A Combinatorial Approach for Exposing Off-Nominal Behaviors

Kaushik Madala
University of North Texas
Denton, Texas
kaushikmadala@my.unt.edu

Hyunsook Do
University of North Texas
Denton, Texas
hyunsook.do@unt.edu

Daniel Aceituna
DISTek Integration, Inc.
Daniel.Aceituna@distek.com

ABSTRACT

Off-nominal behaviors (ONBs) have been a major concern in the areas of embedded systems and safety-critical systems. To address ONB problems, some researchers have proposed model-based approaches that can expose ONBs by analyzing natural language requirements documents. While these approaches produced promising results, they require a lot of human effort and time. In this paper, to reduce human effort and time, we propose a combinatorial–based approach, Combinatorial Causal Component Model (Combi-CCM), which uses structured requirements patterns and combinations generated using the IPOG algorithm. We conducted an empirical study using several requirements documents to evaluate our approach, and our results indicate that the proposed approach can reduce human effort and time while maintaining the same ONB exposure ability obtained by the control techniques.

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; • **Software and its engineering → Requirements analysis**;

KEYWORDS

Off-Nominal Behaviors, Requirements Verification, Combinatorial Approach, Model-based Approach

ACM Reference Format:
Kaushik Madala, Hyunsook Do, and Daniel Aceituna. 2018. A Combinatorial Approach for Exposing Off-Nominal Behaviors. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.
https://doi.org/10.1145/3180155.3180204

1 INTRODUCTION

Off-nominal behaviors (ONBs) are unexpected or unintended behaviors of a system [1, 7, 12]. ONBs can occur for various reasons such as human errors caused by not following the intended procedures, components of a system being in conflicting states of operation, and environmental conditions resulting in such conflicting states. ONBs have been a major concern in the areas of embedded systems [1], medical devices, autonomous robotic systems [43], and safety-critical systems [2] because not addressing ONBs in these application domains might result in hazardous accidents and even catastrohpic results [29]. Unlike expected or nominal behaviors, ONBs requires knowledge acquisition of missing requirements and they are usually not specified in natural language (NL) requirements [12].

Some researchers have tried to address ONB problems [7, 16, 19, 44]. For example, Jensen et al. [19] proposed a model-driven design that uses the concept of safety functions to handle ONBs, and a study by Verma et al. [44] utilizes off-nominal behavior test cases for airplane runway operations to expose possible ONBs. Iverson [16] proposed a simulation–based approach that creates a knowledge base by grouping related behaviors in nominal data sets that contain the acceptable behaviors of the system. The created knowledge base is used to identify ONBs by collecting behaviors that diverge from the acceptable behaviors. Despite the progress this research has achieved, there are some limitations with the existing approaches. Most ONB problems have focused on how a human operator reacts to off-nominal situations within the operating environment rather than from the system's perspective. Also, ONB problems are often addressed only after the system has been implemented, whose solution may involve substantial redesigning, and additional cost, because it is applied late in the development cycle.

Some work has been done at requirements level to address these limitations. Causal Component Model (CCM) [1] and Enhanced Causal Component Model (ECCM) [33] were proposed to expose ONBs in NL requirements by converting the requirements into system state transition rules and parsing them to find user-identified undesired states that result in ONBs. However, CCM and ECCM have some drawbacks. Both approaches involve NL requirements to be converted into transition rules. Because common NL requirements often contain ambiguities [3], directly generating rules from requirements can be an error-prone process and tracing back from rules to NL requirements requires a lot of time and effort. Both models, CCM and ECCM, suffer from these problems.

To reduce human error as well as to reduce the number of states that need to be manually analyzed to expose ONBs, in this paper, we propose a Combinatorial-Causal Component Model (Combi-CCM), a component-driven model. Unlike CCM and ECCM, Combi-CCM does not require writing rules from common natural language requirements, but it requires users to write requirements by following a modified version of Easy Approach to Requirements Syntax (EARS) requirements patterns [34, 36]. The use of these patterns forces elicitation, reduces ambiguities [34], and makes writing rules less error prone. Once the requirements are written to be consistent with patterns, the rules are generated by a direct mapping. The rules generated are component state transition rules, which will be converted into system transition rules by rule expansion. While the expansion in Combi-CCM is similar to that of CCM and ECCM,

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180204

the number of states that users need to examine to identify ONBs is much smaller. Combi-CCM uses the combinatorial algorithm IPOG [28] to reduce the number of system states that a person needs to examine while exposing ONBs. The rules causing ONBs are analyzed, and corresponding requirements are corrected via reverse mapping from rules back to requirements in modified EARS patterns. To evaluate our proposed approach, we conducted an empirical study using 7 requirements specifications, comparing CCM, ECCM, and Combi-CCM with and without state flattening, using 2-way and 3-way combinations. Our results show that the 2-way combination of Combi-CCM performs well in identifying possible ONBs, with a smaller number of states to be analyzed manually than CCM, ECCM, and 3-way Combi-CCM.

The rest of the paper is organized as follows. Section 2 provides brief descriptions of EARS patterns, CCM, and ECCM. Section 3 presents the proposed approach, and Section 4 details the experimentation, threats to validity, and results. Section 5 analyzes the results and discusses limitations, and Section 6 discusses related work. Section 7 presents conclusions and discusses future research directions.

2 BACKGROUND

In this section, we briefly explain EARS requirements patterns, CCM, ECCM and their model elements.

2.1 EARS Requirements Patterns

Common natural language is widely used for writing requirements. However, common NL can be ambiguous, wordy, and hard to analyze and formalize. To address these issues, Mavin et al. [34, 36] proposed an easy approach to requirements syntax (EARS), namely simple sentence patterns for requirements. The sentence patterns include patterns for generic requirements, ubiquitous requirements (requirements that need to be true always), event-driven requirements, state-driven requirements, unwanted behaviors requirements such as failure handling requirements, optional feature requirements, and complex requirements that are logical combinations of earlier stated requirements. An example of an EARS requirements pattern is "WHILE <in a specific state> the <system name> shall <system response>", which is a state-driven pattern. It has been proven in various case studies [34–36] that EARS decreases vagueness, wordiness, and ambiguity, and increases testability and verifiability. EARS requirements patterns are more suitable for system-level requirements than requirements written for component-driven development. Motivated by the benefits of EARS patterns, we adapt state-driven and event-driven patterns into patterns that can be used for component-driven development, as a part of our proposed approach.

2.2 CCM and ECCM

Before we discuss CCM and ECCM approaches, we will explain the model elements of CCM and ECCM.

2.2.1 Model Elements. CCM and ECCM include three major model elements: component, component state, and transition condition. A *component* is a part of a system that has a well-defined functionality and can change states when transition conditions are met. In the case of embedded systems and robots, a component is a piece of hardware and its associated software. For example, in a robot system, the motor, actuator, and ultrasonic sensor are components.

Every component has states. A *component state* describes the current condition of a component. For example, a motor can have two states: 'off' and 'on'. A component state that has a parent state is considered to be a sub-state of the parent state. For example, a motor's 'on' state can have sub-states such as accelerate and decelerate. States without parent states (abstract level of states) are considered to be at level 0 and with an increased level of sub-states, the level value is increased. For example, the level of the motor's 'on' state is 0 while its 'accelerate' state is 1. A component state is represented as Component name(State name). For example, a motor's 'on' state is represented as Motor(on). If a component state is a sub-state, we represent it as the component name followed by a concatenation of ancestral states of the component state using a period (.) as a separator, appended by the component state at the end. For example, the 'accelerate' state of the motor is represented as Motor(on.accelerate).

