A Systematic Approach to Transforming System Requirements into Model Checking Specifications

Daniel Aceituna
DISTek Integration, Inc.
North Dakota State U.
Fargo, ND, USA
daniel.aceituna@ndsu.edu

Hyunsook Do
North Dakota State U.
Fargo, ND, USA
hyunsook.do@ndsu.edu

Sudarshan Srinivasan
North Dakota State U.
Fargo, ND, USA
sudarshan.srinivasan@ndsu.edu

ABSTRACT

We propose a method that addresses the following dilemma: model checking can formally expose off-nominal behaviors and unintended scenarios in the requirements of concurrent reactive systems. Requirements engineers and non-technical stakeholders who are the system domain experts can greatly benefit from jointly using model checking during the elicitation, analysis, and verification of system requirements. However, model checking is formal verification and many requirements engineers and domain experts typically lack the knowledge and training needed to apply model checking to the formal verification of requirements. To get full advantages of model checking and domain experts' knowledge in verifying the system, we proposed a front end framework to model checking and evaluated our approach using a real world application.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing & Debugging—*testing tools*

General Terms

Experimentation, Measurement, Verification

Keywords

Requirements verification, off-nominal behaviors, Causal Component Model

1. INTRODUCTION

Stakeholders and requirements engineers often specify requirements on the assumption that all user interactions with the system are nominal. However, this assumption can be erroneous since users do not always interact with a system in the normal manner intended by the designers of that system. Users have been known to exhibit behaviors that are off-nominal and not anticipated by the designers. Such off-nominal behaviors can result in a potential safety-hazard, if not accounted for in a set of requirements.

According to a survey performed by Boehm and Basili [6, 5], approximately 80% of avoidable rework comes from 20% of the defects. They further state that two major sources of rework that could have been avoided, come from requirements that are both hastily specified, and what they call "nominal-case design and development" [6]. A nominal-case requirement is one in which the requirement is specified on the assumption that scenarios external to the system will be "normal" and in accordance to expectations [25, 19]. Users interacting with a reactive system are inherently a source of non-deterministic, unpredictable actions [26]. The same can be said about an operational environment in which many sensors have been specified to handle stimuli external to the system [27, 32]. Users and environment can therefore be a large source of off-nominal scenarios, which can be a major problem in safety critical systems [25, 31, 5].

The lack of requirements that address off-nominal scenarios is considered a form of incompleteness in a set of requirements, and much research has been conducted in determining ways to verify whether a set of requirements is error-free and complete [28, 24, 1, 27]. There are two general classifications for requirements verification: formal and informal [18]. Formal verification typically involves a mathematical approach that strives to rigorously prove the correctness of a system's specification [21]. By contrast, informal verification is a non-rigorous approach, often consisting of the manual inspection of the requirements document [24].

A formal approach can help in proving the correctness of systems, but it is considered to be an expensive and difficult verification method due to its high application cost and learning curve [24]. Thus, a formal approach has been preferred for verifying critical systems because the consequences of system failure often involve high costs and can even be life threatening. One commonly used formal approach is model checking. Model checking can be used to expose off-nominal requirements, because of its ability to exhaustively check a system model for any combinations of user actions that result in an undesired system behavior [13]. Requirements engineers and non-technical stakeholders who are the system domain experts can greatly benefit from jointly using model checking during the elicitation and analysis of the system requirements [22, 29].

Model checking, however, requires specialized training, both in the development of model checking scripts and in the interpretation of the model checker results [9]. Thus, to get full advantages of model checking and domain experts' knowledge in verifying the system, it is necessary to alleviate the burdens of learning complex model checking techniques for requirements engineers and other non-technical stakeholders. There have been some attempts to make model checking accessible to those who are not trained in formal methods. These include the translation of a formal specification language called RSML [9] into a model checking script, and a set of tools collectively called the Software Cost Reduction (SCR), which can capture, analyze, and verify a set of requirements [8].

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591183

However, these two approaches still require a considerable degree of user training and a knowledge of temporal logic because they do not eliminate the need for translating natural language requirements into their specialized language before applying model checking. To address this problem, we propose a front end framework to model checking that builds a model that exhibits the system behavior, automatically creates the temporal logic properties, helps interpret counterexamples, and provides a user-centric interface that facilitates ease of use. To evaluate our approach, we applied it to a real-world embedded system, a skid loader, focusing on its safety properties, and the results showed that the proposed approach was able to expose undesired system behaviors that would cause a safety hazard.

Overall, this work makes the following novel contributions:

- A key contribution stems from the way in which a set of requirements is viewed. In our case, we view a set of requirements as describing a system of components that interact with one another in a causal relationship. Because we are dealing with requirements, we can model the components interaction at a high level of abstraction, without having to show a lot of detail. This enables us to focus on *WHAT* the system is supposed to do instead of *HOW* it is supposed to do it. We call our modeling scheme: the Causal Component Model (CCM).
- By using the CCM we can model concurrent reactive system requirements, and provide an easy-to-use interface to a formal verification. This enables non-technical domain experts and requirement engineers to directly participate in a formal requirements verification process during requirements elicitation.
- By providing an algorithm that translates a CCM to a model checking script and providing a mechanism that generates temporal logic properties for a model, we enable the use of model checking by requirement engineers and stakeholders who are not trained in formal methods and/or temporal logic.
- By providing a mechanism to simulate and interpret the model checker's counter-example using the CCM, and to resolve a defect by giving feedback to the CCM, we make the requirements verification process easy to apply without attaining formal verification training.

Section 2 presents background information and previous work relevant to the proposed approach. Section 3 describes our approach in detail. Section 4 presents a case study, and discusses our results their implications. Finally, Section 5 presents conclusions and discusses possible future work.

2. BACKGROUND AND RELATED WORK

In this section, we provide background information on model checking to show how our approach fits into the front end of model checking and discuss existing work relevant to model checking with requirements.

2.1 Model Checking

When model checking is applied to a set of system requirements, the requirements must be first translated into a type of finite state model (FSM) and FSM's correctness is then verified using properties written in temporal logic [12]. Typically, an FSM is is written in a Kripke structure, which can represent the behavior of a concurrent system as a graph with each node representing a snapshot of all the system's concurrent states at any given time. This snapshot is typically called a system state. Once a FSM is generated, the model checking process is performed by it, visiting one system state at a time, and determining whether some or all traversals satisfy a set of desired temporal logic properties [12, 13].

The model checking process is defined as follows. Let M be the FSM representing a concurrent finite-state system, and F be a temporal logic formula that expresses a desired property (specification) that M must satisfy. The model checking problem is to determine all the system state traversals that satisfy F. For example, a temporal specification may specify that a given system state must eventually be reached. The model checking algorithm traverses every path in the FSM and determines whether that system state is eventually reached, satisfying the temporal property. If a property is satisfied, the model checker returns true, if not satisfied, the model checker provides a counter example (a path that shows why the property was not satisfied).

