A Knowledge Acquisition Approach For Off-Nominal Behaviors

Kaushik Madala
Computer Science and Engineering
University of North Texas
kaushik.madala@my.unt.edu

Hyunsook Do
Computer Science and Engineering
University of North Texas
hyunsook.do@unt.edu

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

***Abstract*—Natural language requirements often ignore unexpected or off-nominal behaviors (ONBs), which can result in catastrophic accidents in safety-critical systems. While some existing techniques can help identify ONBs, most of them are not systematic and algorithmic, and also they require a lot of human effort. In this paper, we propose an algorithmic and systematic approach for knowledge acquisition of ONBs in component-based systems using a modified Apriori algorithm. Our approach analyzes component state transition rules to identify dependencies among components, which are used to group components that are dependent on each other into component sets. These sets are used for analysis of possible ONBs. We conducted an empirical study to evaluate our approach. Our results indicate that the component sets generated using our approach are able to expose missing dependencies and ONBs with much less human effort when compared to CCM.**

I. Introduction

Systems such as automotive systems, embedded systems, and robotic systems are often susceptible to off-nominal behaviors. Off-nominal behaviors (ONBs) [1] are the behaviors of a system that are unexpected and unforeseen. It is crucial to find ONBs in prior mentioned systems as they can result in hazardous, and catastrophic events or unacceptable events. For instance, if an automated car does not stop when there is a person or a car in front of it, such a behavior is an ONB. Because ONBs can result in such critical situations as above, it is better to consider them from early phases of the software development life cycle. However, knowledge acquisition of ONBs is difficult due to the lack of information in requirements documents. Most natural language (NL) requirements documents consider only expected behaviors of the systems and do not consider ONBs [1].

To date, some researchers have proposed the approaches that analyze possible ONBs or identify their occurrences [2], [3]. For example, Bozanno et al. [3] proposed the correctness, modeling, and performability of aerospace systems (COMPASS) approach, which considers ONBs in the form of an error model, fault injections, and properties. These models and properties are used by various formal approaches to identify the safety issues [4]. Kurtoglu and Tumer [2] proposed a functional failure identification and propagation (FFIP) approach, which identifies functional failures by creating configuration flow graphs and using functional failure logic to create failure identification functions. However, ONBs identified in these approaches are mostly based on simulation experiments, previous domain knowledge, or experiences. Both aforementioned approaches including most of the current approaches [5], [6] do not provide a systematic knowledge acquisition process for identifying ONBs or for checking whether there are any unknown known ONBs. Unknown known ONBs are the ONBs that can be identified by stakeholders, but they are undetected due to the lack of understanding of complex system behaviors.

In our previous work, a causal component model (CCM) [7] approach was proposed that exposes ONBs in NL requirements. The approach requires users to go through all possible states of the system to find ONBs. While the results of the approach are promising, the approach is not systematic and takes a lot of human effort and time for ONB knowledge acquisition. In addition, CCM or other techniques mentioned earlier are ad-hoc approaches, which could result in not identifying some ONBs. From our previous experiments [8], we learned that there is a correlation between dependencies among components and ONBs, and missing dependencies can lead to missing the detection of ONBs.

To address the limitations of the existing approaches and to confirm our findings from our previous experiments, we propose a new knowledge acquisition approach that identifies ONBs in a systematic way using a modified Apriori algorithm. The Apriori algorithm [9] we used in our approach is an association rule mining algorithm. We used the modified version of this algorithm to reduce the human effort and time for manual analysis. Our approach also utilizes the concepts of modeling requirements from CCM [7] and FFIP [2]. It generates component sets that need to be manually analyzed for ONBs and missing dependencies between components in the system. To investigate the effectiveness of our approach, we conducted an empirical study using Digital Home System (DHS) [10] requirements and compared with CCM. The results show that our approach is able to find more ONBs and missing dependencies with less effort compared to CCM. Our approach also helps with identifying additional components and properties of the system that are missing in the requirements.

Section II presents background information and related work. Section III provides preliminary information required to understand our approach. Section IV describes the proposed approach and Section V presents the empirical study, and its results. Section VI discusses results and their implications, and Section VII presents conclusions and future work.

978-1-5386-8410-8/18/$31.00 ©2018 IEEE
DOI 10.1109/RESACS.2018.00012

IEEE computer society

II. Background and Related Work

A. Background

We briefly explain the Apriori algorithm used in our approach. Our approach uses the concepts from the CCM [7] to build component state transition rules, and from the FFIP [2] to identify the undefined variables and missing dependencies. Due to space limitations, we omit the descriptions of these two approaches (see [2], [7] for details).