A *system state* is a combination of concurrent component states, as the behavior of various components together constitute the behavior of a system. For example, for the robot system describe above, a system state would be written as (Motor(on), Actuator(active), Ultrasonic Sensor(no detection)). A *transition condition* is a component state that allows (or enables) the transition of another component state or a trigger that results in a state transition of component states. A transition condition can be environmental, i.e., due to human factors or environmental conditions such as pressure and temperature; or it can be system related such as the transition of a component state due to other component state(s) because of dependency. An example of an environmental transition condition is the temperature being greater than 100°F, which results in the transition of the temperature sensor from a safe to an overheated state. An example of a system transition condition is an ultrasonic sensor detecting an object, which results in the transition of the motor from 'on' to 'off'. Using the model elements, rules are written in the pattern *Transition Condition: Component(Current State) -> Component(Next State).*

2.2.2 CCM. CCM [1] is a modeling technique used to expose ONBs. The major steps in CCM involve the conversion of common NL requirements into component state transition rules. These rules are then converted into numerical form to facilitate computation. These numerical rules are in turn used in rule expansion to convert component state transition rules to system state transition rules. Once the system state transition rules are generated, rules with user-identified undesired states are separated out. The process of identifying rules with undesired states is a semi-automatic process. The user must examine all possible system states and identify the undesired states that result in ONBs; these states are then mapped to rules automatically. Once the rules are separated, any rule with both the current and next state identified as undesired is omitted. For rest of the rules, state profiles for undesired states in the rules are determined using the state profiling algorithm [1] to check whether the state is recoverable without human intervention. Any rule that has an undesired next state that is not recoverable is considered an unrecoverable rule. Rules that are not recoverable are discussed with stakeholders and subject experts, and corresponding requirements are corrected.

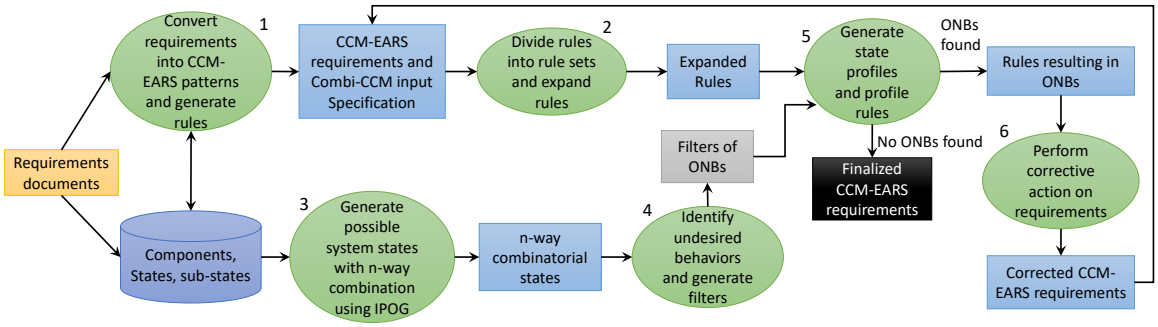

Figure 1: Overview of the Proposed Approach

2.2.3 ECCM. ECCM is an enhanced version of CCM. If sub-states are present, ECCM unlike CCM, does not require state flattening [8]. The state flattening [8] process involves removing hierarchical relationships among states and forcing them to be represented at the same level. This approach instead divides rules into rule sets using the rule set generation algorithm (see Algorithm 1). This helps alleviate the rule explosion problem and state explosion problem [42]. In addition, ECCM requires users to identify undesired states only from a list of nonrecoverable states. The rest of the approach is similar to CCM. CCM cannot accommodate sub-states without performing state flattening, but ECCM can handle the concept of sub-states. Despite the advantages of these approaches, as mentioned in Section 1, both CCM and ECCM requires a lot of human effort and time. To handle these limitations, we propose our approach, Combi-CCM.

3 APPROACH

An overview of the proposed approach is shown in Figure 1. The ovals denote the processes/steps, the rectangles denote inputs/outputs, and the cylinder represents database. The input to the approach is the requirements documents, and the final output is the finalized requirements in modified EARS patterns. The numbers next to the ovals denote step numbers.

Before we describe each step in detail, we provide a brief overview of Combi-CCM. First, NL requirements are converted into requirements written in modified EARS requirements patterns, or CCM-EARS patterns. During this step, components, states, and sub-states are identified and stored in the database. The converted requirements are used to generate component state transition rules (Step 1). The generated rules are divided into rule sets if sub-states are present . The rules in each rule set are expanded to generate system state transition rules (Step 2). While the rules are being expanded, system states are generated using IPOG [28], a n-way combinatorial algorithm (Step 3). The generated states are manually analyzed to identify ONBs, and a filter for each ONB is created (Step 4). Filters are numerical patterns that can identify undesired states and they are further explained later in this section. The filters, along with the state profiles generated using the state profiling algorithm [1], are used to identify rules with ONBs that require human intervention to return to a desirable state (Step 5). The corresponding requirements of these identified rules are found and corrected (Step 6). In this paper, we use the terms 'ONBs' and 'undesired behaviors' interchangeably.

Step 1: Convert requirements to CCM-EARS patterns and generate rules

Most requirements are written in common natural language [32]. Common NL requirements are often wordy, ambiguous [3], and incomplete. EARS patterns [34, 36] were proposed to address these issues. EARS patterns refer to six restricted NL sentence patterns that were created to reduce the problems associated with common NL requirements. Case studies [34–36] that used EARS indicate that EARS patterns are less ambiguous, less vague, less complex, and more verifiable than common NL requirements. Further, EARS patterns facilitate the requirements elicitation process by forcing engineers to fill in missing information. To use EARS patterns in our approach, we modified them because EARS patterns are not suitable for component-driven requirements and we renamed them CCM-EARS. In this paper, we modified only two EARS patterns because they can capture the essential elements in building a component-driven state transition model. The rest of patterns are out of scope of this paper as they are not used in the basic model that we build in our approach. The two EARs patterns we modified are as follows:

Event-Driven: WHEN < *trigger* > < *optionalprecondition* >, the < *systemname* > shall < *systemresponse* >.
State-Driven: WHILE < *systemstate* >, the < *systemname* > shall < *systemresponse* >.

We converted them into the following CCM-EARS patterns:

Event-Driven (For transitions resulting from triggers invoked by environmental reasons): WHEN < *environmentaltrigger* >, the < *componentname* > shall transition from < *componentstate*1 > to < *componentstate*2 >.
State-Driven (For transitions resulting from dependency between components): IF < *component*1*name* > is in < *component*1*state* > state, the < *component*2*name* > shall transition from < *component*2*state*1 > to < *component*2*state*2 >.

In addition to these two patterns, we can also convert requirements into a complex pattern similar to complex requirements pattern of EARS, which contains a logical combination of the two patterns including the logical combinations of patterns. For example, a sample complex pattern is "WHEN < *environmentaltrigger* > AND < *component*1*name* > is in < *component*1*state* > state, the < *component*2*name* > shall transition from < *component*2*state*1 > to < *component*2*state*2 >". As a first step of exposing ONBs, we consider common NL requirements, and find components, their

states, and sub-states, and transition conditions using which we create requirements that follow event-driven, state-driven, or complex CCM-EARS patterns. CCM-EARS patterns force the user to elicit requirements if details are missing in the pattern to be followed. In addition, it is easy to enforce constraints on CCM-EARS patterns than that on CCM rules. This helps identify the unknown components, states, and sub-states. The information about components, states, and sub-states is stored in the database and used later in the generation of filters to expose ONBs.

