Run-time Monitoring of Self-Adaptive Systems to Detect N-way Feature Interactions and their Causes

Byron DeVries
School of Computing and Information Systems
Grand Valley State University
Allendale, Michigan
byron_devries@gvsu.edu

Betty H.C. Cheng
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan
chengb@cse.msu.edu

ABSTRACT

The validity of systems at run time depends on the features included in those systems operating as specified. However, when feature interactions occur, the specifications no longer reflect the state of the run-time system due to the conflict. While methods exist to detect feature interactions at design time, conflicts that cause features to fail may still arise when new detected feature interactions are considered unreachable, new features are added, or an exhaustive design-time detection approach is impractical due to computational costs. This paper introduces *Thoosa*, an approach for using models at run time to detect features that can fail due to n-way feature interactions at run time and thereby trigger mitigating adaptations and/or updates to the requirements. We illustrate our approach by applying *Thoosa* to an industry-based automotive braking system comprising multiple subsystems.

CCS CONCEPTS

• **Software and its engineering → Interoperability**; **Requirements analysis**;

KEYWORDS

Requirements, Feature Interactions, Self-Monitoring

ACM Reference Format:
Byron DeVries and Betty H.C. Cheng. 2018. Run-time Monitoring of Self-Adaptive Systems to Detect N-way Feature Interactions and their Causes. In *SEAMS '18: SEAMS '18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194133.3194141

1 INTRODUCTION

While systems are expected to satisfy their specifications at run time, these specifications can only be satisfied if they are logically sound and do not include conflicts that result in feature interactions (FIs). Detecting feature interactions is challenging due to the exponential growth of potential interactions with respect to the number of features [6] and the range of environmental possibilities. More formally, when defining a feature interaction by the set of features that are necessary for the interaction to occur, the number of possible interactions is $O(2^{|F|})$, where F is the set of features and each feature may either be included or excluded from a feature set [2]. The growth in the number of possible feature interactions and consequently the number of possible feature interactions that must be assessed can result in latent behavior that impacts the system dependability at run time when exhaustive analysis of possible feature interactions is intractable. This problem is particularly prominent in cyber-physical systems where the range of environmental conditions compound the computational cost. However, run-time detection of feature interactions reduces the possible feature and environmental combinations to a single concrete instantiation that must be analyzed for feature interactions, rather than an exponential number of feature sets. This paper presents *Thoosa*,[1] a method for detecting each feature that causes an n-way feature interaction in a requirements goal model at run time, providing a trigger for adaptation.

Design-time n-way FI detection methods typically detect a potentially exponential number of FIs (i.e., $O(2^{|F|})$, where F is the set of features). In order to address the computational intractability of n-way FI detection, many researchers have focused on pair-wise interactions [5, 8, 18, 25]. While this approach decreases the number of potential interactions detected to $O(|F|^2)$, interactions that only emerge when three or more features are present will not be detected [2]. Run-time methods with lower computational costs exist [23, 24], but are focused on resolving or avoiding the interaction without guarantees of correctly modified behavior, rather than identifying suitable information (e.g., the cause of the interaction) to support adaptation away from the interacting specification.

This paper presents *Thoosa*, a method for detecting each feature that fails at run time, due to n-way feature interactions. These interactions might otherwise be undetectable at design-time due to the intractability of the problem at scale, in order to trigger adaptation. That is, feature interaction counterexamples are identified via a feature-oriented analysis based on whether a feature fails due to an interaction given the concrete instantiation of the system at a given point in time. For example, consider 3 features, $\mathbf{F_1}$, $\mathbf{F_2}$, and $\mathbf{F_3}$. If an interaction exists, then at least one feature ($\mathbf{F_1}$, $\mathbf{F_2}$, or $\mathbf{F_3}$) no longer operates as it did independently and thereby causes the feature interaction. *Thoosa* reduces the computational effort by only analyzing a single scenario and set of features (i.e., the current system configuration and environmental values expressed when the

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194141

[1] *Thoosa* is a Greek sea nymph associated with swiftness and the daughter of *Phorcys*.

measurements take place during execution of the system), at run time, while detecting n-way feature interactions and their causes, in order to support subsequent adaptation triggers and adaptation.