Apriori algorithm: Apriori algorithm [9] is an association rule mining algorithm used in data mining. It identifies the items that occur frequently in a database and extends them to larger item sets that occur frequently, i.e., it generates item sets of items that co-occur frequently and used for tasks such as finding a set of items that are purchased together in retail stores. In our approach, we modified the Apriori algorithm to fit the context of components in a system. We are interested in finding components that affect behaviors of other components directly or indirectly (via a sequence of transitions). In Apriori, for an item set to be frequent, the number of occurrences of the item set needs to satisfy a threshold, which is defined manually based on experiences. Because we apply the modified Apriori to find ONBs in systems including safety-critical systems, every dependency between components is important, so we used a threshold of 1, which implies the dependency between components is considered if there is at least one input rule that supports the dependency.

B. Related Work

1) Model slicing: Model slicing [11] reduces large models into a set of small models to make analysis easier and to date, many researchers have proposed model slicing techniques. For example, Wang et al. [11] proposed a technique for model slicing on hierarchical automata by representing a model as a UML state chart, and removing hierarchies or concurrent states that are not relevant to hierarchical automata. Another approach is the modeling slicing proposed by Lano [12], in which UML state machines are sliced by selecting states and eliminating features that do not contribute to the selected states. While these techniques help reduce the model complexity, they are suitable for system level models, but not for component oriented models. Further, these slicing techniques do not cover all possible dependencies among components, and can result in missing ONBs. Our approach overcomes the drawbacks of these approaches by dividing a model into smaller models and helps in finding ONBs.

2) ONB analysis: To find and analyze ONBs, various types of approaches have been proposed. Some researchers have tried to find ONBs at an early design phase by converting requirements into a model or formal specifications. For example, Day et al. [5] used SysML to model ONBs using activity and state diagrams, and then used failure modes and effects analysis (FMEA) [13] to analyze ONBs. Andrews and Ridley [6] proposed cause consequence analysis (CCA) for static systems, which finds ONBs by analyzing failures using fault trees and event trees. Other methods that are used for analysis or detection of ONBs include FMEA [13] and FFIP [2].

Other studies have applied machine learning to train models to learn expected behaviors and ONBs, and these learned models are used to predict if a new behavior of the system is an expected behavior or if it is an ONB [14]. For example, Iverson [14] proposed a machine learning based approach to find ONBs by training a model using the expected and allowed behaviors of an inductive monitoring system, which are archived. Any behavior that is predicted as not expected is considered as an ONB.

However, none of the aforementioned approaches provide or discuss a systematic way to acquiring knowledge of ONBs. Most of the techniques use previous knowledge of ONBs to define them, but do not provide a way to cross-check if any ONB to be addressed is missed. In this paper, we propose a systematic and algorithmic approach for knowledge acquisition of ONBs to address the limitations of the existing approaches.

III. Preliminary

Our approach uses model elements: components, states and, transition conditions similar to CCM. These model elements are illustrated in Figure 1. Due to space limitations, we forgo explaining the details of model elements: components, states and transition conditions but they can be found from [7].

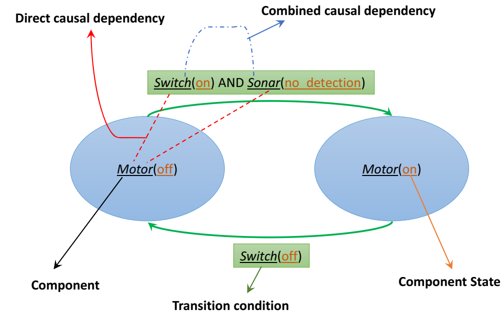

Fig. 1. Model elements and types of dependencies

While components, their states, and transition conditions aid in analyzing the behaviors of a system, we need to consider properties of a component or a group of components, because

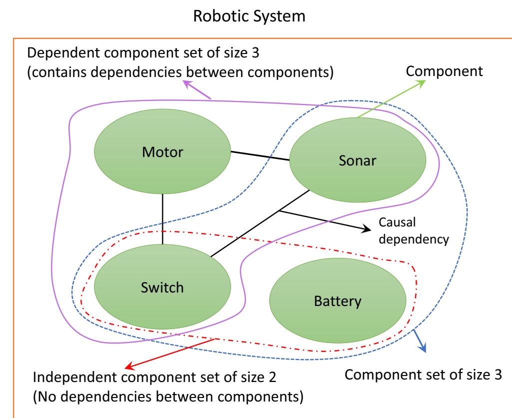

Fig. 2. Component sets, size, and dependent and independent component sets

ignoring properties can result in an incomplete analysis of the model. We defined such properties as variables, which we adopted from the FFIP approch [2]. A variable set with 'n' variables is denoted as V = $\{V_1, V_2, \cdots, V_n\}$, where any variable V_i must satisfy the property that it belongs to at least one component. An example of a variable is "speed" for the component *motor*.

Using the model elements (components, their states and transition conditions), we construct rules. A rule represents a component state transition, and a system can contain many component state transitions. If a system has 'n' component state transitions, then a set of rules, R = $\{R_1, R_2, R_3, \cdots, R_n\}$, denotes these 'n' component state transitions, such that each rule R_i is in the form of *Transition Condition : Component(Current State) $\rightarrow$ Component(Next State)*, and each model element in the rules satisfies their definitions. An example of a rule is, *Switch(off) : Motor(on) $\rightarrow$ Motor(off)*.