An example of the conversion of requirements into CCM-EARS patterns is illustrated as follows:

Requirements Sample 1:

(1) *The robot system shall consist of a push-button switch, a motor, an ultrasonic sensor, and a temperature sensor.*
(2) *The system can be turned off and on using the switch, which also activates the motor.*
(3) *The system must stop if the ultrasonic sensor detects an object.*
(4) *The system must move only when the robot is on and the temperature is safe, i.e., less than 100° F.*

In this example, we have assumed the switch to have 'off' and 'on' states, unlike a stateless switch. Considering this condition, the requirements in Requirements Sample 1 can be converted into the following CCM-EARS requirements:

CCM-EARS requirements for Requirements Sample 1:

(1) *The robot system shall consist of a push-button switch, a motor, a ultrasonic sensor and a temperature sensor.*
(2) *When the user presses the switch, the switch shall transition from off to on.*
(3) *When the user releases the switch, the switch shall transition from on to off.*
(4) *If the switch is in the off state, the motor shall transition from on to off.*
(5) *If the switch is in the on state, the motor shall transition from off to on.*
(6) *When an object is present, the ultrasonic sensor shall transition from no detection to detection.*
(7) *If ultrasonic sensor is in detection state, the motor shall transition from on to off.*
(8) *When the temperature is less than 100° F, the temperature sensor shall transition from overheated to safe.*
(9) *If the temperature sensor is in the safe state and the switch is in the on state, the motor shall transition from off to on.*

Once the CCM-EARS requirements are generated, they are converted into rules. As mentioned in Section 2, a transition rule is of the form *Transition Condition: Component(Current State) -> Component(Next State).*

The state-driven pattern "IF <component1 name> is in <component1 state> state, the <component2 name> shall transition from <component2 state 1> to <component2 state 2>" will be converted into transition rule of form *component1 name (component1 state): component2 name (component2 state 1) -> component2 name (component2 state 2).*

The event-driven pattern "WHEN <environmental trigger> the <component name> shall transition from <component state 1> to <component state 2>" will be converted into transition rule of form *environmental trigger: component name (component state 1) -> component name (component state 2).*

In the case of the complex pattern, the *environmental trigger/ component1 name(component1 state)* will be followed by the conjunction 'AND' followed by other *environmental trigger/component3 name(component3 state)* as transition condition until we specify all conditions in the combination; the generation of component(current state) to component(next state) is similar to the aforementioned transformations.

When we convert the CCM-EARS requirements (Requirements Sample 1) into transition rules, we obtain the following rules (due to space limitations, we illustrate a small portion of rules):

Example Rules from CCM-EARS requirements for Requirements Sample 1:

(1) User(presses_switch): Switch(off) -> Switch(on) (rule 1)
(2) Temperature_Sensor(safe) AND Switch(on): Motor(off) -> Motor(on) (Example of Complex pattern) (rule 8)

Algorithm 1: Generation of Rule Sets

input : A set of components in system $C = \{C_1, C_2, C_3, \ldots, C_n\}$
A set of states at level 0 for each component
maxlevel, maximum level of sub-states
Sets of sub-states for each component at level 1, 2, 3, ..., maxlevel
Rules R = $\{R_1, R_2, R_3, R_4, \ldots, R_w\}$
output : Rulesets

```
1  rulesetnum = 0;
2  for currentlevel ← 0 to maxlevel do
3      if currentlevel == 0 then
4          foreach Rule R_i in R do
5              if states in R_i ∈ any of component-states at level 0 then
6                  Rulesets[rulesetnum] ← Rulesets[rulesetnum] ∪ R_i;
7          rulesetnum ← rulesetnum+1;
8      else
9          foreach Component C_i in C do
10             if Component C_i has sub-states at level = currentlevel
                   then
11                 foreach Rule R_i in R do
12                     if component in R_i == C_i then
13                         if state in R_i ∈ sub-states of C_i at level =
                               currentlevel then
14                             truncate states with lower level
                                   sub-states;
15                             Rulesets[rulesetnum] ←
                                   Rulesets[rulesetnum] ∪ R_i;
16                     else
17                         if states in R_i ∈ any of states at level 0 then
18                             Rulesets[rulesetnum] ←
                                   Rulesets[rulesetnum] ∪ R_i;
19                         rulesetnum ← rulesetnum+1;
```

Step 2: Divide and expand the rules

Once the rules are written, if the system's components have sub-states, the rules can be divided into rule sets. The engineers, domain experts, and stakeholders are at liberty to decide whether to consider sub-states separately or to flatten the state space. State flattening [8] refers to the process of removing hierarchical states and representing all the states at the same level. For example, suppose that a component *Motor* has two states, 'off' and 'on', and that the 'on' state has two sub-states, 'idle' and 'move.' By applying state

flattening [8], we represent the states of *Motor* as 'off,' 'idle,' and 'move.' Algorithm 1 [33] generates rule sets. R_i in the algorithm represents rule number i in the list of transition rules (e.g., *Transition Condition: Component(Current State) -> Component(Next State).* C_i represents i^{th} component in the system. The formal definitions of model elements, and more details on rule set generation algorithm can be found in [1] and [33], respectively.

The algorithm generates only one rule set if state flattening is done for hierarchical states. The algorithm assigns levels to states based on the hierarchy. For example, states at level 0 are states with no parent states, whereas states at level 1 have parent states that are at level 0. The algorithm generates rule sets until all levels of sub-states are covered (line 2). If level 0 states are being explored (line 3), any rule with level 0 states in the input set (line 4) is added to the first rule set (lines 5 and 6). If the level of states is greater than 0, i.e., if there are sub-states (line 8), the following series of operations are performed for each component (line 9). For each level of sub-states of the component, if the component contains sub-states at that level (line 10), a new rule set is created. Any rule with sub-states in the component at that level (lines 11–13) is added to the new rule set (line 15). However, if the one of the states in the rule is at the current level while other state has the next level of information, we truncate the other state to retain only the current level of information (line 14). For the rules whose components are not being investigated, only the rules that have level 0 states (states with no parents) are added to the rule set (lines 16–18).

We illustrate the algorithm using the following example.

Requirements Sample 2:

(1) *User(presses_switch): Switch(off) -> Switch(on)*
(2) *User(releases_switch): Switch(on) -> Switch(off)*
(3) *Switch(on): Motor(off) -> Motor(on.idle)*
(4) *Switch(off): Motor(on) -> Motor(off)*
(5) *User(enters_location): Motor(on.idle) -> Motor(on.move)*
(6) *Destination(reached): Motor(on.move) -> Motor(on.idle)*

As mentioned in Section 2, a sub-state is represented by concatenating its parents' states followed by period (.), and then appending sub-state itself. For example, in the Requirements Sample 2 rules, *motor* has two outermost states: off and on. But in 'on' state, the motor can be in 'idle' or 'move' state. These states are represented in the rules as 'on.idle' and 'on.move.'

By applying Algorithm 1, on the Requirements Sample 2 rules, the algorithm first considers rules with at least one state with no parent state and forms the first rule set as follows:

Rule set 1 - for Requirements Sample 2:

(1) *User(presses_switch): Switch(off) -> Switch(on)*
(2) *User(releases_switch): Switch(on) -> Switch(off)*
(3) *Switch(on): Motor(off) -> Motor(on)*
(4) *Switch(off): Motor(on) -> Motor(off)*

From this example, we can observe that the algorithm truncates any state that has sub-state information when analyzing the parent states; i.e., in the rule *Switch(on): Motor(off) -> Motor(on.idle)*, 'on.idle' is truncated to 'on' as 'idle' represents a sub-state.

