Exposing the Susceptibility of Off-Nominal Behaviors in Reactive System Requirements

Daniel Aceituna
DISTek Integration, Inc.
North Dakota State University
Computer Science Department
Fargo, ND, USA
daniel.aceituna@ndsu.edu

Hyunsook Do
North Dakota State University
Computer Science Department
Fargo, ND, USA
hyunsook.do@ndsu.edu

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 incomplete to account for every possible, unintended, off-nominal behavior (ONB) that the operating environment can create in the system. In this paper, we present a semi-automated approach, based on the causal component model (CCM), which can expose, within a set of requirements, whether ONBs can result in undesired system states. We demonstrate how the CCM approach exposes and helps address potential off-nominal behavior problems in a set of requirements that represents a real-world product. Our case study shows that the approach can expose susceptibility to ONBs and can supply information useful in correcting a set of requirements.

Index Terms- Off-nominal behaviors, requirements, reactive systems, concurrency.

I. INTRODUCTION

In reactive systems where there are considerable interactions with the system's operating environment, it is not unusual for the environment to behave in a manner that is unaccounted for by the system's 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 system designers consider normal behavior for the operator. Stakeholders typically make assumptions when specifying a set of system requirements. The 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 [25]. We refer to these unintended behaviors as off-nominal behaviors (ONBs) [7, 22].

If we model system behavior as a set of system states transitioning from one state to another in reaction to the system's operating environment, ONBs can result in undesired states in one of three ways. 1) The environment behaves faster than the system can react, causing the system to exhibit an unintended path transition. 2) Two separate elements in the environment (sensor, human, etc.) simultaneously invoke a system reaction, causing an unpredictable transition, with no specified contingency to handle address it. 3) An environmental element, such as a sensor, gets stuck in a given state, reducing a set of possible behaviors that the system can enact, possibility resulting in an unintended transition into an undesired state. 4) A human operator, not following the intended "nominal" procedures, causes a sequence of state transitions from which no recoverable contingencies have been specified.

At the heart of our research and subsequent approach are the following questions: *What undesired states can the system's operating environment cause due to a lack of explicitness in the requirements? What have the stakeholders missed 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 problems.

Due to requirement incompleteness, even a simple set of requirements can result in a system that contains more susceptibility to ONBs than anticipated. 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 then starts counting the seconds and turns off the motor 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. 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-nominal manner? Will the system be placed in an undesired state from which it cannot recover? Is the system foolproof with enough contingencies to avoid an undesired state from an operator or from the environment in general?

To date, some researchers have tried to address the ONB problem [2, 3, 4, 6], but most attempts to resolve this problem have focused on how the human operator reacts to off-nominal situations within the operating environment [7]. Our approach addresses the ONB problem from the system's perspective by translating a set of natural language (NL) requirements into a set of rules, which is then used to expose any undesired, non-recoverable states that are unintentionally producible by the requirements.

In our prior work [26], we addressed ONB problems using a model checker. In that work, we used a proposed requirement modeling technique called the causal component model (CCM) [20] as a front end to a model checker [26]. The CCM captured the requirements and translated them into a NuSMV symbolic model-checking script (along with some temporal properties that

were specific to the ONB problem). The approach 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 think that ONB corrections could be deduced more easily if we were not restricted to counter-examples as the only source of information.

There are other potentially useful metrics confined to the model checker's internal operation, such as the number of global system states that contains an undesired combination of component-states, all traversal paths other than counter-examples resulting from failed properties, and whether an undesired state is recoverable by the system. Furthermore, there are two limitations to a model checker when it comes to addressing ONBs. The first limitation is that a model checker internally models a system's state transitions without explicitly labeling what caused those transitions. When dealing with ONBs, we want to algorithmically process the transition causes that come from the environment, thus we need to label those causes in the transition system. The second limitation is that in a model checker's representation of a system, not all possible system states are observable; the model checker's transition system lacks granularity [9]. Both of the aforementioned limitations can be addressed by converting the CCM into a set of rules that explicitly states environmental causes and maps all the possible system state transitions.

Using rules allows for the direct interpretation of results, not only for exposing ONBs, but also for information suggesting how to avoid them. The rules also allow for a more direct ONB exposure, as opposed to having to analyze a system's entire state space as with the model checker, which helps to mitigate the state-explosion problem and to reduce computation time. To investigate whether our approach can expose ONB problems, we implemented the approach and performed a case study using a commercial mini-excavator. The results showed that our approach exposed two instances of ONBs with a suggestion about how to fix them.