Terminology: We define terms that are used in our approach.

1) *Direct causal dependency:* A component or a set of components is directly causally dependent on another component if the component or the set of components causes other component to change its state. For example, in Figure 1, the state of *motor* is caused by the state changes in components *switch* and *sonar*. This implies that *motor* and *switch* as well as *motor* and *sonar* have direct causal dependencies. We also use the term 'direct dependency' to refer to 'direct causal dependency.'
2) *Combined causal dependency:* A combined causal dependency or a combined dependency exists between or among components when their current states collectively cause a state change in a component. For example, in Figure 1, when component *sonar* is in 'no_detection' state and *switch* is in 'on' state concurrently, the transition occurs in *motor*. In this example, we consider *sonar* and *switch* to have a combined causal dependency.
3) *Component set:* A component set is a collection of any number of components in a system. It can have all the components in the system or a subset of them. An example of a component set in Figure 2 is {*sonar*, *switch*, *battery*}, which is one of the component sets of the robotic system. The size of a component set is the number of components that are present in a component set. In our example, the size of the component set is 3.
4) *Dependent components:* Dependent components are the components that have direct or combined dependencies between or among them. For example, Figure 2 contains *motor* and *switch* having a dependency (illustrated as a line between components). These components are considered as dependent components. A component set containing such dependent components is defined as a dependent component set (DCS). Figure 2 illustrates a DCS with components *motor*, *sonar*, and *switch*. We generate DCSs to reduce the number of concurrent components states subjected to manual analysis.
5) *Independent components:* Independent components are the components that do not have direct or combined causal dependencies. For example, in Figure 2, the components *switch* and *battery* do not have any dependencies (no lines between them). These components are considered as independent components. The set of independent components is defined as an independent component set and if such a set contains two components, it is defined as a pairwise independent component set (PICS). We use PICSs to find missing dependencies among components. An example of a PICS shown in Figure 2 is a set of components *switch* and *battery*.
6) *Instance:* An instance is a snapshot of concurrent states of the components in a component set. An example of an instance from the DCS in Figure 2 is {Motor(on), Sonar(off), Switch(on)}. The size or length of an instance refers to the number of component states in that instance. In our example, the size of an instance is 3.

IV. Approach

The overview of our approach is shown in Figure 3. The input to the approach is NL requirements and the output is a list of identified problems. The ovals indicate the processes and the rectangles depict the inputs and outcomes associated with the processes. The numbers over ovals represent step numbers. The requirements will be corrected based on the problems found from the proposed approach similar to [7], but the correction process is out of scope of this paper.

In our approach, first, we find the model elements and use them to create component state transition rules (Step 1). These rules are then processed by the modified Apriori algorithm, which generates component sets based on causal dependencies among components (Step 2). Once the components sets are generated, the stakeholders manually analyze them to find undesired behaviors and missing dependencies (Step 3).

To illustrate our approach, we use an automatic robot vacuum cleaner (ARVC) requirements [15]. The ARVC requirements describe about how the system needs to plan and clean the room, as well as how to detect any obstacles in the way and change directions accordingly. We analyze this system for ONBs as any unexpected behavior by the vacuum cleaner might result in trash and dust disposal in the room. For example, overfilling a vacuum bag might lead to a burst out and have the entire room covered in dust and trash. The detailed description of each step in the approach is provided as follows.

Step 1. *Converting NL requirements into rules*: To convert NL requirements into component state transition rules, we need to identify components, their states, and transition conditions from the requirements [7]. It is possible that the requirements do not have all these model elements. In such cases, we start with identifying the components of the system. A component must be part of the system with a well-defined functionality. It is usually a hardware component with its associated software modules. Once the components are identified, we define the states of the components based on their intended functionality in the product. For example, for a *motor* component in robotic systems, we can define two states, "off" and "on" initially

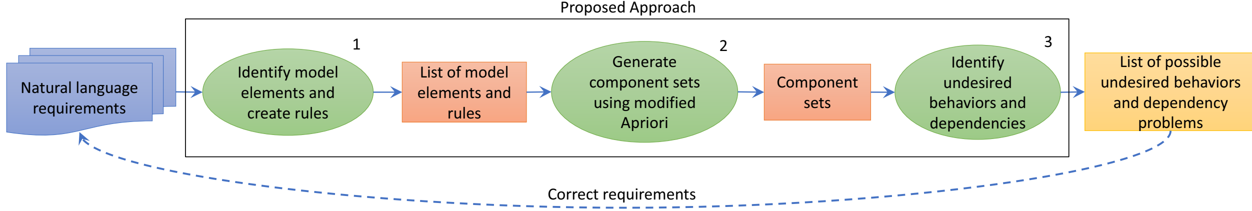

Fig. 3. **Overview of the Proposed Approach**

because the main functionality of the motor is to give a mobility to the system. For every component, we also identify the corresponding variables that represent its properties. For example, for a *motor* component, a possible variable is 'speed'. Regarding variables, we only find variables to be defined using our approach but their analysis is out of scope of this paper, as they need hardware requirements.