Next, the algorithm starts checking sub-states for each component, starting with *Switch*. *Switch* does not have any rules with sub-states, and thus *Motor* is examined. Because *Motor* has sub-states, a second rule set is formed as follows:

Rule set 2 - for Requirements Sample 2:

(1) *User(presses_switch): Switch(off) -> Switch(on)*
(2) *User(releases_switch): Switch(on) -> Switch(off)*
(3) *Switch(on): Motor(off) -> Motor(on.idle)*
(4) *User(enters_location): Motor(on.idle) -> Motor(on.move)*
(5) *Destination(reached): Motor(on.move) -> Motor(on.idle)*

The advantage of partitioning into rule sets is that it reduces the rule explosion problem that can occur when transforming component states to system states. Flattening states increases the possible number of combinations of component states, thereby increasing the number of system transition rules. Partitioning into rule sets allows to process a small number of component states at any given time by restricting the state space, thereby reducing the number of system transition rules that will be generated. The rules we have generated for the example requirements in Step 1 are grouped in only one rule set because the components have no sub-states. Once the rule sets are generated, the rules in a rule set are converted into numerical rules, in which component states are represented in numerical form to make computation easier during rule expansion.

	State 1	State 2	State 3
Component 1	1, 0, 0	2, 0, 0	3, 0, 0
Component 2	0, 1, 0	0, 2, 0	0, 3, 0
Component 3	0, 0, 1	0, 0, 2	0, 0, 3

Substates of 2, 0, 0

Substate 1	Substate 2	Substate 3	Substate 4
2.1, 0, 0	2.2, 0, 0	2.3, 0, 0	2.4, 0, 0

Substates of 2.3, 0, 0

Substate 1	Substate 2	Substate 3	Substate 4
2.3.1, 0, 0	2.3.2, 0, 0	2.3.3, 0, 0	2.3.4, 0, 0

Figure 2: Numerical States Representation

The representation of states and sub-states in numerical form is illustrated in Figure 2. The rows show components and the columns show states. Sub-states are listed in the columns within states. The numerical states are generated by considering the ordinal position of components and states. In the case of sub-states, the ordinal position is concatenated with the parent state with period (.) as a separator. For example, (1, 0, 0), refers to state 1 of component 1, which is in a system that consists of 3 components and (1.2, 0, 0) represents sub-state 2 of state 1 of component 1. The value 0 indicates that the component can be in any of its states. When a specific value is assigned instead of 0, the result is a system state as we have information about all the components' states; i.e., a system state is a combination of components' states.

The first and eighth rules in Requirements Sample 1 are converted into the following numerical rules:

Numerical rules of Requirements Sample 1:

(1) *User(presses_switch): 1, 0, 0, 0 -> 2, 0, 0, 0*
(2) *0, 0, 0, 1 AND 2, 0, 0, 0: 0, 1, 0, 0 -> 0, 2, 0, 0*

Rule expansion is performed once the rules are converted into numerical rules. The main goal of rule expansion is to convert component state transitions into system state transitions. We followed

the same rule expansion process used in CCM [1] and ECCM [33]. When the transition rule has a component's state(s) as a transition condition, it implies that the component in the transition condition must be in the state specified for the transition to happen. It further implies that the current state as well as the next state of the system must have that component in the specified state. To achieve this, we perform absorption and propagation operations. Absorption involves the transfer of state information from the transition condition to the current component state. When we apply absorption to rule 8 in Requirements Sample 1, we obtain *0, 0, 0, 1 AND 2, 0, 0, 0: 2, 1, 0, 1 –> 0, 2, 0, 0*. The propagation operation involves moving the state information from the current state to the next state. After propagation, rule 8 becomes *0, 0, 0, 1 AND 2, 0, 0, 0: 2, 1, 0, 1 –> 2, 2, 0, 1*. Once absorption and propagation are done, the transition conditions that have only the component's states will be converted into a notation of 'T' ('T' refers to a transition caused by system) concatenated with the rule number from the list of rules with the component's state(s) as their transition condition. For example, rule 8 will be changed into *T4: 2, 1, 0, 1 –> 2, 2, 0, 1*. Next, we perform expansion, which involves replacing zeroes in the numerical states with all possible component state values that make up the system's state space. If a rule set has a component's sub-states, we replace the value of zeroes with only the values of sub-states of that component. For example, rule 8, after expansion, generates the following rules:

Expansion of rule 8 in Requirements Sample 1:

(1) *T4: 2, 1, 1, 1 -> 2, 2, 1, 1*
(2) *T4: 2, 1, 2, 1 -> 2, 2, 2, 1*

Step 3: Generate possible system states with n-way combination of component states

As mentioned early, traditional CCM and ECCM approaches require users to examine a large number of states manually to find undesired states. Because the number of possible system states is a Cartesian product of component states, if the target system is large and complex, the number of states that are to be checked manually can be very large. For example, if a system has 20 components, 10 states each, then the Cartesian product of the states would be 10^{20}. Examining all these states manually would take substantial time and human effort. Our goal is to reduce the number of states requiring manual analysis thereby reducing human effort. To reduce human effort, we use the IPOG algorithm to reduce the number of states that need to be manually analyzed. IPOG algorithm [28] is widely used in the combinatorial software testing community. Because ONBs occur due to conflicting component states, checking an n-way combination of component states must be able to cover all types of behaviors between any *n* components. For example, when n=2, 2-way combination ensures that all possible combinations of component states between any two components are covered. In general, 2 and 3 are widely used for *n* value [24, 28]. For example, for Requirement Sample 1, when we applied the IPOG algorithm using 2-way combinations, only 6 states needed to be analyzed instead of 16 states. Further, we can add constraints when we generate combinations so that only sensible combinations are considered. For example, if we consider a server and server communication as two components of a system, server communication is possible only when the server is on, checking whether the server is communicating when it is off is not necessary. Thus, unnecessary behaviors can be omitted by using, in this case, the constraint, "server communication can be in communicating state only if the server is in on state." The value of 'n' must be chosen based on the dependencies between the components. If a component interacts with all other components in the system, then the 'n' value must be the number of components. However, in reality, most systems do not function such that each component is dependent on every other component. Most systems have a component that acts as a controller, and the rest of the components interact with it. As a result, the 'n' values of 2 and 3 are sufficient to represent dependencies in most systems. Our approach is meant for such systems. We plan to work on reducing the number of states in which multiple components must cooperate (e.g., a multiple robot coordination system for moving heavy objects) as part of future work.

Step 4: Identify undesired behaviors and generate filters

Once the n-way combinations of states are generated, the states are analyzed to determine whether any system components of the system are in any conflicting states. It is possible for a single system state to exhibit multiple undesired behaviors, especially when analyzing a complex system with large number of components. To identify all the states in a system with undesired behaviors, we create filters of these states. Filters are numerical patterns of undesired states and are represented similar to system states. Any state that matches the filter is an undesired state and the filters are used for exposing ONBs. If the state value of a particular component in the filter is 0, then the state of that component is ignored when the comparison is made. For example, in Requirements Sample 1, the ONB identified is *motor* is 'on', when *temperature sensor* is 'overheated'. The corresponding filter representation is 2, 0, 2. The states that match this filter in Requirements Sample 1 are: 2, 1, 2 and 2, 2, 2. These states are considered undesired states.

Step 5: Generate state profiles and profile rules

After rule expansion, all the component state transition rules are converted into system state transition rules. Because our goal is to not only expose ONBs but also find the cause of ONBs and check if they are recoverable, we generate state profiles for each system state. State profiles represent the in-degree(ID) and out-degree(OD) of a state that result from environmental transition conditions (E) as well as a component's state(s) as transition conditions (T). State profiling requires finding TOD, TID, EOD, and EID, for each state. Figure 3 illustrates state profiles for a given system state.

TOD is the number of component's state transition conditions resulting in transition from the state being profiled. TID is the number of component's state transition conditions resulting in transition to the state being profiled. EOD is the number of environmental transition conditions resulting in transition from the state being profiled. EID is the number of environmental transition conditions resulting in transition to the state being profiled. These values are calculated for each state by parsing through each rule and checking the current state and the next state of the rule. If the transition condition is environmental, then EOD of the current state is incremented by 1 and EID of the next state is incremented

by 1. If the transition condition is system originated (coming from component's states), then TOD of the current state is incremented by 1 and TID of the next state is incremented by 1.