Thoosa analyzes features represented in goal models that hierarchically decompose a high-level goal down to individual requirements [33]. *Thoosa* executes generated logic, in the form of C++ code, that analyzes each feature with respect to the current feature combinations of the goal model and identifies if the analyzed feature can fail due to a conflict in one or more requirements to be satisfied at run time. Each feature is analyzed for failure due to a feature interaction. Where previous run-time feature interaction detection techniques indicate that a feature interaction exists, *Thoosa* identifies which features fail due to the feature interaction.

The contributions of this paper are as follows:

- We introduce a new approach for analyzing requirements to determine if they fail at run time due to a feature interaction, when design-time approaches are intractable, in order to trigger adaptation,
- We present *Thoosa*, a prototype implementation of the approach, and
- We demonstrate the applicability of *Thoosa* on an industry-based application of an automotive braking system based in part on the feature interaction issues in the 2010 Toyota Prius [12, 27], a hybrid vehicle. Analysis shows that *Thoosa* is able to identify feature failures caused by feature interactions in the braking system goal model involving two or more features.

The remainder of this paper is organized into the following sections. Background information is covered in Section 2. Section 3 details the approach. Section 4 describes an example application, and Section 5 details related work. Finally, Section 6 discusses the conclusions and avenues of future work.

2 BACKGROUND

This section overviews FIs, goal-oriented requirements modeling, and automotive braking systems.

2.1 Feature Interactions

Previously, Calder, *et al.* [7] formally described an FI between two features (F_1 and F_2) and their respective properties (ϕ_1 and ϕ_2). The description is as follows. A feature, F_1, satisfies a property, ϕ_1, denoted as $F_1 \models \phi_1$. Features may be combined, or composed, via a composition operator. Since decomposed requirements, goals, and features in a goal model are declarative and must be satisfied or unsatisfied in parallel, we use a conjunction operator (i.e., $\wedge$) and only consider parallel composition.[2] For example, features F_1 and F_2, may be composed as $F_1 \wedge F_2$. When $F_1 \models \phi_1$ and $F_2 \models \phi_2$, the expected composition of F_1 and F_2 should satisfy the conjunction of their respective properties, that is, $F_1 \wedge F_2 \models \phi_1 \wedge \phi_2$. However, if the composition of the features does not satisfy the conjunction of their respective properties, then there exists an FI [7] as shown in Equation 1:

$$F_1 \wedge F_2 \not\models \phi_1 \wedge \phi_2. \quad (1)$$

[2]Future work may include sequential composition.

2.2 Goal-Oriented Requirements Modeling

Goal-Oriented requirements engineering (GORE) is a graph-based representation of high-level goals (e.g., goals **A**, **A.1**, and **A.2** in Figure 1 that depicts the braking system) and objectives that must be satisfied by the system-to-be. Goals are decomposed from high-level objectives into progressively finer-grained goals and eventually into system requirements (e.g., **B.2** in Figure 1) and environmental expectations (e.g., **B.1** and **B.3** in Figure 1) [10, 33]. The goal modeling used in this paper includes AND and OR goal refinements. A goal that has been decomposed via AND-decomposition may only be satisfied if all of its decomposed goals are satisfied. Similarly, a goal that has been decomposed via OR-decomposition requires at least one of its decomposed goals to be satisfied [14]. The decomposition continues until a goal can be satisfied by an element of the system or the environment (including external systems). If the leaf-level goal is handled by a system element, then that goal is a *requirement*. If the element is part of the environment then the corresponding leaf-level goal is an *expectation* [10, 22, 33].

Goals may be expressed as functional or non-functional goals. Functional goals describe a service of the system-to-be, while non-functional goals apply a constraint on those services. The functional goals may be classified as an invariant or non-invariant, denoted 'Maintain' and 'Achieve,' respectively [33]. When used for leaf nodes of a goal model, the functional goals are requirements due to their relation to the system-to-be. In this paper, goal model labels are in **bold courier**.