The rest of this paper is structured as follows. In Section II, we present the background information and existing work that is relevant to our approach. Section III describes the proposed approach. In Section IV, we examine the case study where we applied the CCM approach to real-world requirements. In Section V, we discuss the results and limitations. Section VI summarizes the results of our research in the conclusion.

II. Background and Related Work

ONBs can occur for various reasons [13]. 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, because human behavior cannot be fully predictable and is prone to unexpected behaviors [2]. Many efforts have been made in the area of 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 [13]. Verma et al.'s study utilizes off-nominal behavior test cases for airplane runway operations to expose possible ONBs [2]. As with testing in general, off-nominal behavior testing can have limited effectiveness because exhaustive testing is not practical [12].

It is not unusual for a reactive 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" [10]. "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 [7]. 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 [10, 12]. Of course, we are not excluding the use of formal methods that often strive to prove the correctness of a system's critical parts [15] but typically require a considerable learning curve.

A major industry that has invested considerable efforts to address ONBs is aviation where erroneous human behavior can have catastrophic results [23, 24]. 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 [4]. 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 [11]. Giese and Kruger have suggested an iterative methodology to develop a scenario specification from an initial set of nominal scenarios, which is subsequently generalized and used to produce additional off-nominal scenarios [1]. Fraccone et al. have used simulation-based models to create off-nominal conditions for air-traffic procedures [3]. Scenarios and simulation-based models have also been used to anticipate what contingencies have not specified and improve the system's robustness, making them more "foolproof" [8].

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 that focuses on the requirement phase). The other methods include fault tree analysis (FTA) [16], event tree analysis (ETA) [17], cause-consequence analysis (CCA) [18], and the use of model checking [9, 26]. ETA requires historical knowledge of an existing system, whereas we are trying to assess unintended behaviors from requirement incompleteness. FTA uses a cause-and-effect diagram with digital logic symbols to deal with known failures and tries to assess the root cause [16]. CCA integrates fault trees and event trees to predict the effect of a failure scenario.

All these methods require knowledge that pertains to an established 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 reachability 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 rule-based approach can readily address these questions without needing the temporal-logic knowledge required for model checking [5].

III. The CCM Approach

Our approach operates on the premise that incompleteness 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 have 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.

A. CCM Overview

We now describe, in detail, the major steps involved with the CCM approach as shown in Figure 1. The squares in Figure 1 refer to the artifacts produced while the numbered ovals describe the processes producing the artifacts.

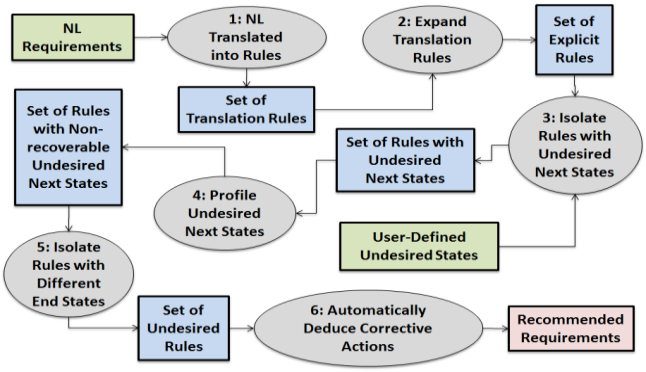

Figure 1. Overview of the CCM Approach.

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

1) 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 process explained in Section III.B. A translation rule specifies the transition between two given system states and is expressed in the form of a mapping function: *Transition_Cause: Present_State → Next_State.* The *Transition_Cause* enacts the transition between two component-states and 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 defines 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" means to determine 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 ONBs from the environment can cause an undesired state.

6) Stakeholders Address ONB Problems: From the final set of undesired rules, we create a list that describes 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 ONBs.

B. 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 into one another using rules that are 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 those undesired states. We now formally define these concepts and operations. We explain them using an example.