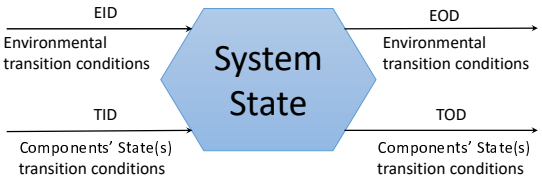

Figure 3: Representation of state profile

For example, let us consider the following expanded rules from Requirements Sample 1:

(1) *User(Releases): 1, 1, 1, 1 -> 2, 1, 1, 1*
(2) *T4: 2, 1, 1, 1 -> 2, 2, 1, 1*

When the first rule stated above is parsed, because the transition condition has an environmental cause, the EOD value of the system state 1, 1, 1, 1 is incremented by 1 and the EID value of 2, 1, 1, 1 is incremented by 1. In the second rule, because the transition condition is system originated, the TOD value of 2, 1, 1, 1 is incremented by 1 and the TID of 2, 2, 1, 1 is incremented by 1.

The state profiles generated along with the filters of undesired behaviors that are generated by manual identification of undesired behaviors (explained in Step 4) will be used to profile rules. Filters, as stated earlier, are patterns that represent undesired states. These filters are used to find rules with undesired states and thereby expose ONBs. This is done by checking whether the current system state or next system state in the rule matches with the pattern of the filter. Any rule that has undesired current and next states will be removed, and the corresponding requirement is corrected or removed. Any rule with a next state as the undesired state will be analyzed further. If a rule contains only an undesired next state, then the state profile of the undesired next state is examined to determine whether its TOD is 0 or greater than 0. If TOD is 0, this indicates that the undesired state cannot be recovered automatically and human intervention is required because it has no component's state (a system originated cause), that can transition into a desirable system state. We identify such rules as nonrecoverable rules.

We illustrate this concept using Requirement Sample 1. In this example, one of the undesired behaviors is that *motor* is 'on', when *temperature sensor* is 'overheated'. To identify states that are related to undesired behavior, we use filter 0, 2, 0, 2. When we applied this filter, we found a rule that has the undesired next state, which is: $T2$: 2, 1, 1, 2 –> 2, 2, 1, 2. The TOD value of undesired state 2, 2, 1, 2 is zero, which implies that the rule leads to the nonrecoverable state.

Step 6: Perform corrective action

Once the nonrecoverable rules are identified, stakeholders, requirements engineers, and domain experts discuss and analyze the causes of those rules and correct the CCM-EARS requirements. In Requirements Sample 1, the nonrecoverable rule found in Step 5 was caused by requirement 5 because having "Switch(on)" without checking temperature sensor is letting motor to transition from 'off' to 'on'. To correct requirement 5, we omitted it as the expected behavior is covered by requirement 9. Once the requirements are corrected, the process is repeated until no undesired behaviors are found.

4 EMPIRICALSTUDY

To evaluate our proposed approach, we conducted an empirical study to investigate the following research questions:

- RQ1: Does Combi-CCM reduce the number of system states that should be analyzed manually compared to CCM and ECCM?
- RQ2: Does a reduction in the number of states affect the ONB exposure ability?

4.1 Objects of Analysis

We used 7 requirements specification documents to investigate our research questions as follows:

(1) **Automatic Delivery System (ADS)**: Automatic Delivery System [30] is a 175–page requirements document. The requirements specifications concern a robot asked with delivering packages. In this study, we considered only the robot's behavioral requirements.
(2) **AB Mail Robot (ABMR)**: AB Mail Robot [38] is a 13–page requirements document written for a course project. The requirements specifications concern a robot asked with delivering mail from mailboxes to people. We have considered behavioral aspects in this study.
(3) **Autonomous Robotic Vacuum Cleaner (ARVC)**: The requirements specifications of Autonomous Robotic Vacuum Cleaner [39] is a 9–page document written for a course project that lists requirements for a robot vacuum cleaner that must be able to clean a room.
(4) **Oddbotics**: Oddbotics requirements [11] are a 46–page document. The requirements specifications are for a robot that must to capture video while moving around in an environment.
(5) **Digital Home Software (DHS)**: The requirements specifications for Digital Home Software [17] is a 15–page document that details a complex system that helps automatically control temperature, humidity, as well as small appliances using a mobile or other personal device. We considered only 20 of 50 contact sensors to find and analyze ONBs.
(6) **Pacemaker**: Pacemaker specification [4, 37] is a 35–page document that describes the functionality of a pacemaker. This document is widely used by other researchers [15, 18, 20, 41]. In this study, we concentrate on the pulse generator component of pacemakers whose malfunction results in complications to the patient.
(7) **Excavator**: Excavator requirements [1] were used for a CCM case study. They consist of 8 sample requirements about an excavator, a real-world system, with details about the operator safety features in an excavator.

4.2 Variables and Measures

Our study manipulated one independent variable, the modeling technique that exposes ONBs. We considered two control techniques (CCM and ECCM) and four heuristic techniques (Combi-CCM F2, Combi-CCM H2, Combi-CCM F3, Combi-CCM H3). A description of these techniques can be found in Table 1.

Table 1: Independent Variables (Modeling Techniques)

Type	Technique	Description
Control	CCM	CCM requires users to analyze all possible system states to identify ONBs and determine the causes of these ONBs.
	ECCM	The enhanced version of CCM. ECCM generates rule sets given hierarchical states. ECCM requires users to manually analyze only nonrecoverable states.
Heuristic	Combi-CCM F2	This technique performs state flattening when there are hierarchical states and generates the states to be analyzed using 2-way combinations.
	Combi-CCM H2	This technique generates rule sets when there are hierarchical states and generates the states to be analyzed using 2-way combinations.
	Combi-CCM F3	This technique performs state flattening when there are hierarchical states and generates the states to be analyzed using 3-way combinations.
	Combi-CCM H3	This technique generates rule sets when there are hierarchical states and generates the states to be analyzed using 3-way combinations.

The dependent variable for RQ1 is the number of states that needs manual analysis, and for RQ2, the dependent variable is the number of ONBs exposed by the techniques.

4.3 Experimental Setup and Procedure

To perform the experiment, we implemented the Combi-CCM tool in Java. We used ACTS [47] to generate system states that a person needs to analyze manually. The ECCM and CCM tools were implemented in Java and C#, respectively.

The experimental procedure followed the approach explained in Section 3. The common NL requirements of all the requirements documents suitable for creating model are converted into CCM-EARS requirements[1]. These CCM-EARS requirements are used to generate rules by directly mapping sentence patterns to rule patterns. The rules are then divided into rule sets and rule expansion is performed. The user can decide whether to perform state flattening or not. In our experiment, we investigated research questions by considering techniques with and without state flattening. After rule expansion, using ACTS, we generated the system states that users would need to examine to find ONBs. In our experiment, we used both 2-way and 3-way combinations to generate system states. Based on the ONBs found by the users, filters are created. These filters are used to expose rules with ONBs. The rules with ONBs that are not automatically recoverable are reverse–mapped to CCM-EARS requirements, and corresponding requirements are corrected.

