotech 2012*, 2012, pp. 19-21.

[152] S. Thummalapenta, J. DeHallex, N. Tillmann, S. Wadsworth, S. "DyGen: Automatic Generation of High-Coverage Tests via Mining Gigabytes of Dynamic Traces." *Lecture Notes in Computer Science*, vol. 6143, pp. 77-93, July 2010.

[153] M.A. Neerincx, "Situated Cognitive Engineering for Crew Support in Space." *Personal and Ubiquitous Computing*, vol. 5, pp. 445-456, August 2011

[154] J.B. Cohen, D. Plakosh, K. Keeler, "Robustness Testing of Software-Intensive Systems: Explanation and Guide." *CMU Software Engineering Institute. Technical Report Number: CMU/SEI-2005-TN-015*, April 2005.

[155] D.C. Foyle, B.L. Hooey, B.F. Gore, C.D. Wickens, S. Scott-nash, C. Socash, E., Salud, C. David, *Modeling Pilot Situation Awareness. Human Modelling in Assisted Transportation*, Springer, Heidelberg, Germany . 2010.

[156] D.L. Long, L.A. Clarke, "Task Interaction Graphs for Concurrency Analysis," *In proceedings of the International Conference on Software Engineering (ICSE)*, 2012, pp. 44-52, 2012.

[157] F. Ortmeier, G. Schellhorn, "Formal Fault Tree Analysis: Practical Experiences." *Electronic Notes in Theoretical Computer Science, Elsevier*, vol. 185, pp. 139-151, July 2007.

[158] T.A. Wei-tek, R. Paul, C. Fan, L. Yu, "Automated Event Tree Analysis Based-on Scenario Specifications," *In proceedings of IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2003, pp. 47-74.

[159] L.M. Ridley, J.D. Andrews, "Application of the Cause-Consequence Diagram Method to Static Systems," *Reliability Engineering and System Safety, Elsevier*, vol. 75(1), pp. 47-58. January 2002

[160] T. Prevot, J.M. Homola, J. Mercer, M., Mainimi, C. Cabrall, "Initial Evaluation of NextGen Air/Ground Operations with Ground-Based Automated Separation Assurance." *Eighth USA/Europe Air Traffic Management Research and Development Seminar (ATM2009)*, 2009.

[161] D. Aceituna, H. Do, S. Srinivasan, "A Systematic Approach to Transforming System Requirements into Model Checking Specifications." *International Conference on Software Engineering. (ICSE)*, 2014, pp. 165-174.

[162] M. Gupta, D. Aceituna, G. Walia, H. Do. "Evaluating the Use of Model-Based Requirement Verification Method: An Empirical Study," *The 26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2014.

[163] Kalaidjian, A. Software Requirements Specification Elevator System Controller 2011, Sample Requirements Prepared for D.M. Berry, University of Waterloo, Canada. (2001).

[164] Sample Case Study Requirements, Canal Monitoring and Control System 2011, Model-Driven Requirements Engineering Workshop, Requirements Engineering Conference (2011).

[165] D. Aceituna, “Survey of Concerns in Embedded Systems Requirements Engineering” *SAE International. Journal of Passenger Cars-Electronic Systems*. vol. 7, pp. 1-13, May 2014.

[166] T. Brants, “TnT - a statistical part-of speech tagger,” *In Proceedings of the Sixth Applied Natural Language Processing Conference (ANLP)*, 2000, pp. 224-231.

[167] K.G. Larsen, P. Pettersson and W. Yi, “UPPAAL: Status Developments,” *In proceedings of the 9th International Conference on Computer Aided Verification CAV'97*, 1997, pp. 456-459.