When we applied these steps to the ARVC system, we identified the components, their states and variables listed in Table I. In Table I, any state in the form *a.b* implies *b* is a sub-state of state *a*. For example, "on.detect" state of *sonar* means, the component is in "detect" sub-state of "on" state.

We then check how components can change their states and identify transition conditions between states. For example, for the *ultrasonic sensor* component in a robotic system, a transition from "no detection" state to "detected" state can occur when an object is blocking the path of the robot. Once the model elements are identified, we write component state transition rules, where each rule is in the form of *Transition Condition : Component(Current State)* $\rightarrow$ *Component(Next State)*. While creating rules, we ensure that no known component of the system is excluded in the rules.

TABLE I
COMPONENTS, STATES AND VARIABLES OF ARVC

Component Name	States	Variables
Sonar	off, on.detect, on.nodetect	distance
Digital compass	off, on	None
Battery level signal	low, high, normal	battery level
Battery	outofplace, inplace.charging, inplace.discharging	battery level
Vacuum Control	off, on	None
Motor Control	off, on.move, on.idle	speed
Switch	off, on	None
LED	off, on	None
Bumper sensor	off, on.detect, on.nodetect	None
Vacuum bag	full, filling, empty	volume

The number of rules created for ARVC is 45, but due to space limitations, we show a small portion of the rules:

1) User(switch_on) : Switch(off) $\rightarrow$ Switch(on)
2) User(switch_off) : Switch(on) $\rightarrow$ Switch(off)
3) Switch(on) : Sonar(off) $\rightarrow$ Sonar(on.nodetect)
4) Switch(off) : Sonar(on.detect) $\rightarrow$ Sonar(off)
5) Switch(off) : Sonar(on.nodetect) $\rightarrow$ Sonar(off)

Step 2. *Generate component sets using the Modified Apriori*: Once the rules are created, the modified Apriori algorithm processes the rules. As explained in Section II, Apriori is an association rule mining algorithm, and it is often used to find frequent item sets (e.g., a group of items that occur together). Given a set of items, Apriori finds subsets of item sets that occur frequently if the number of times they occurred is greater than a threshold. In our approach, rather than just finding components that interact frequently, we also need to consider causal dependencies. In addition, we generate pairwise independent components to cross-check the presence of missing dependencies. Due to these reasons, we modified the Apriori algorithm to fit our context.

The modified Apriori algorithm generates component sets based on dependencies and thus it reduces the size of instances need to be manually analyzed unlike CCM, where no dependencies among components are taken into account and all system components are analyzed simultaneously. For example, in the ARVC system, when we use CCM, we need to analyze all 10 concurrent components' states, but in our approach, we only analyze concurrent components' states in the selected component sets (e.g., when we analyze a component set containing motor controller, sonar, and switch, we analyze concurrent component states of only these three components because the changes in sonar and switch will change the state in a motor controller regardless of other components' states).

Algorithm 1 illustrates the modified Apriori algorithm. First, the Algorithm 1 creates the pairwise component dependency profiles (line 1). Algorithms 2 and 3 can be used to generate dependency profiles.

Algorithm 2 considers direct causal dependencies between components and generates the number of times the two components are directly causal dependent from a given set of rules. The direct causal dependency refers to a relationship between components where one component triggers a state transition in other. Algorithm 2 extracts transition conditions and a component that has a state transition from each rule in the ruleset (lines 1–3). If the transition condition has multiple conditions (e.g., In the following rule, Temperature(>80) AND Switch(on) : Motor(on) $\rightarrow$ Motor(off), the transition condition has two conditions, one being temperature >80 and the other being switch is on), the transition condition is split into tokens such that each token contains individual condition (line 4). If the token is a component state, then the component in the token and the component whose state is changed because of it are considered to have a dependency and the number of times the dependency occurred between them is incremented by 1 (lines 5–9).

We also created dependency profiles considering combined causal dependencies. As defined in Section III, for component state transitions that occur as a result of other components states, if the transition condition has more than one component, we

Algorithm 1: Modified Apriori algorithm

```
input : Set of Rules, R = {R_1, R_2, R_3, ··· , R_n}
        Set of Components, C = {C_1, C_2, ··· , C_n}
        Threshold, th = 1
        Number of component in system, num_comp
output: Set of Dependent component sets, DCS
        Set of Pairwise independent component sets, PICS
        Set of Independent component sets, ICS
1  Generate pairwise component dependency profiles, PCDP
2  foreach component set CS in PCDP do
3      if dependencyprofile < th then
4          PICS ← PICS + CS
5      end
6      else
7          DCS ← DCS + CS
8      end
9  end
10 for size ← 3 to num_comp do
11     SS ← generate subsets with length of size - 1
12     n ← number of subsets generated
13     if at least n-1 subsets ss_1,ss_2 ∈ SS are dependent then
14         DCS ← DCS + CS
15         DCS ← DCS - SS
16     end
17     else
18         ICS ← ICS + CS
19     end
20     if all component sets in SS ∈ (ICS ∪ PICS) then
21         Stop
22     end
23 end
```