While CCM and ECCM require translation of NL requirements into rules, it is an error-prone process and tends to create incorrect rules. In order to have fair comparisons between control and heuristic techniques in terms of human effort reduction, we used the same rules generated from Combi-CCM for CCM and ECCM. CCM performs rule expansion using the rules. CCM does not divide rules into rule sets as it takes only flattened states as an input. The users manually analyze all possible system states and identify the undesired states. These undesired states are used to expose ONBs. Similar to Combi-CCM, CCM uses state profiles to check for the automatic recoverability of a system; rules with ONBs that cannot be addressed without human intervention are separated out, and corresponding common natural language requirements are addressed. The ECCM approach is similar to CCM, but in ECCM, the rules are divided into rule sets when hierarchical states are present, rather than flattening states. ECCM also requires users to go through only nonrecoverable states to identify undesired states.

[1]Sample CCM-EARS requirements: https://goo.gl/VKPm4U

4.4 Threats to Validity

In this study, we assumed that the generated CCM-EARs requirements are complete. This assumption might affect our ability to determine ONBs. We can address this threat by checking the requirements for completeness before the model is built. Further, the identification of ONBs involves human judgment, which can be subjective. We plan to mitigate this threat by using multiple domain experts during decision making.

4.5 Results

In this section, we present the results of our study considering each research question.

4.5.1 Reduction of manually analyzed system states (RQ1). Our first research question (RQ1) asks whether the Combi-CCM approach can reduce the number of system states compared to CCM and ECCM. To answer this research question, we compared the two control and four heuristic techniques shown in Table 1. The number of states that require manual analysis for each technique and requirements document is shown in Table 2.

From Table 2, we can observe that CCM produced the largest number of states that require manual analysis. For example, CCM produced 46,656,000 states for ADS and approximately 10^{22} states for DHS, which are prohibitively large numbers of states for manual analysis. ECCM requires users to examine only nonrecoverable states, which reduces the number of states for manual analysis. Heuristic techniques produced much smaller numbers of states than the control techniques. Combi-CCM F2 produced the smallest number of states, ranging from 6 to 108, among heuristics. The number of states produced by Combi-CCM H2 ranges from 6 to 1,439. This technique produced a similar number of states compared to Combi-CCM F2 for Oddbotics and Excavator, because these requirements do not have sub-states, but it produced greater number of states than Combi-CCM F2 for the rest of the requirements. Combi-CCM F3 produced a number of states ranging from 12 to 206. Combi-CCM F3 produced a smaller number of states for ABMR, ARVC, and DHS than did Combi-CCM H2.

4.5.2 ONB exposure ability (RQ2). From the results in RQ1, we learned that Combi-CCM was able to substantially reduce the number of states that require manual analysis compared to control techniques. However, if Combi-CCM is not effective in detecting ONBs, state reductions would not be beneficial. Thus, in RQ2, we investigated whether a reduction in the number of states can affect

Table 2: Number of System States Being Analyzed Manually

Req.Doc.	CCM	ECCM	Combi-CCM F2	Combi-CCM H2	Combi-CCM F3	Combi-CCM H3
ADS	46656000	3670016	**26**	110	141	410
ABMR	1728	34	**12**	40	34	114
ARVC	11664	615	**15**	72	66	205
Oddbotics	20736	10488	**14**	**14**	43	43
DHS	$\approx 10^{22}$	$\approx 10^{14}$	**38**	1439	198	1622
Pacemaker	448	396	**108**	128	206	252
Excavator	32	12	**6**	**6**	12	12

ONB exposure ability. To answer this question, we examined the number of states created as part of addressing RQ1, and identified the ONBs exposed by each technique. The number of ONBs found using each technique is shown in Table 3.

CCM requires users to examine all possible system states; thus, CCM expose all possible ONBs based on the knowledge of users. Because it is not possible to know all the ONBs in the requirements (which would be analogous to exposing all the faults in the program), in this experiment, we use the number of ONBs found by CCM as a baseline. Table 3 shows that CCM exposed 4 ONBs in ABMR, 3 in ARVC, 4 in Oddbotics, 3 in Pacemaker, and 3 in Excavator. The ONB numbers for ADS and DHS were estimated by considering ONB values from 2-way and 3-way heuristic techniques because they have too many states to manually examine. The actual number of ONBs that we could find by analyzing all those states might be larger than what we estimated.

ECCM did not detect as many ONBs as CCM. ECCM found 24 ONBs in ADS, 2 in ABMR, none in ARVC and Oddbotics, 40 in DHS. The ONB numbers found by ECCM were smaller than those from CCM, and for some cases, it failed to expose any ONBs. We conjecture that the reason for this result is that ECCM allows users to check only nonrecoverable states. The technique does not consider undesired states that transition into other undesired states.

When examining the results from heuristic techniques, all four techniques produced results similar to those of CCM, with one exception, in ABMR. In ABMR, the 2-way techniques, Combi-CCM F2 and Combi-CCM H2, produced a smaller number of ONBs than CCM. In ABMR, one of the ONBs is caused by concurrent conflicting states of three components; as a result, 2-way techniques did not cover the combination. For all other requirements documents, heuristics found the same number of ONBs as CCM.

From Tables 2 and 3, we can observe that heuristic techniques with state flattening such as Combi-CCM F2 and Combi-CCM F3 found the same number of ONBs with a smaller number of states compared to the techniques that do not use state flattening (Combi-CCM H2 and Combi-CCM H3). To investigate whether it is always preferable to use techniques with state flattening, we measured the number of expanded rules for each technique as shown in Table 4. The rules in Table 4 are automatically parsed and analyzed based on undesired states found.

In Table 4, the number of rules that need to be analyzed using a heuristic techniques with state flattening is very large when compared to the heuristic technique that do not use state flattening. For example, as shown in Table 4, the number of expanded rules estimated to be analyzed using techniques with state flattening is approximately 10^{25}, whereas techniques without state flattening require only 10^{17}. If the system has no sub-states, techniques with and without state flattening must analyze the same number of rules, as in Oddbotics and Excavator, with all the techniques requiring to parse through 168,541 and 116 rules, respectively. These results indicate that the benefits of state flattening depend on the number of components, their states, and their sub-states. For example, in the case of DHS, the total number of expanded rules generated by replacing every unknown component state with all possible component states is approximately 10^{25}, which can result in rule explosion and scalability problems. For such a case, it would be wise not to use state flattening.

Table 3: Number of ONBs Found

Req.Doc.	CCM	ECCM	Combi-CCM F2	Combi-CCM H2	Combi-CCM F3	Combi-CCM H3
ADS	25	24	25	25	25	25
ABMR	4	2	3	3	**4**	**4**
ARVC	3	0	3	3	3	3
Oddbotics	4	0	4	4	4	4
DHS	104	40	104	104	104	104
Pacemaker	3	3	3	3	3	3
Excavator	3	2	3	3	3	3

5 DISCUSSION AND LIMITATIONS

We proposed our approach to reduce human efforts and time needed to identify ONBs and correct requirements to make a system foolproof. Our approach used CCM-EARS requirements patterns to convert common NL requirements into simple, easy–to–understand, and verifiable requirements. Based on our example in Section 3 and requirements we created as part of the study, we conclude that CCM-EARS patterns help users with elicitation if requirements have missing model elements or details. We also found that tracing back from rules to CCM-EARS requirements is easier than tracing back from rules to common natural language requirements. This is because rules are created by mapping CCM-EARS patterns to rule patterns, and tracing back to CCM-EARS requirements is done by mapping rule patterns to requirements patterns. Further, we also found that CCM-EARS requirements can be reused for analysis more easily compared to state transition rules or NL requirements.

We can draw the following conclusions from the results obtained in our empirical study. First, overall, the results indicate that the proposed approach can reduce human efforts and time by reducing the number of states that need to be analyzed manually. Further, the results also demonstrate that a system with *N* components does not necessarily need N-way combinations to expose all ONBs because, for most systems, not every component in the system interacts with other *N-1* components. Second, our results imply that state flattening decreases the number of states that users need to analyze but increases the number of expanded rules that require machine power rather than human power. In the case of approaches without state flattening, the number of expanded rules decreases, but the number of states that require manual analysis increases. Therefore, there is a trade-off in selecting one technique over the other. Because the rules can be analyzed automatically, we can use state flattening approaches as long as they do not result in rule explosion problems due to flattening hierarchical states.