Model checking has successfully been used to find previously undetected errors in hardware implementations, such as the IEEE Futurebus+ cache coherence protocol [14] and the IEEE scalable coherent interface [30]. It has also been used in protocol verification. One of the primary problems associated with model checking is the state explosion problem, in which the number of possible system states grows exponentially with each addition system element and the number of states it can produce [13]. Subsequently, there are various forms of model checking, each trying to manage the problem of state explosion, such as partial order reduction [33], bounded model checking [4], and symbolic model checking [11].

Among these approaches, symbolic model checking that uses a symbolic model verifier (SMV) has been a de facto model checking approach [11]. A SMV represents the Kripke structure implicitly using a propositional logic formulas, and has been able to model thousands of states [13], without having to construct a Kripke structure, with potentially thousands of states, explicitly. A SMV will also reduce the number of states in the model, using a binary decision diagram. In a SMV, the FSM is described using a scripting language.

Our proposed method uses a finite state model named Causal Component Model (CCM), which is relatively easier to construct and interpret than defining a model and specifications for a model checker. We use CCM as a front end to a symbolic model checking, specifically, the new symbolic model verifier (NuSMV) [11]. In the NuSMV model checker, the model and properties are specified using the NuSMV scripting language. We chose NuSMV, in part, because the IF-THEN structure of the scripting language maps directly with the casual component model (CCM), that we developed as a stakeholder friendly front end to the model checker.

2.2 Related Work

In the area of model checking with requirements, there has been research demonstrating that model checking can indeed be effective in verifying safety properties in event driven system requirements [3, 29], and in large scale requirements, such as the traffic alert collision avoidance system [7]. There has been an effort to apply model checking to early requirements specification, as in the development of a formal modeling language called Tropos [20]. Tropos was developed by extending the i* modeling language [34], which can be used to model the system within the context of its operating environment, and gain understanding about the problem domain, which occurs early in the elicitation of requirements. Other studies have shown the possibility of model checking a complete set of Software Cost Reduction requirements specifications containing various data types and invariants [22]. Another study addressed systems with large input domains, such as user interface and input sensors [10].

There have been other attempts to make model checking accessible to those who are not trained in formal methods. These include the translation of an formal specification language called RSML [9] into a SMV model checking script, and a set of tools collectively called the Software Cost Reduction (SCR) [8]. RSML uses a modeling language similar in syntax to SMV, therefore the learning curve for RSML is similar to learning SMV. SCR is a set of eight tools originally developed as a means to capture, analyze, verify, and document requirements. There is presently a SCR tool to act as a front-end to a SMV model checker. SCR uses a tabular notation that requires a substantially large learning curve.

Our approach eliminates the need to learn a special language or notations that are required by RSML and SCR. Also, both RSML and SCR requires that the user have training in developing the temporal logic properties, whereas our approach generates the properties automatically (Section 3.7). Further, both RSML and SCR require the user to interpret any counter-examples generated by the model checker, whereas our approach can enable non-technical stakeholders to create a NuSMV script using CCM, as well as interpret the counter-examples using the same CCM. The aforementioned research by others showed that model checking is both feasible, and valuable in the area of requirements, but most did not focus on making the application into the requirements arena user friendly, which we feel is important in promoting model checking's proven value into industry.

3. PROPOSED APPROACH

This section presents the details about our proposed approach, Causal Component Model (CCM), which was evolved from NL-toSTD [2, 1] through multiple controlled experiments.

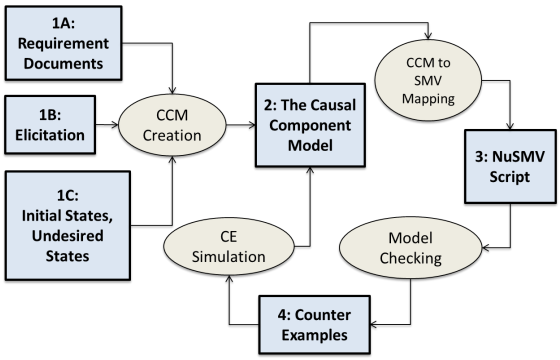

Figure 1: Overview of the proposed approach

3.1 Approach Overview

Figure 1 summarizes our approach. The CCM is created by reading system artifacts that are gathered from an existing document (*1A*) or elicitation (*1B*). The CCM can be displayed to the user to provide feedback during the elicitation process. As the CCM is created, a CCM notation is also created. This notation is designed to map to the model checker's scripting language, via a mapping algorithm implemented by the support tool. The CCM is represented in the NuSMV script (3), along with temporal logic properties, most of which are generated by the tool. Some of the information for the temporal properties are user-defined (*1C*), such as the desired initial states, and the unsafe system states that could result from an off-nominal scenario . At any time during the elicitation, the CCM can be model checked. Any properties that fail to hold in the CCM will produce a counter example (4), which is then enacted by the support tool in the form of an animated CCM (2).

The animated CCM becomes a simulation of the counter-example. This is done as a means to help the stakeholders better interpret the counter-example. Steps 1 to 4 can be executed in a continuous fashion, providing valuable and timely feedback during the elicitation process.

3.2 The Causal Component Model (CCM)

The CCM requires that we view a set of requirements as a collection of statements that describe how various system components interrelate with one another on a causation basis [1]. The CCM consists of a set of interrelated state transition systems with each transition system modeling the behavior of a specific system component. Each component's behavior occurs concurrently with the other components, and the CCM models how one component's behavior affects another component's behavior. We first formally define the CCM, and then provide a real world example.

3.3 Formal Definition of CCM

We define a state transition system as a tuple $\{S, R\}$ where S is a set of states and $R \subseteq SXS$ is a set of binary relations over S, known as transitions. For example, for $(s_1, s_2) \in S$, we say that there is a transition from s_1 to s_2 expressed as $(s_1 \rightarrow s_2)$. The CCM consists of a set of interrelated state transition systems with the following properties. We define C as the set of all components {a, b, c, d, . . .} in the system.

Our modeling approach uses system components as the smallest level of granularity, because it facilitates the partitioning of a system. For example, the desired system may consist of the set of components {*motor*, *switch*, *temp_sensor*}. A component can assume one or more states, so we define a component-state as_i as the state s_i that component a can assume. A given component a ($a \in C$) has a set of component states, *Sa*, where *Sa* = $\{as_1, as_2, as_3, as_4\}$. A given component can only transit between its own component states $as_1 \rightarrow as_2$. The CCM is designed to model synchronous systems. Thus, transitions between two given component-states occur on a given clock cycle as allowed by a transition guard, *tg*. The relationship between *tg* and a transition $(a_1 \rightarrow a_2)$ is defined using a mapping notation $tg : as_1 \rightarrow as_2$. This reads, as_1 is allowed to transit to as_2 if *tg* is true. In other words, *tg* can be viewed as the cause of the transition.