Component-States and System States: We view a system as a set, C, of interacting components, $c \in C$, that can each assume a component-state, s, expressed as $c(s)$. A system state, gs, is a global state that represents the simultaneous status of all component-states at a given instance of time and can be expressed as follows: $gs = (c_1(s_n), c_2(s_n), c_3(s_n), \ldots, c_n(s_n))$. To facilitate algorithmic manipulation, we can assign both Component-states and system states a numeric value that is derived from how the component-states are captured in an entry 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 1 illustrates the numeric values (e.g., 0.2.0.) generated when entering three components and two states. From Table 1, we derive $c_2(s_1) = 0.1.0.$ because c_1 is in the second row (thus second position) and s_1 is in the first column (thus, a value of 1). All numeric values have three positions because there are three components. Note that the zeroes are an indication of requirement incompleteness because they represent component-states that have not been explicitly defined.

Table 1: Component-state entry table showing the numeric value of component-states

	S_1	S_2	$\ldots S_n$
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 allow for the logical union of component-states into one number, e.g., $c_1(s_1) \wedge c_3(s_2)$ allows for *1.0.0. + 0.0.2. = 1.0.2.*, where 1.0.2. = {c_1(1), c_2(0), c_3(2)}.

The Translation Rules: Our approach begins with the manual translation of requirements into translation rules, which we now define. Let E_C be the set of all possible causes originating from the environment and $ec \in E_C$ 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*) : $ECR = \{ecr \mid ecr = (cc : cs_p \rightarrow cs_n), cs_p \in C_n(S_p), cs_n \in C_n(S_n)\}$, where (cc = ce, ce $\in Ec$) or (cc = ce $\wedge cs_2 \wedge \ldots \wedge cs_n$, $cs_n \notin C_n(S_p)$, $cs_n \notin C_n(S_n)$). 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 is read as "*ecr* causes component *C* to transition from its present state, s_p, to its next state, s_n." Recall that component-states, CS_p and CS_n, can assume a numerical value, yielding, in a system with three components, for example: ec: 1.0.0. → 2.0.0. We define a set of system caused translation rules as $SCR = \{scr \mid scr = sc : cs_p \rightarrow cs_n, cs_p \in C_n(S_p), cs_n \in C_n(S_n)\}$, where (sc = cs_x, $cs_x \notin C_n(S_p), cs_x \notin C_n(S_n)$) or ($sc = cs_1 \wedge cs_2 \wedge \ldots \wedge cs_n$, $cs_n \notin C_n(S_p)$, $cs_n \notin C_n(S_n)$). Numerically, an *ECR* member can have the form *ec*: 0.1.0. → 0.2.0., whereas an *SCR* member can be 0.0.3. ∧ 1.0.0. : 0.1.0. → 0.2.0. and can be further simplified to 1.0.3. : 0.1.0. → 0.2.0.

Absorption and Propagation Operations: In numerical form, a translation rule contains zeroes which denote multiple interpretations for the rule. Translation rules must be explicit through what we call expansion. Before the translation rules are expanded, we require that they be subjected to a process called absorption which isolates the cause on the rule's left side. For members of *ECR*, rules in the form of ec $\wedge c_x s : c_y s_p \rightarrow c_y s_n$, must be converted to ec : $c_x s \wedge c_y s_p \rightarrow c_y s_n$ $ec : c_x s \wedge c_y s_p \rightarrow c_y s_n$. For *GSR* members, rules in the form of tc $\wedge c_x s : c_y s_p \rightarrow c_y s_n$ become tc *:* $c_x s \wedge c_y s_p \rightarrow c_y s_n$, 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 ∧ 0.0.3. ∧ 1.0.0. : 0.1.0. → 0.2.0. becomes tc : 1.1.3. → 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. → 0.2.0. becomes tc : 1.1.3. → 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.

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 CS_n be a set of component-states for a given component and $Card = \{|CS_1|, |CS_2|, \ldots, |CS_n|\}$ be the set of each component-state's cardinality. The system state space, *SSP*, is the Cartesian product $SSP = Card \times Card$. 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, . . ., ss_n}. We expand the translation rules by applying the following algorithm to both *ECR* and *SCR*, one set at a time.

1. **FOR EACH** ordered pair ‹cs_n, ss_n› derived from {cs_1, cs_2, . . . , cs_n} x {ss_1, ss_2, ... , 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** ($cs_n = 0 \wedge ss_n \neq 0$), **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›, of *ECR* and *SCR* 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 2 shows a given state with its in/out-degrees.