2.3 Braking System Goal Model Example

The goal model in Figure 1 represents an automotive braking system with several features. In general, a feature reads the actual brake pedal position and relates that information to a commanded brake force that is translated to an external braking force via a braking mechanism. However, for this system, the braking system has been developed as four individual features (i.e., **B**, **C**, **D**, **E**) rather than as a single, monolithic system. The use of multiple brake features is based on a known 2010 Toyota Prius braking system issue [12]. The braking system features defined here have been independently developed without respect for each other and is intended to be a realistic example of industrial design artifacts at the requirements level, based on our collaboration with automotive industrial practitioners.[3]

Goal Models [33] are notationally extended in this work to include the concept of features. While a goal model defines the system-to-be [21], we add an additional notation, '**(f)**,' to denote specific goals and their decompositions representing features within the system. Expectations are annotated with '**(pre)**' and '**(post)**' to indicate goal and requirement pre- and post-conditions, respectively. While the pre- and post-conditions used in the goal models in this paper could, semantically, be represented by operationalized [28] pre- and post-conditions for each requirement, we intend the pre- and post-conditions used to be specified at a higher, more abstract level than an operationalized perspective. For example, specific ratios used in the braking system goal model are commonly included

[3]Collaboration with automotive industrial practitioners included reviewing our goal modeling approach to automotive systems, however the specific braking system presented here was not explicitly reviewed.

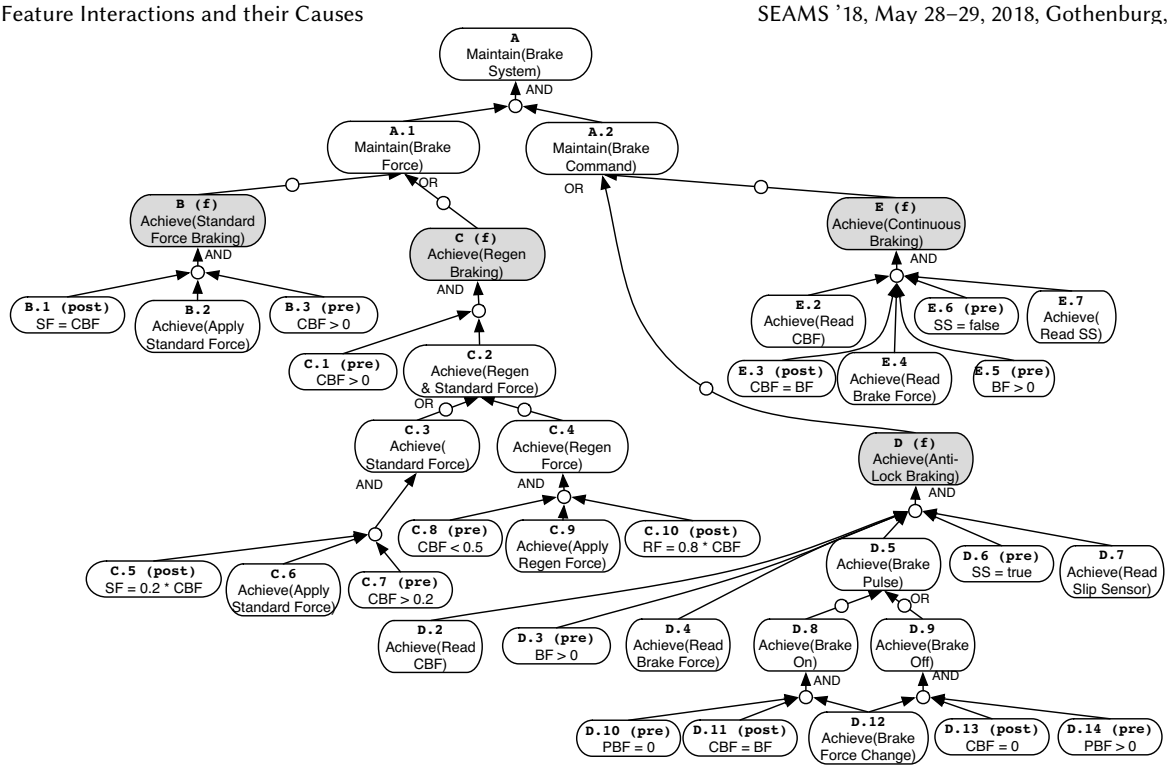

Figure 1: Braking System Goal Model

in high-level specifications regarding input and output limitations of line-replaceable components that interface with the system under development. The use of expectations in this manner is similar to awareness requirements for domain assumptions [31, 32].