A transition guard *tg* is a boolean function in a sum of product form where every literal is in normal form. The product terms consist of component states $(as_i \wedge bsi \wedge cs_i ...)$ where no product term has two or more component-states from the same component. This is because a given component cannot be in two or more states at a given instance of time. For example, the following expression $as_1 \wedge bs_2 : cs_2 \rightarrow cs_3$ reads cs_2 is allowed to transit to cs_3 if system component a is in state s_1 AND system component b is in state s_2. Note that expressions can be as simple as $as_1 : cs_2 \rightarrow cs_3$, in which case the transition is allowed solely by as_1, and we do not care what state the other systems components are in. The expressions can be as complex as $(as_1 \wedge bs_2) \vee (as_3 \wedge bs_3) : cs_2 \rightarrow cs_3$, in which a combination of alternative component states for a and b is allowed, providing they are in separate product terms.

3.4 Example of a CCM

Figure 2 shows the CCM of a real world example: a simple motor controller. From Figure 2, we make the following observations. A CCM consists of interrelated state transition diagrams (STD), with each STD modeling the behavior of a system component. The components in this case are *motor*, *switch*, and *temp_sensor*. Note that each STD models how that component will transit between its various component-states, designed as component(state). For example,

switch(on) can transit to *switch(off)*. The STD's are interrelated by virtue of the fact that transitions in one STD is determined by the state of one or more of the other STD's in the system. The CCM is designed to model a synchronous system, where each component may or may not change states concurrently, on a given clock cycle. Thus, whether a component actually changes states, depends on the state of the other components, at the time of that clock cycle.

For example, in Figure 2, the motor will transit from *motor(on)* to *motor(off)* if and only if the switch is on *(switch(on))* OR if the *temp_sensor* is sensing a safe operating temperature (*temp_sensor (safe)*). Note that the USER's interaction with the system is also captured; it is considered an external component-state. By convention external component-states are capitalized. Other external component-states can include the ambient temperature of the system's operating environment, in this case written: *TEMP(>100)*, and *TEMP(<80)*. Overall the CCM is component-state dependent, and provides a high-level view of the system. To construct a CCM the stakeholders only need to specify the state-dependent behavior of the system components, and how those behaviors interrelated. This keeps the system specification focused on "what" the system is supposed to do, without straying into "how" the system is supposed to achieve its behavior. The "how" is left up to the design phase, as it should be. We propose that this high level view is sufficient to expose incompleteness stemming from off-nominal behaviors, while being easy to conceive by non-technical stakeholders who should not be distracted by the details of the system's design.

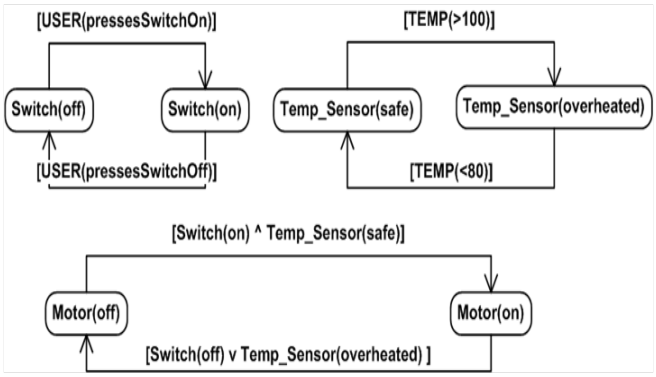

Figure 2: An example of Causal Component Model

To elicit a set of requirements directly into a CCM, a user is required to follow a four step process, where each step involves the elicitation of one of the model's artifacts: C, *C(S)*, and Tg. As the artifacts are gathered, the model is automatically constructed by our tool (CCMChecker – see Section 4) that supports the proposed approach. The users can view the model at any time because CCMChecker can construct and display a graphical representation of the CCM, upon request. The four steps are:

- **Step 1: Elicit a set of system components**, as expressed in the requirements. A component is something in the system that can change states and/or cause a change in state.
- **Step 2: Elicit a set of component(state)s** for each component as expressed in the requirements. A component(state) is the state of a component at a given instance of time.
- **Step 3: Elicit a set of inter-component transitions**, forming a state transition diagram (STD) for each component as expressed in the requirements. Inter-component transitions relate two given component(state)s to one another ($as_1 \rightarrow as_2$).
- **Step 4: Elicit a set of causal relationships.** To model an entire system, we must capture the way the various components interact with one another. Causal relationships allow for the modeling of interactions between concurrent behaviors in a system.

To provide the means of algorithmically converting a CCM into a NuSMV script, we have developed a CCM notation. This notation serves as a textual representation of the CCM. The following two sections explains the formal derivation of this notation, and how it maps to, and is translated into a NuSMV script.

3.5 The Causal Component Model Notation

To develop the CCM notation, we built upon the function notation that specifies the mapping of elements between two domains, commonly expressed as $F : X \rightarrow Y$. The reason for doing so is because the CCM notation must be able to represent a mapping between the component(state)s (*C(s)*). Furthermore, this mapping must be dependent on some transition (function) F. On the left-hand side of Figure 3, the function notion, $F : X \rightarrow Y$, specifies that function F maps element "a" from domain X to element "b" in co-domain Y.

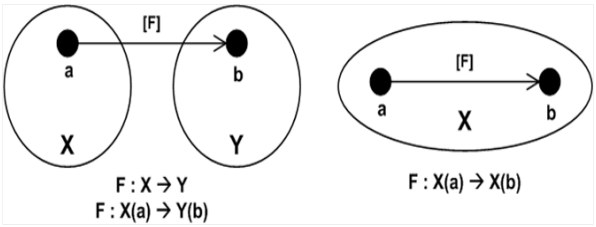

Figure 3: The CCM formal notation is based on function notation

We use the notations *X(a)* and *Y(b)* to denote that "a" and "b" are within components X and Y respectively. For our purposes, we will now assume that $X = Y$ and restrict the function F to within one component. The reason for doing so is because we want the CCM notation to map transitions within the same component, and not between two separate components. Thus, on the right-hand side of Figure 3, we redefine the function F as a transition that maps "a" to "b" within component X. We replace X with C, "a" with $S1$, and "b" with $S2$, which represents a component containing states $S1$ and $S2$, and replace F with inter-component transition T; this gives us $T : C(S1) \rightarrow C(S2)$.

In a real world example, a component would likely have more than two states. These various states would be interconnected via a set of transition functions. The left hand side of Figure 4 shows an example of a component $C1$ that has three states ($S1$, $S2$, and $S3$), with three transitions ($T1$, $T2$, and $T3$). Since the CCM is geared toward embedded systems it is important that we be able to model concurrency. To do so, we need the ability to simultaneously represent more than one component, and their transitioning states.

Figure 4 shows two component ($C1$ and $C2$), and their corresponding states. In Figure 4, each component has their states interconnected in a state transition diagram (STD). Each STD can act independently and thus concurrently with the other components. Components $C1$ and $C2$ of Figure 4, each has three transitions, labeled $T1$, $T2$, and $T3$. A transition can be a single component(state) or a logical combination of component(state)s. By using component(state)s as part of an transition we establish the casual relationship between the component(state)s of two components within the system boundary.