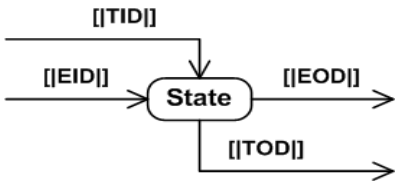

Figure 2. System state with labeled in/out-degrees.

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 : sep → sen
4. **LET** TC be the set of all rules: tc : stp → stn
5. **LET** S be the set of all system states, s
6. **FOR EACH** system state, s ∈ 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|}. If state 1.2.1. has |EOD| = 1, |EID| = 2, |TOD| = 1, and |TID| = 3, we write 1.2.1.{1, 2, 1, 3}.

C. CCM Example

Due to requirement incompleteness, 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 set of 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.*

We will apply the process outlined in Figure 1 to the requirements we just stated. For clarity, we will use the same process headings 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) [19, 21, 26]. Figure 3 shows a CCM that models the translation rules. The interrelationships among the three STDs occur on the basis of causation.

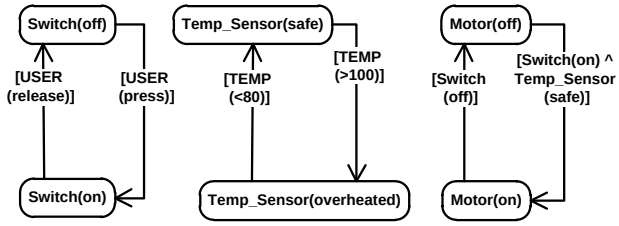

Figure 3. A causal component model (CCM) of the requirements.

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 Problems: 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.

D. Approach Implementation

We have developed a tool that implements our 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 graphic library called Graphviz to generate a graphical representation of the CCM. Among the various features the tool offers, it enables entering of component-states, generates translation rules, and automatically expands those rules. The tool can 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 behaviors 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.

E. Scalability of the CCM Approach

The CCM approach has the potential to generate a large set of rules for a given set of requirements. This raises the question of scalability. There are currently two approaches that can address the scalability issue. 1) Using different levels of abstraction. 2) Creating the rules from a subset of the total component states. We explain each of them as follows.

1) Levels of Abstraction: 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. For example, given the two sets of rules: USER(Presses) : Switch(off) → Switch(on) and Switch(on) : Motor(off) → Motor(on), we can condense the two rules into one rule by treating Switch(on) as an environmental cause rather than a component-state. Thus, we condense the two rules into the one rule: SWITCH(userPressesOn) : Motor(off) → Motor(on).
2) State Space Restriction: recall in the rule expansion process of section III-C, we used the set of every possible state in the system, a number determined from the permutation of all the possible component-states. However, the rules can be expanded by using smaller subsets of all the possible component-states. We can partition the entire state space into smaller subsections, and independently analyze those subsections, using a "divide and conquest" approach. For example if the switch has two states, the motor two states, and the temp sensor two designed states (hot and cold), the total number of possible system states is {2 x 2 x 4} = 16 states. If we analyze the requirements only when the temp sensor is hot, we now have {2 x 2 x 1} = 4 states.

IV. Case Study

To evaluate our approach, we applied the CCM approach to a real-world reactive system. The system is an actual, commercially available excavator that can be rented to consumers for use with landscaping projects. The system consists of more than 100 requirements, but in this study, we focus on those pertaining to the excavator's operator-safety feature. 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) that specify the desired behavior of the safety feature. The operator-safety feature will consist of the following requirements:

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 5 shows the relative position of the components described by the requirements.

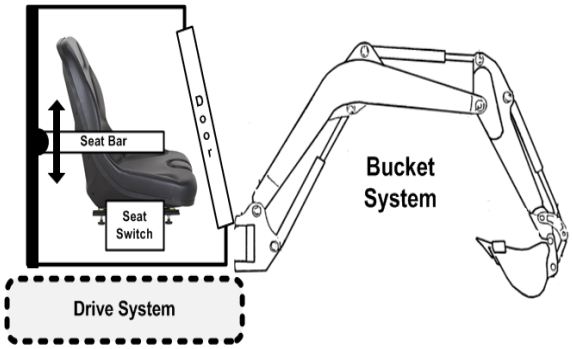

Figure 5. The components used in the excavator case study.

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 6.

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

Figure 6: The tool's Component State entry Grid (CSG) used to enter the component-states.