The features included are standard force braking, regenerative braking, continuous braking, and anti-lock brakes, where the target vehicle is a hybrid (i.e., contains both an electric motor and gas engine). Acronyms used are: PBF (**P**revious **B**rake **F**orce), CBF (**C**ommanded **B**rake **F**orce), SS (**S**lip **S**ensor), RF (**R**egenerative Brake **F**orce), SF (**S**tandard Brake **F**orce), and BF (**B**rake **F**orce). Each of these features was specified without specific regard for the other features, and operates as described below.

Feature B: *'Achieve(Standard Force Braking)'* applies (via **B.2**) a brake force through standard force brake methods (e.g., disk brakes on the wheels) based on a commanded brake force using the *Hydraulic Brake Actuator*.

Feature C: *'Achieve(Regen Braking)'* applies a brake force using both standard brake force methods (via **C.3**) as well as regenerative braking methods (via **C.4**) that help capture and store electrical energy from braking. Due to limitations in the amount of braking force that regenerative braking may provide, three combinations are used for different braking needs: regenerative force is used alone for low-braking force needs (e.g., slight decrease in speed); a combination of both is used when a mid-range braking force is needed (e.g., a gradual slow down); and standard braking force is used when a large amount of braking force is needed (e.g., immediate/emergency stopping needed). In all cases, the amount of braking force is dependent on the commanded brake force and is applied using the *Hydraulic Brake Actuator* and/or the *Regeneration Brake Actuator*, depending on the *Commanded Brake Force* value.

Feature D: When the *Slip Sensor* detects slippage of the wheels on the driving surface (**D.6** and **D.7**) indicating a skid *'Achieve(Anti-Lock Braking)'* reads the current brake force from the *Brake Pedal Sensor* (**D.4**) and achieves a "pulsing" brake capability by alternating between applying the *Commanded Brake Force* (**D.8**) or applying no commanded brake force (**D.9**).

Feature E: *'Achieve(Continuous Braking)'* reads (**E.4**) the current brake force from the *Brake Pedal Sensor* and applies it to the *Commanded Brake Force* (**E.2**) when the *Slip Sensor* does not detect any slippage (**E.7**).

These features either *command* the brake force (e.g., **D**, **E**) or physically *apply* the brake force (e.g., **B**, **C**). In order to satisfy the top-level braking goal (**A**), there must be a commanded and applied brake force. Features **D** and **E** are mutually exclusive due to pre-conditions **E.6** (*SS* = false) and **D.6** (*SS* = true), since the slip sensor, *SS*, cannot both be true and false simultaneously. Therefore, due to the AND decomposition of both goal **D** and goal **E**, if either pre-condition **D.6** or **E.6** is unsatisfied, then the entire feature is inactive. The set of features that satisfies the top-level goal is captured via the decompositions in goals **A**, **A.1**, and **A.2**.

3 APPROACH

Figure 2 overviews the *Thoosa* process with a data flow diagram (DFD) where the circles represent processing elements, the parallel horizontal bars represent persistent data, the labeled arrows represent data flows, and the boxes represent external entities. The external element ("Adapt and/or Record Failures") is described, but not included in the *Thoosa* implementation. As shown in Step 1 of Figure 2, *Thoosa* accepts a goal model defined by the system designer (including feature and pre-/post-condition annotations),

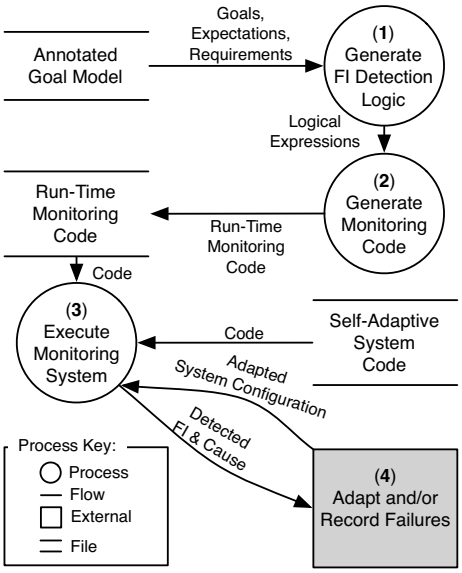

Figure 2: *Thoosa* Data Flow Diagram

such as the one in Figure 1, and generates feature interaction detection logic. Step 2 generates monitoring code that can be executed at run time based on the feature interaction detection logic. The *Thoosa* output from Step 2 contains the generated monitoring code that can detect feature failures due to feature interactions. In Step 3, that output is used to monitor a self-adaptive system for feature interactions and trigger adaptation and/or logging. Possible adaptations are described in Step 4. Next, we describe each of the steps in the DFD in turn.