Algorithm 2: Generate pairwise component dependency profiles (only direct dependencies)

```
input : Set of Rules, R = {R_1, R_2, R_3, ··· , R_n}
        Set of Components, C = {C_1, C_2, ··· , C_n}
output: Component dependency profiles, CDP
1  foreach R_i in R do
2      TC ← Transition Condition in R
3      c ← Component in R
4      TCtokens = TC.split(" ") ;    // divides the transition
        condition string into tokens
5      foreach TCtoken in TCtokens: do
6          if TCtoken ∈ C then
7              CDP(TCtoken, c) ← CDP(TCtoken, c)+1
8          end
9          else
10             break
11         end
12     end
13 end
```

Algorithm 3: Generate pairwise component dependency profiles (direct and combined dependencies)

```
input : Set of Rules, R = {R_1, R_2, R_3, ··· , R_n}
        Set of Components, C = {C_1, C_2, ··· , C_n}
output: Component dependency profiles, CDP
1  foreach R_i in R do
2      TC ← Transition Condition in R
3      c ← Component in R
4      TCtokens = TC.split(" ") ;    // divides the transition
        condition string into tokens
5      foreach TCtoken in TCtokens: do
6          if TCtoken ∈ C then
7              CDP(TCtoken, c) ← CDP(TCtoken, c)+1
8          end
9          else
10             break
11         end
12     end
13     foreach TCtoken1 in TCtokens do
14         if TCtoken1 ∈ C then
15             foreach TCtoken2 in TCtokens do
16                 if TCtoken2 ∈ C AND TCtoken1 ≠ TCtoken2 then
17                     CDP(TCtoken1, TCtoken2) ← CDP(TCtoken1,
                         TCtoken2)+1
18                 end
19             end
20         end
21     end
22 end
```

consider that there is a combined dependency between or among the components in the transition condition. We considered these combined dependencies because we want to evaluate if these dependencies can aid in finding more ONBs.

Algorithm 3 generates dependency profiles considering direct and combined dependencies. The lines 1–9 in the algorithm are similar to Algorithm 2. In addition to these steps, if multiple components resulted in a transition of other component, the algorithm considers the multiple components in the transition condition to be combined causally dependent on each other, and increments the dependency profile by value 1 (lines 10–14). Once the dependency profiles are calculated, using Algorithm 2 or Algorithm 3, Algorithm 1 generates DCSs and PICSs. To generate DCSs and PICSs, we start with the component sets that have two components each. This is because the dependency profiles are generated for pairs of components. Any pairwise component set, whose profile value is smaller than 1, is considered to be a PICS (lines 3–4 of Algorithm 1). If the profile value is greater than 1, the pairwise component set is considered to be a DCS (lines 5–6 of Algorithm 1). We consider the value of the threshold to be 1 in our approach because the systems we focus include safety-critical systems in which every dependency is important to examine. If the goal is to find frequent component sets, then the threshold can be varied according to interest of users. Once we are done with pairwise components, we consider component sets from size three to *num_comp*, where num_comp is the total number of components in the system and the size as mentioned in Section III is the number of components present in the component set (line 7). For each size of the component sets, we find dependent component sets. When the size of the component sets is greater than 2, for a component set to be DCS, at least 'n-1' subsets of the component set must be DCSs, where 'n' is the number of subsets for the component set considered, whose size is 'component set size - 1' (lines 8–10 in Algorithm 1).

We considered only 'n-1' component sets as we found 'n-1' component sets are good enough to find a possible dependency. If a component set is found to be dependent, all the subsets of that component set are removed from the dependent component set because the component set covers those subsets (lines 11–12 in Algorithm 1). If less than 'n-1' subsets of the component set are dependent, then the component set is considered as an independent component set (lines 13–14 in Algorithm 1). For any size of component sets 'n', if all component sets

with size 'n-1' are independent, there is no need to further analyze a component set of size 'n' for DCSs, and therefore can stop algorithm as all possible DCSs are found (lines 15–16 in Algorithm 1).

In our running example, let us consider the sample rules introduced earlier. The first two rules do not give any component dependency profile because the transition condition is purely environmental. However, rules 3 and 4 have system originated transition conditions. By parsing rule 3, the Apriori algorithm creates a component set CS = {Switch, Sonar} and assigns value 1 to their dependency profile. This value is incremented when parsing next two rules (rules 4 and 5) because *switch* causes a state transition in *sonar*. The value of the dependency profile is 3, which makes the component set, *switch* and *sonar*, to be a dependent component set. Once all the dependency profiles are created, we consider component sets of size greater than 2, CS = {Switch, Sonar, Motor Control} in our example. To evaluate if this component set is dependent, we generate its subsets of size 2: {Switch, Sonar}, {Switch, Motor Control}, and {Sonar, Motor Control}. Because all three subsets are dependent component sets, the component set {Sonar, Switch, Motor Control} is a dependent component set and the subsets are removed from the list of dependent component sets.