While Combi-CCM was able to reduce the number of states that require manual analysis, the approach can face a scalability problem because given a large number of components and states, Combi-CCM can generate a very large number of expanded rules.

Table 4: Total Number of System Transition Rules to be Analyzed Automatically

Req.Doc.	CCM	ECCM	Combi-CCM F2	Combi-CCM H2	Combi-CCM F3	Combi-CCM H3
ADS	744163200	51806208	744163200	51806208	744163200	51806208
ABMR	36863	35836	36863	35836	36863	35836
ARVC	263087	136128	263087	136128	263087	136128
Oddbotics	168541	168541	168541	168541	168541	168541
DHS	$\approx 10^{25}$	$\approx 10^{17}$	$\approx 10^{25}$	$\approx 10^{17}$	$\approx 10^{25}$	$\approx 10^{17}$
Pacemaker	9496	1880	9496	1880	9496	1880
Excavator	116	116	116	116	116	116

One way to address this limitation is to group the components that interact with each other and analyze their behaviors. This solves the problem by restricting the state space being analyzed, thereby reducing the number of system states being generated. Another alternative is to identify the ONBs first and then expand the rules that identify ONBs. This would reduce the number of rules we must analyze considerably and thereby addresses scalability problems.

6 RELATEDWORK

We can relate our work to two topics. The first topic is model slicing, and the second topic is the methods used to find and expose ONBs.

Model Slicing: Many techniques have been proposed for model slicing [22, 27, 45, 46] to reduce the state space being analyzed. For example, Wang et al. [45] proposed a slicing method for UML state machines using hierarchical automata. The method removes any concurrent states or hierarchies that are not relevant to the automata and reduces state space. Another example is the slicing method proposed by Lano [27], which divides state machine models into smaller and simpler models for analysis by considering the slicing of the state machine at a higher level of abstraction.

While these methods reduce state space being analyzed, none of them are suitable for when representing the component states together as system states. Therefore, we used a combinatorial approach to reduce the number of states that people need to analyze.

ONB analysis: To date, different techniques have been proposed to find and expose ONBs. Many researchers have analyzed ONBs using modeling techniques or modeling frameworks at early design stages. For example, Kurtoglu et al. [25] proposed a functional-failure identification and propagation (FFIP) framework to assess and evaluate ONBs during the conceptual design stage. FFIP uses component-driven hierarchical modeling in which ONBs are checked during behavioral simulation using functional failure logic(FFL) [25]. Day et al. [7] proposed a method to model ONBs using SysML. Their approach used SysML activity and state diagrams to model ONBs, and then used FMEA [40] to analyze ONBs. Other examples include correctness, modeling, and performance of aerospace systems (COMPASS) approach [5], failure mode effects analysis (FMEA) [9, 40], failure mode, effects and criticality analysis (FMECA) [6, 21], fault tree analysis (FTA) [23, 31], and probability risk assessment (PRA) [10]. In addition to these methods, many simulation based models [14, 16, 26] have been used to find ONBs.

Other researchers have used machine learning techniques to identify ONBs [43]. For example, Vemuri et al. [16, 43] proposed a neural–network based approach to classify ONBs in robot manipulators. In thier method, the robot first learns the series of ONB conditions and then identifies similar ONBs during simulation.

Some researchers have analyzed ONBs as part of testing by considering them at the experimental design stage. For example, Foyle and Hooey [13] proposed an off-nominal testing approach to locate ONBs in software under development. In this approach, off-nominal events based on human-system interactions are located and then used for testing.

While these techniques are able to find ONBs, none of these methods provide a systematic way of finding ONBs, and all consider ONBs exclusively from a human point of view. Unlike these approaches, we proposed an systematic approach that considers off-nominal behaviors from system's view point making the system more foolproof. The systematic approach we proposed can be used as requirements acquisition, which can be later used to build a better formal model.

7 CONCLUSION AND FUTUREWORK

In this paper, we have proposed Combi-CCM, a combinatorial approach to reduce the number of states that require manual analysis to expose ONBs. We have also introduced two CCM-EARS patterns to help engineers to elicit information when writing component-driven requirements that are less ambiguous, less wordy, and more verifiable than common natural language requirements. To evaluate the effectiveness of our approach, we conducted an empirical study using 7 requirements specifications. Our results indicate that the use of combinatorial techniques can reduce the number of states that require manual analysis while maintaining the ability to expose ONBs. From our results, we also learned that we have to consider a trade-off between the number of states that must be manually analyzed and the number of expanded rules that must analyzed when we choose an ONB detection technique.

While the results from Combi-CCM are promising when compared to CCM and ECCM, the approach still has some limitations, as discussed in Section 5. As part of future work, we plan to address these limitations using the strategies we mentioned in Section 5. In addition to addressing our limitations, we plan to create a broader set of CCM-EARS patterns that can accommodate undesired behavior requirements and other types of requirements that can be handled by EARS patterns. We also plan to improve the combinatorial approach so that it can generate fewer combinations of component states retaining dependencies among components, especially for systems that need multiple components to work simultaneously. Finally, we plan to perform additional experiments using different types and sizes of requirements documents to better understand and improve Combi-CCM.

ACKNOWLEDGMENT

This work was supported, in part, by NSF CAREER Award CCF-1564238 to University of North Texas.

REFERENCES

[1] D. Aceituna and H. Do. 2015. Exposing the susceptibility of off-nominal behaviors in reactive system requirements. In *IEEE 23rd International Requirements Engineering Conference (RE)*. 136–145. https://doi.org/10.1109/RE.2015.7320416

[2] C. M. Belcastro. 2012. *Validation and Verification (V&V) of Safety-Critical Systems Operating under Off-Nominal Conditions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 399–419.

[3] D. M. Berry. 2007. Ambiguity in natural language requirements documents. In *Monterey Workshop*. Springer, 1–7.

[4] Boston Scientific. 2007. *PACEMAKER system specification*. Technical Report. Boston Scientific.

[5] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. 2009. The COMPASS approach: Correctness, modelling and performability of aerospace systems. In *International Conference on Computer Safety, Reliability, and Security*. 173–186.

[6] Reliability Analysis Center. 1993. Failure Mode, Effects and Criticality Analysis (FMECA). (1993). http://www.dtic.mil/dtic/tr/fulltext/u2/a278508.pdf

[7] J. Day, K. Donahue, M. D. Ingham, A. Kadesch, A. Kennedy, and E. Post. 2012. Modeling Off-Nominal Behavior in SysML.. In *AIAA Infotech*. 19–21.

[8] X. Devroey, M. Cordy, P. Schobbens, A. Legay, and P. Heymans. 2015. State machine flattening, a mapping study and tools assessment. In *IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–8.

[9] V. Estivill-Castro, R. Hexel, and D. A. Rosenblueth. 2012. Failure mode and effects analysis (FMEA) and model-checking of software for embedded systems by sequential scheduling of vectors of logic-labelled finite-state machines. In *7th IET International Conference on System Safety, incorporating the Cyber Security Conference*. 1–6. https://doi.org/10.1049/cp.2012.1510

[10] N. Fenton and M. Neil. 2014. Decision Support Software for Probabilistic Risk Assessment Using Bayesian Networks. *IEEE Software* 31, 2 (Mar 2014), 21–26. https://doi.org/10.1109/MS.2014.32

[11] E. Feuvrier-Danziger, C. Dunkers, M. Kosowski, and D. Marschner. 2015. Oddbotics. (2015). https://sites.google.com/site/mrsdproject201415teamd/documents/presentations

[12] D. Firesmith. 2012. The Need to Specify Requirements for Off-Nominal Behaviors. (2012). https://insights.sei.cmu.edu/sei

[13] D. C. Foyle and B. L. Hooey. 2003. Improving evaluation and system design through the use of off-nominal testing: A methodology for scenario development. In *Wright State University*. 397–402.