3.1 Step (1): Generate FI Detection Logic

The 2-way interaction concept in Equation 1 can be extended to represent n-way interactions where any number of composed features result in the satisfaction of their aggregate properties:

$$\bigwedge_{i=1}^{|F|}(F_i) \not\models \bigwedge_{i=1}^{|F|}(\phi_i), \tag{2}$$

where F is a set of features that satisfies any overall feature composition requirements from the full feature set, S, that contains all features of the system-to-be (i.e., $F \subseteq S$). Equation 2 specifies that if the set of features does not fulfill their respective set of properties, then there is an FI.

Thoosa uses pre- and post-conditions to represent both the features and the feature properties. Therefore, in Equation 2, the features (i.e., F) are represented by a set of pre-conditions (e.g., for feature **B** the pre-condition $F_{\mathbf{B}}$ is **B.3** and for feature **C** the pre-condition $F_{\mathbf{C}}$ is **C.1**, **C.7**, and **C.8**), indicating the feature is included and/or executed. Similarly, the feature properties (i.e., ϕ) are represented by the feature post-conditions (e.g., for feature **B** the post-condition $\phi_{\mathbf{B}}$ is **B.1** and for feature **C** the post-condition $\phi_{\mathbf{C}}$ is **C.5** and **C.10**).

Features that can fail due to an FI are detected based on generated logical expressions for each feature represented in the goal model provided by the system designer. *Thoosa* automatically generates executable code based on the logic generated to detect failures that

```
// Calculate the top-level goal satisfaction
(topLevelSat = ((B() || C())&&(D() || E())));

// Calculate the satisfaction of each
//      feature's preconditions
(B_precondition = B_P());
(C_precondition = C_P());
(D_precondition = D_P());
(E_precondition = E_P());

// Calculate features that cause an FI
(B_fails = B_PP());
(C_fails = C_PP());
(D_fails = D_PP());
(E_fails = E_PP());
```

Figure 3: Variables Calculated for FI Detection in Figure 1

are caused by feature interactions. Specifically, the following items are calculated and assigned to variables based on the current state of goal model variables (i.e., the values those variables take on at run time while executing the implemented system) representing the state space:

- The satisfaction of the goal model based on the decomposition of the top-level features and their pre-conditions (e.g., $(F_{\mathbf{B}} \vee F_{\mathbf{C}}) \wedge (F_{\mathbf{D}} \vee F_{\mathbf{E}})$),
- The satisfaction of each of the feature pre-conditions (i.e., F_i in Equation 2 where $F_i \in F$) corresponding to *if* the feature should be included in the feature set and satisfied, and
- At least one feature with an unsatisfied post-condition, despite the satisfaction of its pre-condition, due to the inclusion of another feature or features (i.e., $F_i \wedge \neg\phi_i$ where $F_i \in F$ and $\phi_i \in \phi$) indicating an included but unsatisfied feature due to the inclusion of another feature or features.

3.2 Step (2): Generate Executable Code

An example of the generated code, intended to be platform independent, is shown in the partial code listing in Figure 3, where the top-level goal satisfaction, feature pre-condition satisfaction, and feature satisfaction are all calculated.