For ARVC, by applying the modified Apriori, in total, 38 sets are produced: 16 of them are DCSs and the rest are PICSs. In ARVC, the number of the component sets (both dependent and pairwise) generated using only direct dependencies is same as the number of the component sets generated using direct and combined dependencies because the input rules do not have any components with combined dependencies. A sample of the component sets identified is shown as follows:

DCSs: (1) Switch, Sonar, Motor Control; (2) Switch, Bumper sensor, Sonar; and (3) Switch, Bumper sensor, Motor Control.

PICSs: (1) Switch, Battery; (2) Vacuum bag, Battery; (3) Sonar, Battery; and (4) Vacuum bag, LED.

Step 3. *Identify undesired behaviors and dependencies*: Once a list of the component sets is generated, any sets that contain only sensors as components are automatically disposed as sensors do not interact with or affect each other. The stakeholders go through the rest of PICSs and discard any sets that are of no interest. For example, in our running example, we can discard a vacuum bag and LED that shows the battery status because they do not affect each other. The PICSs that are of interest will be analyzed further along with DCSs. For example, the behaviors of sonar and battery are not considered together, but the behavior of sonar can change with low battery. Thus, these sets are analyzed further to check if ONBs can occur. During this analysis, we also check if new variables are required to represent the properties or the status of the components, and if missing a component resulted in missing dependencies between components. We also cross-check what factors in the system and environment need to be considered to address missing dependencies. This is usually done by discussion among stakeholders.

The DCSs and interested PICSs are manually analyzed including the combinations of components' states within a component set. In our running example, we analyzed all 38 sets in 45 minutes, which is much faster than CCM that requires 32.4 hours to analyze. We estimated the time for CCM based on the random sample of 400 system states. Because our approach requires 437 instances for the manual analysis, we analyzed a similar number of the system states in CCM.

In our running example, upon analyzing the system for ONBs, and dependencies, we found not only missing dependencies and ONBs but also found the need for new components, and variables. Examples of ONBs we found are: vacuum bag is full but the vacuum controller is on, battery is charging and a motor controller began to move, and sonar detected an obstacle but the motor controller is still moving. We are also able to find missing dependencies, such as the dependency between sonar and battery level and the dependency between a vacuum controller and a vacuum bag. In this system, we found that additional components need to be considered. For example, in order for the vacuum controller to get a signal that indicates the vacuum bag is full, it is necessary to have a component that detects the vacuum bag is full. We also found that to check how the low battery level can affect the system, we need to consider variables that indicate the amount of energy being consumed by the components in the system. The information about ONBs and missing dependencies can help correct and improve the requirements document.

V. Empirical Study

Our empirical study considers two research questions:

RQ1: Does the proposed approach help in identifying missing dependencies?

RQ2: Does the proposed approach reduce human effort and time compared to CCM?

A. Object of analysis

This study uses a Digital Home System (DHS) requirements document [10]. The 15-page document details the functionality of a digital home system: temperature and humidity can be controlled via digital devices such as smart phones and personal digital assistants (PDAs); safety alarms will be triggered automatically if contact sensors near doors or windows are set. For our study, we have considered only behavioral requirements of the system.

B. Variables and Measures

Independent Variables: The independent variable is the technique used to find ONBs and missing dependencies.

1) Apriori I (heuristic) uses the modified Apriori algorithm that considers only direct causal dependencies to generate component sets.
2) Apriori II (heuristic) uses the modified Apriori algorithm that considers direct and combined dependencies among components to generate component sets.
3) CCM (control) [7] requires stakeholders to examine all possible system states to identify possible ONBs in the system.

We did not consider any other control techniques because, to our knowledge, no other techniques are proposed to expose and identify ONBs at requirements engineering phases.

Dependent Variables: The dependent variable for RQ1 is the number of missing dependencies between components found by each technique and for RQ2, the amount of manual analysis time taken by each method.

C. Experimental Setup and Procedure

To perform our study, we implemented the Apriori algorithm in Java. CCM tool is implemented in C#. We followed the procedure described in Section IV to collect data. We found 76 components (a web server, a gateway device, thermostats, humidistats, contact sensors, and alarms) and created 483 rules. Using these rules, our tool generated dependency profiles that are used to generate DCSs and PICSs. Once the component sets are generated, PICSs that only contain sensors as components are automatically discarded. We also discarded the sets from the rest of PICSs if they contain components that do not affect each other. We analyzed all combinations of component states for DCSs to find ONBS and for the retained PICSs to see if there are missing dependencies. During this process, we also check whether additional variables (e.g., variables representing component properties) need to be defined. The final output is the list of ONBs and missing dependencies found.