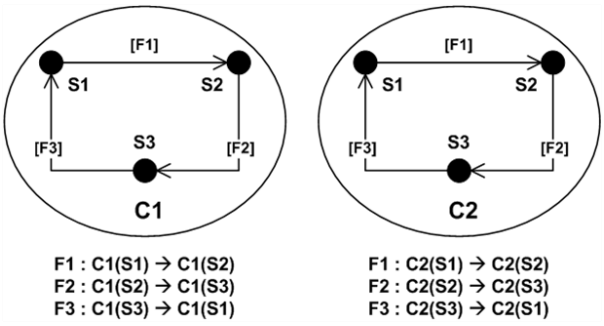

Figure 4: Three transitions within a component ($C1$). $C2$ is a second component with concurrent behaviors

A transition can also be natural physical properties and/or external stimuli from the system's operational environment. By convention, external stimuli in transitions are written in uppercase. In Figure 2, an example of an external stimuli in a transition is *USER(pressesSwitchOn)*. Finally, in a synchronous system, transitions occur on the next system clock Tc cycle. This means that while not always stated, a transition is actually *ANDed* with with Tc.

3.6 Translating a CCM into a NuSMV Script

Recall the CCM is used as a stakeholder friendly front-end to a NuSMV model checker. This requires the CCM's translation into the script notation used in NuSMV. To achieve this we use a mapping algorithm that leverages the fact that both CCM and NuSMV notations have an IF-THEN structure. Another similarity is that both CCM and NuSMV notations represents components, present states, next states and transitions. Thus, both share similar artifacts and semantics, while expressing those semantics using different syntax. Figure 5 shows a typical NuSMV script. Note that Figure 5 is a partial listing of the real world example of Section 4; some of the *bucketBoom* behaviors are not included, as indicated by ". . .".

```
MODULE main --designates the start of a system module.
VAR --start of the variable definition section.
   bucketBoom : {movDown, movUp, notMov};
   seatBar: {down, up};
   driver : {tiltLPBack, tiltLPForw, lpDefault, raiseSeatBar,
   lowSeatBar};
ASSIGN --section where a component's behavior is defined.
     init(seatBar) := up; --init keyword sets initial states.
     init(bucketBoom) := notMov;
ASSIGN --section where a component's behavior is defined.
     next(seatBar) := case -- Defines component behavior
          seatBar = up & driver = lowSeatBar : down;
          seatBar = down & driver = raiseSeatBar : up;
          1:seatBar;
          esac; --end of ASSIGN section
ASSIGN
      next(bucketBoom) := case
          bucketBoom = movDown & (seatBar = down & driver =
          tiltLPForw) : movUp;
      next(bucketBoom) := case . . .
SPEC -- section where one temporal logic property is
        expressed.
    !EF(seatBar = up & bucketBoom = movUp)
```

Figure 5: An example of partial NuSMV script.

Note that the ASSIGN sections of Figure 5 use an IF-THEN structure to describe a component's behavior. For example, The statement, $seatBar = up\&driver = lowSeatBar : down$, reads IF *seatBar* is *up* and the *driver* lowers the *seatBar*, THEN the *seatBar* will transit to a *down* state. Expressed in CCM notation, the same behavior is written as: $driver(lowSeatBar) : seatBar(up) \rightarrow seatBar(down)$.

In a general form, $T : C(S1) \rightarrow C(S2)$ syntactically maps to: $C = S_1\&T : S_2$. Transition (T) can be a disjunctive and/or conjunctive expression of component states. In general terms, given a set of CCM expressions, the form is shown in line (1):

(1) $C_n(S_2) : C_p(S_1) \rightarrow C_p(S_2)$

The algorithm used to create the ASSIGN sections iterates through each CCM expression, performing the following steps:
Convert $C_p(S_1)$ to $C_p = (S_1)$
Convert $C_p(S_2)$ to : S_2;
Convert $C_n(S_2)$ to $C_n = S_2$
Convert $C_p(S_1)$ to $nextC_p$
Convert $C_p(S_1)$ to $1 : C_p$
Concatenate $ASSIGN$, $nextC_p$, $:= case$, $C_p = (S_1)$, &, $C_n = S_2$, : S_2;, 1:C_p, and $esac$;

The concatenated terms form the ASSIGN section (2):

(2) ASSIGN
next(C_p) := case
$C_p = S_1\&C_n = S_2 : S_2$;
1:C_p;
esac;

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

(3) $C_n(S_2) : C_m(S_1) \rightarrow C_m(S_2)$
(4) $C_p(S_1) : C_n(S_1) \rightarrow C_n(S_2)$
(5) VAR
$C_m : \{S_1, S_2\}$;
$C_n : \{S_1, S_2\}$;

3.7 Creating the NuSMV Temporal Properties

A CCM is automatically converted into the model portion of the NuSMV script. However, to complete the NuSMV script, we still need to specify the temporal logic properties that will be used to expose any off-nominal scenarios that may be produced in the CCM. While the creation of temporal logic properties is traditionally the most knowledge intensive endeavor in model checking, we will facilitate the automatic creation of the properties by leveraging two areas of research that has been conducted in the pass. The first is the development of temporal logic patterns as a means to simplify the specification process [16, 23]. The second area of research, which directly relates to the first area, is the fact that the most useful temporal properties can be achieved with a handful of patterns [16, 17].

For our purposes, we have leveraged this prior research to achieve the desired defect coverage with the following four temporal logic patterns (written in the NuSMV script language).

A: $AG(!(C_n = S_1) \rightarrow EF(C_n = S_1))$
B: $AG((C_n = S_1) \rightarrow EF(!(C_n = S_1)))$
C: $AG(!(C_n = c(s)1\ \&\ C_m = c(s)2) \rightarrow EF(C_n = c(s)1\ \&\ C_m = c(s)2))$
D: $!EF(C_n = c(s)1\ \&\ C_m = c(s)1)$

In each pattern, C_n represents component, and S_n is that component's state. For example, $C_1 = S_1$ means State number 1 of component number 1. Patterns A and B are used to expose incompleteness in the CCM, which would translate into an undesirable behavior in the system. Pattern A verifies whether a given component(state) can be entered. This is a measure of the state's

in-degree, of which there should be at least one in-degree per component(state). Pattern B verifies whether a given component(state) can be exited. This is a measure of the state's out-degree, of which there should be at least one per component(state). Together, patterns A and B verify whether a round-trip traversal is possible in a given component's STD. Reactive systems typically do not terminate, while running, and are in a given state at any given time. Thus, every component(state) in the CCM should have at least one in-degree and one out-degree. Patterns A and B are generated automatically for each component(state) by the tool.