[14] G. C. Fraccone, V. Volovoi, A. E. Colón, and M. Blake. 2011. Novel air traffic procedures: investigation of off-nominal scenarios and potential hazards. *Journal of Aircraft* 48, 1 (2011), 127–140.

[15] A. O. Gomes and M. V. M. Oliveira. 2011. *Formal Development of a Cardiac Pacemaker: From Specification to Code*. Springer Berlin Heidelberg, Berlin, Heidelberg, 210–225.

[16] D. L. Iverson. 2004. Inductive system health monitoring. In *In Proceedings of The 2004 International Conference on Artificial Intelligence (IC-AI04), Las Vegas*.

[17] M. Jackson. 2010. DigitalHome Software Requirements Specification. (2010). http://fmt.isti.cnr.it/nlreqdataset/

[18] E. Jee, I. Lee, and O. Sokolsky. 2010. *Assurance Cases in Model-Driven Development of the Pacemaker Software*. Springer Berlin Heidelberg, Berlin, Heidelberg, 343–356.

[19] D. C. Jensen and I. Y. Tumer. 2013. Modeling and Analysis of Safety in Early Design. *Procedia Computer Science* 16 (2013), 824 – 833. https://doi.org/10.1016/j.procs.2013.01.086 2013 Conference on Systems Engineering Research.

[20] Z. Jiang, M. Pajic, and R. Mangharam. 2011. Model-Based Closed-Loop Testing of Implantable Pacemakers. In *IEEE/ACM Second International Conference on Cyber-Physical Systems*. 131–140. https://doi.org/10.1109/ICCPS.2011.28

[21] Y. Jou, K. Yang, M. Liao, and C. Liaw. 2016. Multi-criteria failure mode effects and criticality analysis method: a comparative case study on aircraft braking system. *International Journal of Reliability and Safety* 10, 1 (2016), 1–21.

[22] H. Kim, D. Bae, V. Debroy, and W. E. Wong. 2011. Deriving Data Dependence from/for UML State Machine Diagrams. In *5th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*. 118–126.

[23] J. Kloos, T. Hussain, and R. Eschbach. 2011. Risk-Based Testing of Safety-Critical Embedded Systems Driven by Fault Tree Analysis. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 26–33. https://doi.org/10.1109/ICSTW.2011.90

[24] D. R. Kuhn, R. N. Kacker, and Y. Lei. 2016. Estimating t-Way Fault Profile Evolution During Testing. In *IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. 596–597. https://doi.org/10.1109/COMPSAC.2016.110

[25] T. Kurtoglu and I. Y. Tumer. 2008. A graph-based fault identification and propagation framework for functional design of complex systems. *Journal of Mechanical Design* 130, 5 (2008), 051401.

[26] T. Kurtoglu, I. Y. Tumer, and D. C. Jensen. 2010. A functional failure reasoning methodology for evaluation of conceptual system architectures. *Research in Engineering Design* 21, 4 (01 Oct 2010), 209–234.

[27] K. Lano. 2009. Slicing of UML state machines. In *Proceedings of the 9th WSEAS international conference on Applied informatics and communications*. World Scientific and Engineering Academy and Society (WSEAS), 63–69.

[28] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. 2008. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability* 18, 3 (2008), 125–148. https://doi.org/10.1002/stvr.381

[29] N. G. Leveson. 2004. Role of software in spacecraft accidents. *Journal of spacecraft and Rockets* 41, 4 (2004), 564–575.

[30] L. Liu, B. Pan, T. Wang, Q. Li, M. Aktas, and M. Gamell. 2012. Automatic Delivery System. (2012). http://eceweb1.rutgers.edu/~marsic/books/SE/projects/OTHER/2012-g4-report3.pdf

[31] H. K. Lo, C. Y. Huang, Y. R. Chang, W. C. Huang, and J. R. Chang. 2005. Reliability and Sensitivity Analysis of Embedded Systems with Modular Dynamic Fault Trees. In *TENCON - IEEE Region 10 Conference*. 1–6. https://doi.org/10.1109/TENCON.2005.300968

[32] M. Luisa, F. Mariangela, and N. I. Pierluigi. 2004. Market Research for Requirements Analysis Using Linguistic Tools. *Requirements Engineering* 9, 1 (2004), 40–56.

[33] K. Madala, H. Do, and D. Aceituna. 2017. Hierarchical Model Exploration for Exposing Off-Nominal Behaviors. In *14th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa) co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017)*.

[34] A. Mavi, P. Wilkinson, A. Harwood, and M. Novak. 2009. Easy approach to requirements syntax (EARS). In *17th IEEE InternationalRequirements Engineering Conference (RE)*. 317–322.

[35] A. Mavin. 2012. Listen, Then Use EARS. *IEEE Software* 29, 2 (March 2012), 17–18. https://doi.org/10.1109/MS.2012.36

[36] A. Mavin and P. Wilkinson. 2010. Big Ears (The Return of "Easy Approach to Requirements Engineering"). In *18th IEEE International Requirements Engineering Conference (RE)*. 277–282. https://doi.org/10.1109/RE.2010.39

[37] McMaster University. 2007. Pacemaker Formal Methods Challenge. (April 2007). http://sqrl.mcmaster.ca/pacemaker.htm

[38] M. Melin. 2010. Requirements Specification AB Mail Robot. (2010). http://www.isy.liu.se/edu/projekt/tsrt10/2010/postrobot-2010/images/general_doc.pdf

[39] E. So, J. Ajtum, Y. Moy, and Y. L. Quach. 2005. Requirements Specification AB Mail Robot. (2005). http://www.ecs.umass.edu/ece/sdp/sdp05/preston/sdp_data/Requirement%20Specification.doc

[40] S. Teng and S. Ho. 1996. Failure mode and effects analysis: an integrated approach for product design and process control. *International journal of quality & reliability management* 13, 5 (1996), 8–26.

[41] L. A. Tuan, M. C. Zheng, and Q. T. Tho. 2010. Modeling and Verification of Safety Critical Systems: A Case Study on Pacemaker. In *4th International Conference on Secure Software Integration and Reliability Improvement*. 23–32. https://doi.org/10.1109/SSIRI.2010.28

[42] A. Valmari. 1998. *The state explosion problem*. Springer Berlin Heidelberg, Berlin, Heidelberg, 429–528.

[43] A. T. Vemuri, M. M. Polycarpou, and S. A. Diakourtis. 1998. Neural network based fault detection in robotic manipulators. *IEEE Transactions on Robotics and Automation* 14, 2 (Apr 1998), 342–348. https://doi.org/10.1109/70.681254

[44] S. Verma, S. Lozito, K. Thomas, and D. Ballinger. 2008. Procedures for Off-Nominal Cases: Very Closely Spaced Parallel Runway Operations. In *IEEE/AIAA 27th Digital Avionics Systems Conference (DASC)*. 2.C.4–1âĂŞ2.C.4–11.

[45] J. Wang, W. Dong, and Z. Qi. 2002. Slicing Hierarchical Automata for Model Checking UML Statecharts. *Lecture Notes in Computer Science* 2495 (2002), 435–446.

[46] N. Yatapanage, K. Winter, and S. Zafar. 2010. Slicing behavior tree models for verification. *Theoretical Computer Science* (2010), 125–139.

[47] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn. 2013. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*. 370–375.