For the CCM approach, to find ONBs, we generated all possible system states, which need manual analysis. However, because the number of system states generated by CCM are 7.12×10^{34} and it takes a lot of time to analyze them, we only considered a random sample of 400 system states, given each system state contains 76 component states that need to be examined simultaneously. The results of CCM are estimations derived based on the analysis of these 400 system states.

D. Results

RQ1 Results: To evaluate RQ1, we measured the number of missing dependencies for each technique. As shown in Table II, we were able to find missing dependencies using Apriori I, and Apriori II. The missing dependencies we found are between contact sensors and a gateway device, as well as a gateway device and alarms. The dependencies between them must be considered as they play a major role in maintaining an effective home security system. However, using CCM, we were not able to find missing dependencies. This might be because we only considered a random sample of 400 system states out of 7.12×10^{34} system states that need to be manually analyzed. Further, each system state represents 76 concurrent component states and we might have overlooked some missing dependencies due to such a large number of system states.

TABLE II
NUMBER OF MISSING DEPENDENCIES FOUND

Technique	Hueristic		Control
Technique name	Apriori I	Aprioi II	CCM
# of missing dependencies	2	2	0

RQ2 Results: To evaluate RQ2, we measured the time required for analyzing ONBs and missing dependencies for all three approaches. The results are shown in Table III. Because the control technique does not have the concept of component sets, all the entries related to the component sets are not applicable. As shown in Table III, Apriori I generated 475 more component sets than Apriori II. We also observe that the numbers of DCSs and PICSs are higher in Apriori I than Apriori II. However, after discarding the PICSs automatically and manually, there are only 104 independent component sets that need to be analyzed for both Apriori I and Apriori II.

The results from Table III show that the number of instances that need to be manually analyzed is also higher in Apriori I than in Apriori II. This is because Apriori I has a higher number of DCSs than Apriori II. However, the size of instances (i.e., the number of concurrent component states in an instance) in Apriori I is smaller than one in Apriori II. As shown in Table III, the total number of instances that must be manually analyzed by Apriori I is 9076. Out of 9076, 624 instances are of size 2 (i.e., they have only two component states), and the rest of them (8452 instances) are of size 3 (i.e., three component states must be considered concurrently). In the case of Apriori II, the number of instances that need to be analyzed manually is 8836, and out of 8836, 7666 instances are of size 4, 576 instances are of size 3, and 624 are of size 2. The number of instances that need to be analyzed manually in CCM are 7.12×10^{34} where each instance is of size 76, as each instance in CCM represents a system state, which represents all concurrent components' states. As mentioned earlier, for CCM, we manually analyzed only a random sample of system states. The number of ONBs found for each approach based on manual analysis is illustrated in Table III. We can observe that Apriori I and Apriori II were able to find 268 ONBs, whereas CCM was able to find only 164 ONBs. The additional 104 ONBs found by the heuristics caused by missing dependencies, which we were not able to discover when using CCM. The amount of time taken for each technique is shown in Table III. We can see that Apriori I took the smallest amount of time among all three approaches. Apriori II took around 2 hours more than Apriori I. This is because Apriori I has instances of a smaller size than instances of Apriori II. When we compare the time taken by heuristic techniques to that of time estimated for CCM, we can observe there is a huge difference between them, which is about 10^{27} years. The estimated time for CCM is based on time we calculated for analyzing the random sample of 400 system states.

Further, using heuristic approaches, we were able to identify a variable called "the number of contact sensors that went off at a given time," as we found it helps in identifying false alarms, which we cannot find in CCM because it does not analyze dependencies among components.

TABLE III
EFFORT AND TIME REQUIRED FOR MANUAL ANALYSIS

Technique	Hueristic		Control
Technique name	Apriori I	Aprioi II	CCM
Total # of component sets	2952	2477	N/A
# of DCSs	289	154	N/A
# of generated PICSs	2663	2322	N/A
# of discarded PICSs	2559	2218	N/A
# of analyzed PICSs	104	104	N/A
# of instances to be analyzed	9076	8836	7.12×10^{34}
Size of instances	3 and 2	4, 3 and 2	76
Number of ONBs found	268	268	164
Manual analysis time	7.38 hrs	9.32 hrs	$\approx 10^{27}$ yrs

VI. Discussion

We drew the following observations from our results. First, our results indicate that considering dependencies among the components reduces human efforts and time. Our approach is able to find more ONBs and missing dependencies with far less time and effort compared to CCM. As presented in Section V-D, our heuristics took 7.38 hours and 9.32 hours for manual analysis and found 268 ONBs, while CCM would require approximately 10^{27} years and found 164 ONBs (estimated based on analysis of random sampling). These results indicate that our approach would be more beneficial when it is applied to industrial applications, which are often larger than the system we used in our study. We also observed some differences between our two heuristics. From the results, we learned that if a set of components in a system frequently affects other sets of components, Apriori II is more appropriate because it considers combined causal dependencies. However, for the rest of the cases, Apriori I could be more suitable because it reduces more human efforts. This means trade-offs should be considered when choosing an algorithm.