Pattern C also verifies round-trip traversal but on a system level. In this case, the question is whether the initial system state can be reentered. The component(states) that comprise the initial system state is selected by the user, while the temporal property is created by the tool. Pattern D, is the strongest property used to exposed off-nominal behavior. It determines whether an undesired system state is eventually reachable or not. The component(states) that comprise the undesired system state is selected by the user, while the temporal property is created by the tool. Referring back to listing one, we see one example of a temporal logic property, in the SPEC (last line) section of the script: $!EF(seatBar = up \,\&\, bucketBoom = movUp)$. Note that this property uses pattern D, which determines whether an unsafe system state is eventually reachable or not. To create the temporal properties the support tool uses a mapping algorithm similar to the one used for the script model.

We now turn to a case study, derived from the design of a real world skid loader presently available on the market. The Skid Loader's manufacturer will be kept confidential.

4. A CASE STUDY: A REAL WORLD APPLICATION

To support our approach, we implemented a support tool, CCMChecker, which is written in C-Sharp and uses a NuSMV model checker as a back end. Using CCMChecker, we conducted a study using a real world embedded system, a commercial skid loader to evaluate our approach. In particular, we focused on the the method's ability to expose off-nominal behaviors of the system, which is the major concern for safety critical systems because off-nominal behaviors that are not handled properly can result in safety-related accidents.

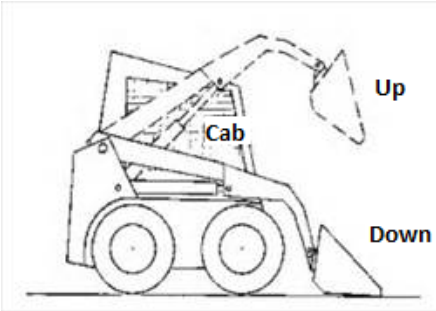

Figure 6: A typical skid loader with the load bucket up (dashed lines) and down (solid line).

The safety function in question is the automatic disabling of the skid loader *bucketBoom* when the safety *seatBar* is lifted. A skid loader, shown in Figure 6, is typically usable by anyone without special training or certification. Thus, it can potentially be misused.

The manufacturer's objective is to make it as fool-proof as possible. In a typical skid loader design, the *bucketBoom* moves up and down directly in front of the cab, creating a potential problem for someone not confined within the cab during the bucket's use.

Figure 6 shows the relative proximity of the bucket's up/down movement and the operator cab. For safety reasons, the driver should be confined within the skid loader cab during the bucket's movement. So, when the driver climbs into the cab, there is a *seatBar* that comes down over the driver, similar to the passenger restrain used in roller-coaster. For our case, we do not want the *bucketBoom* to move up or down if the *seatBar* is up. A *seatBar* that has been raised could lead to the driver protruding part of their body outside the cab, within the bucket's envelope of movement. This would create a potential safety hazard. We analyzed a total of 76 requirements focusing on those requirements that pertain to the controlling of the bucket. Of the 76 requirements we found an issue that revolved around four requirements; we focus our case study on those four requirements. The portion of the requirements [1] that pertains to that feature is:

- REQ1: The bucket boom shall move upward when the left pedal is tilt forward with the driver's toe.
- REQ2: The bucket boom shall move downward when the left pedal is tilt back with the driver's heel.
- REQ3: The bucket boom shall stop moving when the left pedal is allowed in the default position by the driver's foot.
- REQ4: All bucket boom operations are allowed only if the seat bar is lowered.

For confidentiality reasons, we have paraphrased the wording of the original requirements that were written for the embedded system company while maintaining the intended meaning. Note that in the initial set of requirements, REQ4 does state that "All bucket boom operations are allowed only if the seat bar is lowered." This would appear to address the desired safety requirement. However, to the stakeholders stating REQ4, there may be certain assumptions based on what they perceive as the driver's normal behavior. For example, a typical normal behavior would be that the driver climbs into the cab, lowers the *seatBar*, and then operates the bucket. Upon existing, the driver would then stop operating the bucket (placing it in a down position), and then raise the *seatBar* when he is departing the cab.

The following subsections describe how our approach is applied to the skid loader's requirements and detects possible safety-critical defects.

4.1 Creating the CCM

To create CCM, and subsequent NuSMV script, a user must enact the four steps described in section 3.4 We explain how these steps are applied to the skid loader's requirements.

Step 1: Elicit a set of system Components (C), as expressed in the requirements.

From the set of requirements above, we can readily identify three components: *bucketBoom*, *seatBar*, and *leftPedal*, which we have abbreviated. Note that, in this case, the three components are quickly identified because they are system artifacts that are either controlled by the system, or provide an interface to the driver. Three components will produce a CCM with three STDs.

One advantage to our method is the ability to model the interactions between the components we wish to focus on, without having to model the entire system; this results in a compositional approach [15]. In this case, we are focusing on the interaction between the *bucketBoom* and the *seatBar*, so we can either model the

[1]Note that the skid loader's requirements have been generalized in order to preserve manufacturer's confidentiality.

leftPedal as a third STD or treat it as part of a transition. Having the option to abstract additional STDs as part of a transition helps manage the complexity and size of the resulting CCM. For our purposes, we will only model the *bucketBoom* and the *seatBar*.

Step 2: Elicit a set of Component(States)) as expressed in the requirements.

Next, CCM creation requires that we assess what possible states the two components will be in at various times during the desired system operation. This gives us the following list of component states: *seatBar(up)*, *seatBar(down)*, *bucketBoom(notMoving)*, *bucketBoom (movingUpward)*, and *bucketBoom(movingDownward)*.

Step 3: Elicit a set of inter-component Transitions (T), forming a State Transition Diagram (STD) for each component as expressed in the requirements.

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

T1: $seatBar(down) \rightarrow seatBar(up)$
T2: $seatBar(up) \rightarrow seatBar(down)$

We can do the same with the bucketBoom, which three states and six possible transitions ($T3 \rightarrow T8$) between them:

T3: $bucketBoom(notMoving) \rightarrow bucketBoom(movingUpward)$
T4: $bucketBoom(movingUpward) \rightarrow bucketBoom(notMoving)$
T5: $bucketBoom(notMoving) \rightarrow bucketBoom(movingDownward)$
T6: $bucketBoom(movingDownward) \rightarrow bucketBoom(notMoving)$
T7: $bucketBoom(movingDownward) \rightarrow bucketBoom(movingUpward)$
T8: $bucketBoom(movingUpward) \rightarrow bucketBoom(movingDownward)$

Step 4: Elicit a set of Causal Relationships.