Note from Figure 6 that we refer to the entire drive system as "Drive." The scope of the requirements involves the activation and deactivation of the drive system. Therefore, for our purposes, it is sufficient to abstract the drive system without concerning

ourselves with its operational details. The same applies to the bucket system. This abstracting 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 7. 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 7: The tool's translation rules entry grid (TRG) used by the stakeholders to create the translation rules.

The rules are formatted according to the column headings in Figure 7: *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 states 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 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.

V. Discussion And Limitations

Our approach is designed to address the ONB problem from a system susceptibility standpoint, which amounts to making a system more robust and foolproof as early as the requirements phase. This facilitates the involvement of stakeholders who are domain experts and are able to access those contingencies that the system may have to react to. We view our approach as contributing to a defensive approach to system specification.

Further observations from using our approach have revealed that correcting ONB problems in a set of requirements often involves adding one or more requirements to account for a missing contingency. Thus, it appears that many, if not most, ONB problems can stem from a lack of explicitness in the requirements, either due to ambiguities and/or incompleteness. This apparent connection between ONBs and incompleteness suggests that the CCM approach can be further extended to specifically address incompleteness in the requirements.

Since CCM produces a rule-based representation of the requirements, we can use the rules to create a set of scenarios to help validate the requirements. We can also translate the rules back into a structured English (IF-THEN) form of the requirements to facilitate the creation of implementation test cases. We can even apply rule-based system techniques such as backward chaining, as a means of starting from an undesired state and determining all the scenarios that can lead to that state.

Finally, the CCM approach can potentially be applied to other deliverable documents such as operation manuals, and handling procedures, where the lack of accounted contingencies and the assumption of nominal operator behavior can be prevalent.

While our study shows promising results, there are still some limitations that remain with the CCM approach. The approach is semi-automated, since human intervention is needed to initially translate the requirements into rules, define the undesired states, and manually correct the original set of requirements, in response to the ONB susceptibilities exposed. There is also a question of whether the use of component states and rules is sufficiently expressive to model any set of embedded system requirements. We have purposely tried to maintain the CCM vocabulary as small as possible to promote greater stakeholder involvement. And there is a scalability issue that we have already discussed in section III. To address those limitations, we have considered the following resolutions.

There are two ways to address the issue of human intervention during the creation of rules. The intervention can either be purposely factored into the approach, or mitigated via automation. Human intervention can be factored in if we produce the component-states and related rules directly from stakeholders' inputs through elicitation sessions rather than from a requirements document. During the elicitation session, a group of stakeholders would follow the four step process that produces the translation rules. The requirements would be directly constructed into a CCM, and would complement a model driven development paradigm. During elicitation, human intervention has the advantage (over an automated approach) of allowing the stakeholders to collectively evaluate the selections made for component-states, and the system's behaviors.

To semi-automate human translation, we plan to use Part-Of-Speech (POS) tagging as a means to semi-automate the translation process from a requirements document. The POS tagger would produce component candidates (nouns) and state candidates (e.g., verbs). Other techniques in natural language processing (e.g., relationship extraction and discourse analysis) have been considered toward automating the creation of the rules.

The question of model expressiveness will have to be addressed as more case studies are performed. We plan to carry out a series of controlled experiments with human subjects in which typical characteristics of embedded systems (e.g., interrupts) are modeled using CCM. Through these experiments, we can determine if the CCM vocabulary must be expanded. As mentioned earlier, our goal has been to keep the vocabulary as small as possible.

VI. Conclusion

Requirement engineering has a unique position in the software-development cycle. Unlike the rest of the development cycle, the requirement phase deals directly with the problem domain and must do so using an informal means of representing the problem. This lack of formality introduces ambiguities and incompleteness that can lead to unintended system behaviors that are traditionally exposed and addressed in a latter phase at a higher cost. Our goal with this research has been to expose and address a system's susceptibility to off-nominal and unintended behaviors during the requirement phase.

We address this problem by first translating a set of requirements into a set of rules. We then expand those rules into a larger, complete set of rules that defines both the behaviors (in the form of transitions) the stakeholders intend and the unintended behaviors. From there, we determine whether the rules can produce undesired states and whether those states are recoverable by the system. The approach also exposes what environmental cause results in the undesired, non-recoverable states that, in turn, help the stakeholder formulate requirement corrections.