Second, our results suggest that often the properties of components are not considered in NL requirements. Ignoring properties might result in incomplete analysis of the generated model, as in safety-critical systems such as embedded and robotic systems, the properties play an important role in making system more robust. For example, as we observed in our study, the number of contact sensors that went off is used to identify false alarms. When we ignore this property, it can result in improper analysis of the security system in the DHS.

The reduction in time and effort, and a small size of instances that need manual analysis make the proposed approach easier to be accepted by practitioners. Our technique can also be leveraged in the existing formal verification tools by providing input models for them. For example, the ONBs, missing dependencies, and properties found in our approach can be utilized to create error models, and properties that can be used as inputs to COMPASS [3], a formal verification tool.

Limitations: One limitation is that if all components affect each other, our approach still requires stakeholders to examine all possible system states. However, this is not the case for most of the systems, as not every component in the system affects every other component. Another limitation of our approach is that manual analysis can be error-prone. We plan to address this issue by identifying ONBs from knowledge bases of previously learned ONBs.

Threats to Validity: One of the threats to validity is the time that we measured for manual analysis. The time reported in this study is the time taken by a single person who has knowledge on the system. Non-experts might take longer than we reported. However, in practice, typically more than one person is involved in requirements analysis, so the analysis time of our proposed approach can be reduced. Another threat to validity is the results of our approach may not be generalized to other systems. Every system is unique and can have requirements of better or poor quality which can affect the results. This threat can be addressed by performing additional experiments with various types of systems and requirements.

VII. Conclusion and Future Work

We proposed a systematic and algorithmic approach for knowledge acquisition of ONBs from NL requirements. Our approach uses a modified Apriori algorithm to create DCSs and PICSs based on causal dependencies among components. These component sets are analyzed to find ONBs and missing dependencies. We evaluated our approach using a running example of ARVC requirements and by conducting an empirical study on DHS requirements. Our results indicate that our approach can find more ONBs and missing dependencies, and also reduce a significant amount of human efforts and time compared to the existing technique, CCM. The results also show that our approach can identify additional components and properties of the system. While the study shows the promising results, our approach has some limitations as discussed in Section VI. We plan to address these limitations as part of future work. We also plan to extend our approach to find ONBs considering environmental contexts (e.g., different terrains for robots) and physical or hardware properties.

VIII. Acknowledgment

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

References

[1] D. Firesmith, "The Need to Specify Requirements for Off-Nominal Behaviors," 2012. [Online]. Available: https://insights.sei.cmu.edu/sei{_}blog/2012/01/the-need-to-specify-requirements-for-off-nominal-behavior.html

[2] T. Kurtoglu and I. Y. Tumer, "A graph-based fault identification and propagation framework for functional design of complex systems," *Journal of Mechanical Design*, vol. 130, no. 5, p. 051401, 2008.

[3] M. Bozzano, A. Cimatti, J. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri, "The compass approach: Correctness, modelling and performability of aerospace systems," in *SAFECOMP*, 2009, pp. 173–186.

[4] T. Noll, "Safety, dependability and performance analysis of aerospace systems," in *FTSCS 2015*, 2015, pp. 17–31.

[5] J. Day, K. Donahue, M. D. Ingham, A. Kadesch, A. Kennedy, and E. Post, "Modeling off-nominal behavior in SysML." in *AIAA Infotech*, 2012, pp. 19–21.

[6] J. D. Andrews and L. M. Ridley, "Application of the cause–consequence diagram method to static systems," *Reliability Engineering & System Safety*, vol. 75, no. 1, pp. 47–58, 2002.

[7] D. Aceituna and H. Do, "Exposing the susceptibility of off-nominal behaviors in reactive system requirements," in *RE 2015*, Aug 2015, pp. 136–145.

[8] K. Madala, H. Do, and D. Aceituna, "A combinatorial approach for exposing off-nominal behaviors," in *ICSE 2018*, 2018.

[9] R. Agarwal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. of the 20th VLDB Conference*, 1994, pp. 487–499.

[10] M. Jackson, "Digitalhome software requirements specification," 2010. [Online]. Available: http://fmt.isti.cnr.it/nlreqdataset/

[11] J. Wang, W. Dong, and Z. Qi, "Slicing hierarchical automata for model checking UML statecharts," *LNCS*, vol. 2495, pp. 435–446, 2002.

[12] K. Lano, "Slicing of UML state machines," in *WSEAS AIC'09*, 2009, pp. 63–69.

[13] S. Teng and S. Ho, "Failure mode and effects analysis: an integrated approach for product design and process control," *IJQRM*, vol. 13, no. 5, pp. 8–26, 1996.

[14] D. L. Iverson, "Inductive system health monitoring," in *IC-AI04*, 2004, pp. 605–611.

[15] E. So, J. Ajtum, Y. Moy, and Y. L. Quach, "Requirements specification AB mail robot," 2005. [Online]. Available: http://www.ecs.umass.edu/ece/sdp/sdp05/preston/sdp{_}data/Requirement{\%}20Specification.doc