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

- $DRIVER(raisesSeatBar) : seatBar(down) \rightarrow seatBar(up)$
- $DRIVER(lowersSeatBar) : seatBar(up) \rightarrow seatBar(down)$

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

- $DRIVER(tiltLPForward) \wedge seatBar(down) : bucketBoom(notMoving) \rightarrow bucketBoom(movingUpward)$
- $DRIVER(LPDefault) : bucketBoom(movingUpward) \rightarrow bucketBoom(notMoving)$
- $DRIVER(tiltLPBack) \wedge seatBar(down) : bucketBoom(notMoving) \rightarrow bucketBoom(movingDownward)$
- $DRIVER(LPDefault) : bucketBoom(movingDownward) \rightarrow bucketBoom(notMoving)$
- $DRIVER(tiltLPForward) \wedge seatBar(down) : bucketBoom(movingDownward) \rightarrow bucketBoom(movingUpward)$
- $DRIVER(tiltLPBack) \wedge seatBar(down) : bucketBoom(movingUpward) \rightarrow bucketBoom(movingDownward)$

Figure 7 shows the CCM model derived from the requirements by following all the steps described in Section 3. Note that the transitions for the *bucketBoom* incorporate the behavior of the *DRIVER* as well as that of the *seatBar*. The *DRIVER* is written in uppercase because it is considered a transition from the system's operating environment; a cause that originates external to the system. As to the *seatBar*'s transitions, they are solely external, since their direct cause is the *DRIVER*. We point this out because the CCM we have specified is heavily affected by external actions to the system.

The *DRIVER* is a direct cause of behavior in both the *bucketBoom* and the *seatBar*. This means that there are potentially many ways in which the *DRIVER* can cause an unexpected (off-nominal) behavior in the system. With the nominal behavior in mind, the most intuitive way to model REQ4 (Figure 7) would be by *ANDing* the *seatBar* component state with the *bucketBoom*, which enables the *bucketBoom* to move only when the *seatBar* is down. However, the real question is whether there is off-nominal behavior by the driver that would defeat the desired safety feature. In other words, can the *DRIVER* somehow create a scenario in which the *bucketBoom* is moving while the *seatBar* is up?

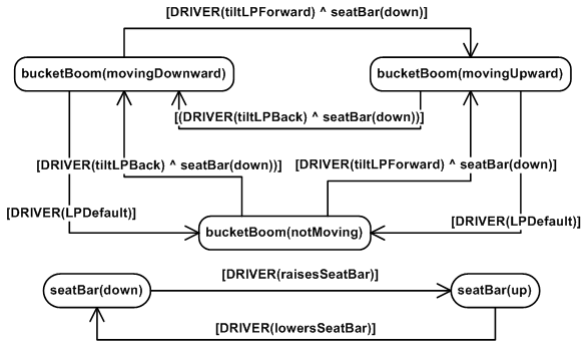

Figure 7: The partial CCM of the Skid Loader

4.2 Automatic Creation of the NuSMV Temporal Properties

The CCM of Figure 7, is automatically converted into the model portion of the NuSMV script as shown in Figure 5. The temporal logic properties are also generated, and placed in the NUSMV script. The followings are the properties generated for the skid loader example:

- $!EF(seatBar = up \& bucketBoom = movUp)$: This verifies that an undesired system state cannot occur. It is using pattern D as listed in Section 3.7
- $AG(!(seatBar = up \& bucketBoom = notMov) \rightarrow EF(seatBar = up \& bucketBoom = notMov))$: This verifies that you can always get back to initial states. It is using pattern C as listed in Section 3.7.

The following properties verify that there is at least one in-degree for each of the *seatBar* states. Similar in-degree properties using the same pattern below were created for the *bucketBoom* (not shown). They are using pattern A as listed in Section 3.7.

- $AG(!(seatBar = up) \rightarrow EF(seatBar = up))$
- $AG(!(seatBar = down) \rightarrow EF(seatBar = down))$

The following properties verify that there is at least one out-degree for each of the *seatBar* states. Similar out-degree properties using the same pattern below were created for the *bucketBoom* (not shown). They are using pattern B as listed in Section 3.7.

- $AG((seatBar = up) \rightarrow EF(!(seatBar = up)))$
- $AG((seatBar = down) \rightarrow EF(!(seatBar = down)))$

4.3 Model Checking Results

After CCMChecker combines the model and temporal properties into a NuSMV script, the script is executed by a NuSMV command line executable. CCMChecker then reads the standard out from the executable, parses out any and all counter-examples that represent a property that did not hold. The counter example can then be represented as either a traversal path consisting of only those states and transitions that lead up to the undesired state, or as a sequence of CCM simulation snapshots, where each snapshot highlights a system state. Both representations will be demonstrated later in this section. Figure 8 shows the results of running the skid loader NuSMV script through the model checker; only the single resulting counter example is shown, using a traversal path representation.

The requirements as stated will result in an undesired system state, that of the *seatBar* being up, while the *bucketBoom* is moving. Furthermore, the model checker was able to reveal this because when CCMChecker created the NuSMV script, it purposely did not assign any rules or initial values to the *DRIVER* component states. This allowed for an unconstrained enactment of the *DRIVER*'s behavior upon the model. Subsequently, all possible behavioral combinations by the *DRIVER* were simulated. This includes any off-nominal behaviors that stakeholders may have assume the driver would never enact under normal operation of the skid loader.

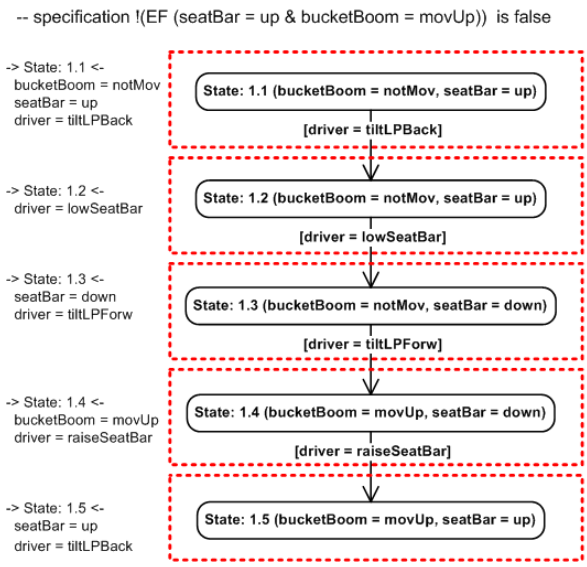

Figure 8: Counterexample from first run through Model Checker.

In this case, from the counter-example, we see that the undesired system state is reached when the driver decides to lift the *seatBar* while the *bucketBoom* is in motion; a trivial action on the driver's but one that was unaccounted for in the requirements. REQ4 says: "All bucket boom operations are allowed only if the *seatBar* is lowered." This requirement is satisfied in the case where the *bucketBoom* is not allowed to move until the *seatBar* is lowered, but no requirement covers the scenario where the *seatBar* is raised, once the *bucketBoom* is moving.

This is an example of a "trivial" yet potentially dangerous oversight on the part of stakeholders. In response to the exposed missing requirement, we could make the adjustment shown in Figure 9. Since the undesired state occurs as a result of the driver's unconstrained behavior with the *seatBar*, we have added a *seatBar* lock in order to constraint the *seatBar* and limit the results of driver's behavior. The addition of the *seatBar* lock represents one or more requirements that were missing and could have sooner or later been elicited or conceived in order to address the unconstrained behavior of the driver. If not caught during the requirements phase, this issue would have likely been caught down-stream during the system's development at a higher cost.