Our approach is facilitated by several contributions. We develop a rule-based modeling technique, CCM, that formalizes a set of requirements at a high enough abstraction level that, with

minimal training, is understandable by stakeholders while still formal enough to allow for automated analysis. CCM also abstracts the causes originating from the environment that, in turn, enables us to automatically match environmental causes to both intended and unintended behaviors. As a modeling technique, CCM could be used with other rule-based applications.

We developed the technique of filling a system state space with every possible transition that is. Finally, we showed how our approach can expose and help address potential unintended behaviors that could compromise the safety features of a real-world application.

ACKNOWLEDGMENTS

We want to thank Dr. Sudarshan Srinivasan for his input in the area of transition systems. This work was supported, in part, by NSF CAREER Award CCF-1149389 to North Dakota State University.

REFERENCES

[1] H. Giese, I. Kruger, "A Summary of the ICSE 2004 Workshop on Scenarios and State Machines: Models, Algorithms, and Tools." SIGSOFT Software Engineering Notes. January 2005.

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

[3] G.C. Fraccone, V. Volovoi, A.E. Colón, M. Blake, "Novel Air Traffic Procedures: Investigation of Off-Nominal Scenarios and Potential Hazards." 2011. Journal of Aircraft, Vol. 48, No. 1, pp. 127-140.

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

[5] 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. 1986, Vol. 8, pp. 244-263.

[6] D.C. Foyle, B.L. Hooey, "Improving Evaluation and System Design Through the Use of Off-Nominal Testing: A Methodology for Scenario Development." Proceedings of the 12th International Symposium on Aviation Psychology. 2003, pp. 397-402.

[7] J.C. Day, K. Donahue, A. Kadesch, A.K. Kennedy, E. Post, "Modeling Off-nominal behavior in SysML." AIAA Infotech 2012. June 2012, pp. 19-21.

[8] D. Aceituna, H. Do, and S. Lee, "SQ2E: An Approach to Requirements Validation with Scenario Question." Proceedings of the 17th Asia-Pacific Software Engineering Conference pp. 33-42, 2010.

[9] E. Clarke, O. Grumberg, D. Peled, "Model Checking." The MIT Press. Cambridge, MA. 2008.

[10] 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. 2010, Vol. 6143, pp. 77-93.

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

[12] J.B. Cohen, D. Plakosh, K. Keeler, "Robustness Testing of Software-Intensive Systems: Explanation and Guide." CMU Software Engineering Institute. Technical Note. April 2005.

[13] 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

[14] D.L. Long, L.A. Clarke, "Task Interaction Graphs for Concurrency Analysis." International Conference on Software Engineering. June 2012, pp. 44-52.

[15] A. Lamsweerde, "Formal Specification: A Roadmap." ICSE '00: Proceedings of the Conference on the Future of Software Engineering. 2000, pp. 147-159.

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

[17] T.A. Wei-tek, R. Paul, C. Fan, L. Yu, "Automated Event Tree Analysis Based-on Scenario Specifications." Proceedings of IEEE ISSRE. 2003, pp. 47-74.

[18] L.M. Ridley, J.D. Andrews, "Application of the Cause-Consequence Diagram Method to Static Systems." Reliability Engineering and System Safety, Elsevier. 2002, Vol. 75, Issue 1, pp. 47-58

[19] D. Aceituna, H. Do, S. Lee, "Interactive Requirements Validation for Reactive Systems Through Virtual Requirements Prototype." MoDRE at RE11. 2011, pp. 1-10.

[20] D. Aceituna, H. Do, G. Walia, S. Lee, "Evaluating the Use of Model-Based Requirements Verification Method: A Feasibility Study." EmpiRE at RE11. 2011, pp. 13-20.

[21] D. Aceituna, H. Do, G. Walia, S. Lee, "Model-Based Requirements Verification Method: Conclusions from Two Controlled Experiments." Information and Software Technology. March 2014, Vol. 56, Issue 3, pp. 321-334.

[22] B. Boehm, V. Basili "Software Defect Reduction Top 10 List." IEEE Computer. 2001, Vol. 34, pp. 135-137.

[23] 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.

[24] N. Leveson, "The Role of Software in Recent Aerospace Accidents." Proceedings of the 19th International System Safety Conference. 2001.

[25] N. Leveson, G. Leveson, "Systemic Factors in Software-Related Spacecraft Accidents." AIAA Space 2001 Conference and Exposition. 2001.

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