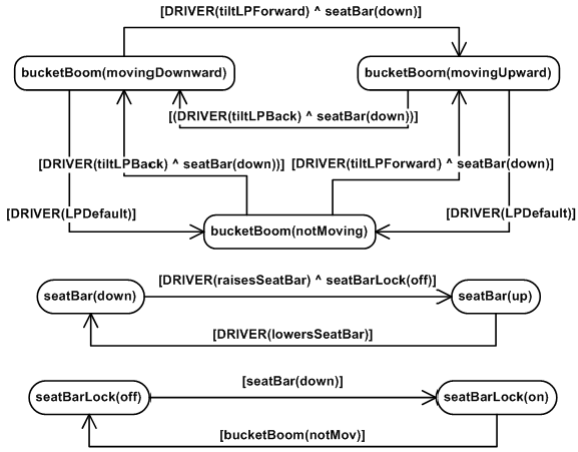

Figure 9: The modified CCM in response to the counter-example of Figure 8

However, using our approach, it would have likely been caught very quickly, as the requirements are elicited, since CCMChecker allows for continuous simulation of the requirements, on an incremental basis. The additional requirements describing the *seatBar* lock could be worded and added to the set of requirements at the stakeholder's discretion. Also, the wording can be taken directly from the CCM by explicitly describing the STD pertaining to the added *seatBar* lock.

For example, additional requirements could be expressed as "REQ5: If the *seatBarLock* is off, and the *seatBar* is pulled down, the seatBarLock shall activate," and "REQ6: If the seat bar is down, it can only be raised up, if the driver tries to raises the *seatbar* and the *seatBarLock* is off." Notice that the wording is taken from the modified CCM in Figure 9. This new CCM was automatically described in a new NuSMV script, run again in the NuSMV executable, and produced the result as shown in Figure 10.

The new counter example in Figure 10 shows that the addition of a *seatBarLock* requirement was not enough to avoid the undesired system state. To obtain a better understanding of how to address the remaining problem, we analyzed the new counter-example, using two sequence snapshots from the CCM simulations enacted by CCMChecker. Figure 11 shows the succession of two CCM snapshots leading up to the undesired state of the *seatBar* being up while the *bucketBoom* is moving.

The two snapshots in Figure 11 are created by CCMChecker, as it runs a simulation of the counter-example using the same CCM diagram generated by CCMChecker during elicitation. Generating a CCM-based simulation is one way that CCMChecker helps the stakeholders interpret the counter-example produced by the Model Checker. The two CCM snapshots in Figure 11 are labeled "State 1.5" (top) and "State 1.6" (bottom), which corresponds to the same two states of the counter example in Figure 10.

Starting from the top of Figure 11, the states with the bold line represent the states the three components are in during State 1.5. The transitions in bold will allow a state transition on the next clock cycle, resulting in State 1.6. In State 1.5 of Figure 11, the

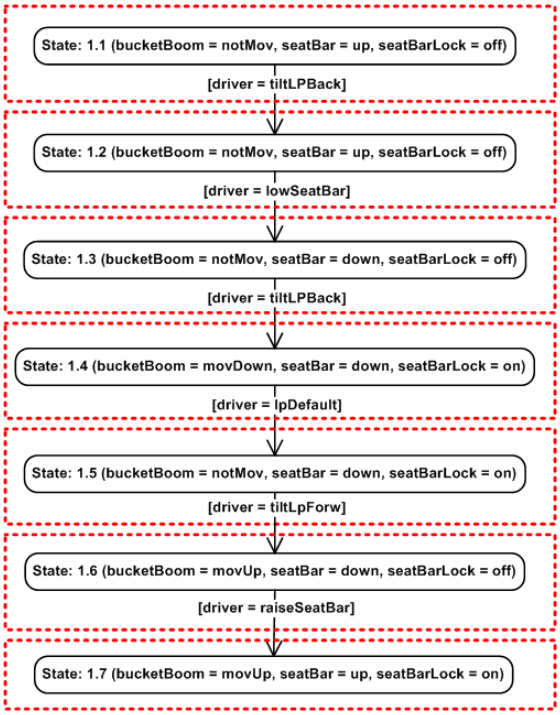

Figure 10: Counterexample from running the first corrected set of requirements.

system is poised for the *bucketBoom* to start moving upward, and the *seatBarLock* to unlock during the next clock cycle. With the *bucketBoom* moving and the *seatBar* unlock (State 1.6), there is nothing to prevent the unsafe condition if the driver decides to lift up the *seatBar*. Therefore, the only apparent solution is to further constraint how State 1.6 occurs, by enabling the *seatBar* to unlock solely on the basis of a non-moving *bucketBoom*.

By examining the transitions in bold in State 1.5 of Figure 11, the stakeholders might decide that it is a bad idea to allow the unlocking of the *seatBarLock* only when the *bucketBoom* is not moving. Therefore, they may try placing a further restriction on how the *seatBar* is unlocked. One additional restriction would be to logically AND the driver's attempt to raise the *seatBar* (*driver (raiseSeatBar)*), with *bucketBoom(notMov)*, which would be the only safe *bucketBoom* state for the driver to raise the *seatBar*. We see this modification in Figure 12.

While it may not be obvious that this additional restriction would finally solve the problem, the fact is that CCMChecker's model checker will take very little time to verify whether it is indeed a solution. As it turns out is, when running the model checker on the CCM in Figure 12, all properties hold (they all return true). The additional *seatBarLock* and associated restrictions, can be expressed as additional requirements at the stakeholders discretion.

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

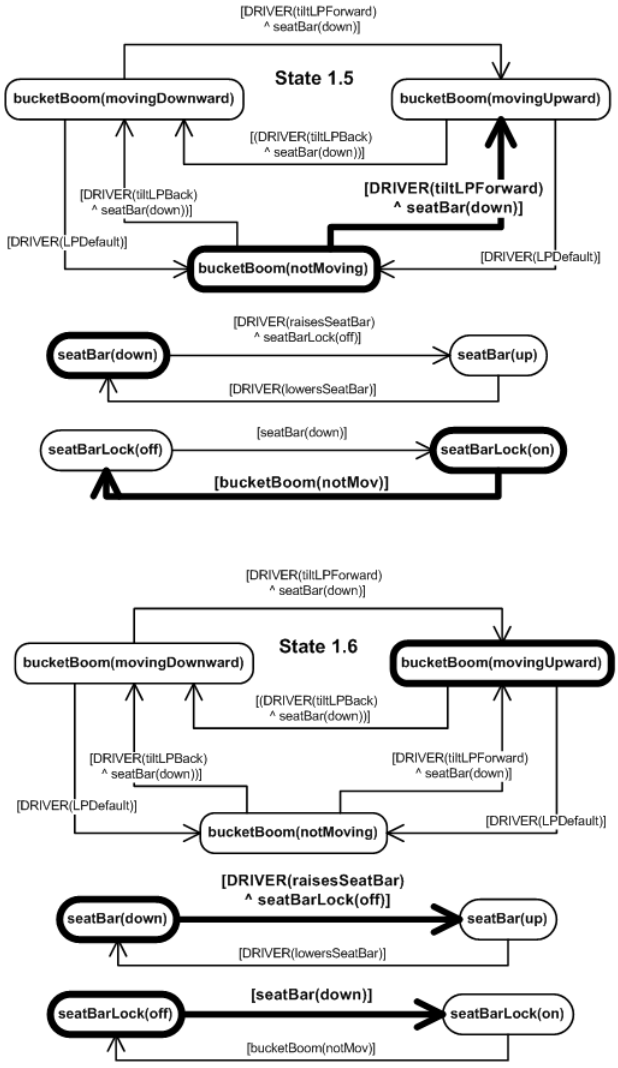

Figure 11: Two snapshots in CCM-based counterexample simulation.

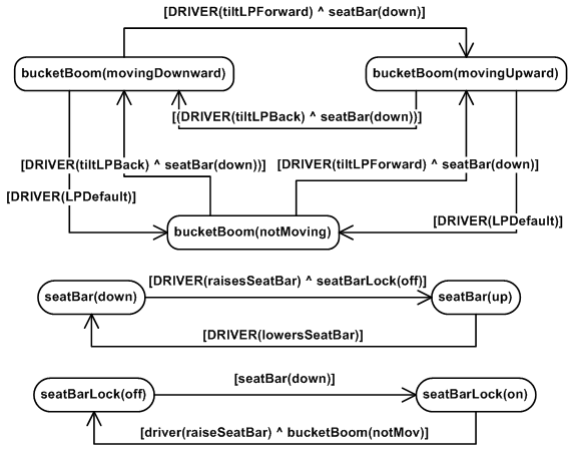

Figure 12: Additional restriction is placed on the unlocking of the seatBarLock: (driver(raiseSeatBar $\wedge$ bucketBoom(notMov))

5. CONCLUSIONS AND FUTURE WORK

We have proposed an approach to creating a model checking front-end that allows for direct stakeholder/domain expert interaction during the requirements phase. The approach involves using a requirements model we developed called the Casual Component Model (CCM). The CCM is designed so as to facilitate the systematic construction of a requirements model that can then be automatically translated into a NuSMV model checker script. The constructed CCM also provides the means to simulate any resulting counter example, to aid in the correction of the requirements. We applied the proposed method to a real world set of requirements, and showed how an off-nominal behavior can be exposed, analyzed, and addressed by the addition of one or more requirements. We are encouraged by the results and plan on applying the technique to a greater set of requirements, particularly where the system is intended to interact with a high percentage of external stimuli, up and beyond that of merely user interaction. This would include sensor data, and interrupts.

While our focus has been on exposing off-nominal behaviors, we think that other anomalies can be potentially exposed as well, such as inconsistency between concurrent processes, and possible deadlock and starvation scenarios. Some of the remaining questions pertaining to our approach, include the question of scalability. CCM's ability to focus on the interaction between a subset of a system's components allows for the analysis of subsystems. This in turn, enables us to divide and conquer the system in such a way that modeling a complete system may not be necessary in most cases. Partial modeling would keep the number of states manageable.

6. ACKNOWLEDGMENTS

This work was supported in part by NSF CAREER Award CCF-1149389 to North Dakota State University.

7. REFERENCES

[1] D. Aceituna, H. Do, and S. Lee. Interactive requirements validation for reactive systems through virtual requirements prototype. In *MoDRE*, pages 1–10, 2011.

[2] D. Aceituna, H. Do, G. Walia, and S. Lee. Evaluating the use of model-based requirements verification method: A feasibility study. In *EmpiRE*, pages 13–20, 2011.

[3] J. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *TSE*, 19:24–40, 1993.

[4] A. Bierel, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[5] B. Boehm. Anchoring the software process, 1995.

[6] B. Boehm and V. Basili. Software defect reduction top 10 list. *IEEE Computer*, pages 135–137, 2001.

[7] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *TSE*, 24(7):156–166, 1998.

[8] K. Cheung. Verify SCR requirements using XSPIN model checking to elevator case study, 2001.

[9] Y. Choi and M. Heimdahl. Model checking RSML-e cequirements. In *HASE*, pages 109–118, 2002.

[10] Y. Choi, S. Rayadurgam, and M. Heimdahl. Toward automation for model-checking requirements specifications with numeric constraints. *RE*, 7(4):225–242, 2002.

[11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *CAV*, pages 495–499, 1999.

[12] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *TOPLAS*, pages 244–263, 1986.

[13] E. Clarke, O. Grumberg, , and D. Peled. *Model Checking*. The MIT Press, first edition, 1999.

[14] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *FMSD*, 6(2):217–232, 1995.

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

[16] M. Dwyer, G. Avrunin, and J. Corbett. Property specification patterns for finite-state verification. In *FMSP*, pages 7–15, 1998.

[17] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[18] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *FMSD*, 4(3):243–263, 1994.

[19] D. Foyle and B. Hooey. Improving evaluation and system design through the use of off-nominal testing: A methodology for scenario development. In *Internaltional Symposium on Aviation Psychology*, pages 397–402, 2003.

[20] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in tropos. In *RE*, pages 174–181, 2001.

[21] M. Giese and R. Heldal. From informal to formal specifications in UML. In *UML*, pages 197–211.

[22] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *TSE*, 24(11):927–948, 1998.

[23] S. Konrad and B. Cheng. Facilitating the construction of specification pattern-based properties. In *RE*, pages 329–338, 2005.

[24] A. Lamsweerde. Formal specification: a roadmap. In *ICSE*, pages 147–159, 2000.

[25] N. Leveson. The role of software in recent aerospace accidents. In *International System Safety Conference*, 2001.

[26] N. Leveson. Systemic factors in software-related spacecraft accidents, 2001.

[27] S. Mallick and S. Krishna. Requirements engineering: Problem domain knowledge capture and the deliberation process support. In *DEXA*, pages 392–397, 1999.

[28] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *ICSE*, pages 35–46, 2000.

[29] T. Sreemani and J. Atlee. Feasibility of model checking software requirements: A case study. In *Computer Assurance*, pages 77–88, 1996.

[30] U. Stern and D. Dill. Automatic verification of the SCI cache coherence protocol. In *IFIP WG10.5*, pages 21–34, 1995.

[31] K. Suda and N. Rani. The importance of risk radar in software risk management: A case of a Malaysian company. *J. of Business and Social Science*, 1(3), 2010.

[32] N. Ubayashi, Y. Kamei, M. Hirayama, and T. Tamai. A context analysis method for embedded systems - Exploring a requirement boundary between a system and its context. In *RE*, pages 143–152, 2011.

[33] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *TACAS*, pages 382–396, 2008.

[34] E. Yu. Social modeling and i. In *Conceptual Modeling: Foundations and Applications*, pages 99–121, 2009.