For example, feature **B** would be detected (i.e., by *B_PP()* in Figure 3) as causing an interaction in the case that its pre-conditions (evaluated by *B_P()*, *C_P()*, *D_P()*, and *E_P()*) include it in the feature set *and* its post-conditions are not satisfied as shown in Equation 3, where F is the set of features in the feature set that satisfies $(F_{\mathbf{B}} \vee F_{\mathbf{C}}) \wedge (F_{\mathbf{D}} \vee F_{\mathbf{E}})$:

$$\bigwedge_{i=1}^{|F|}(F_i) \wedge \neg\phi_{\mathrm{B}}. \tag{3}$$

Importantly, *all* features are analyzed for failure using one analysis, or execution of the generated code, instead of analyzing features sequentially, one at a time. That is, each feature is analyzed for its current individual ability to cause a feature interaction (e.g., 'B_fails,' 'C_fails,' 'D_fails,' and 'E_fails' in Figure 3) by sharing common sub-computations. The size and execution time of the generated code grows linearly with the number of goals, requirements, and expectations in the input goal model.

3.3 Step (3): Execute Monitoring System

The combination of the generated run-time monitoring code as well as the self-adaptive system code is compiled and deployed within the system under study. During the execution of the compiled and deployed code, the monitoring system is executed periodically. While the system designer may dictate any desired monitoring frequency, interactions are best detected when the detection logic is executed each time sensors are read or actuators are written.

The execution of the generated detection logic included in the self-adaptive system triggers an adaptation and/or a recording of the interaction whenever an interaction is detected.

3.4 Step (4): Adapt and/or Record Failures

Based on the values that are returned from the run-time detection of feature interactions and the features that fail due to the interaction, a partial list of appropriate systems actions that could be manually defined by the system designer are:

- The feature interaction along with the failing feature(s) could be recorded for later analysis. For example, a recorded feature interaction, especially those that may be difficult to observe externally when the system is still satisfied as a whole, may be used by the system designer to update the system requirements and expectations to mitigate that feature interaction.
- Specific mitigation requirements may be put in place to allow fail-safe or fail-silent operation.
- Specific feature failures due to feature interactions and their satisfied pre-conditions may be used as guidance or input to run-time adaptations to adapt from the feature set that causes a feature interaction.

Additionally, automated adaptation that could be employed including the following:

- Planning and executing methods that adhere to the MAPE-K loop [26] where *Thoosa* monitors system and environmental variables to analyze them for feature interactions. These systems could adapt behavior to ensure desirable system behavior [11, 30, 34] or adapt test cases to more accurately identify unwanted/latent behavior issues [21].
- RELAXing [35] pre- and post-condition specifications [20].
- One of the many adaptation methods described for cyber-physical systems [29].

3.5 Limitations

Counterexamples may be present in a scenario that occurs only between sensor readings. In such cases, the values calculated never represent a feature interaction and, therefore, no counterexample is detected. Additionally, the input goal model is assumed to represent the system correctly and be complete [14, 15, 17].

4 EXAMPLES

This section provides the results of applying the *Thoosa* tool with simulated run-time inputs. The braking system defined in Figure 1 identifies two potential sources for driving brake commands (features **D** and **E**) that provide the inputs to two methods of physically applying the brakes (features **B** and **C**). Previous analysis of the braking system has shown that both features **B** and **C** can cause an interaction [13, 16]. We apply the variables identified in that previous analysis to ensure the run-time approach of *Thoosa* correctly identifies features that can fail due to an interaction given a concrete scenario rather than exhaustive analysis. The time required to calculate if failures exist due to feature interactions is less than one ten-thousandth of a second.

Run-Time Analysis of Feature **B** *for Failure Due to FI.* The values of the variables in the goal model counterexample when Feature **B** is the cause of a feature interaction are shown in Table 1a and are applied to the run-time system for detection. Figure 4 illustrates the applicable parts of the goal model (at the feature level) for the 3-way feature interaction (across features **B**, **C**, and **D**) caused by feature **B** where the red text (SF = ?) indicates the conflict that should be detected. The brake force (BF = 0.35) is provided externally by the driver pressing on the brake pedal. This value is read via the *'Brake Pedal Sensor'* and converted to a commanded brake force (CBF = 0.35) by feature **D**. The features that apply the brake force, **B** and **C**, read this commanded brake force (CBF) from memory and use it to apply external brake forces. Feature **B** applies, via **B.2**, the entire commanded brake force to standard braking force (SF = 0.35) (e.g., drum or disk brakes on the wheels). Feature **C** (i.e., regenerative braking feature) applies a portion (defined by **C.5**) of the commanded brake force to the standard braking force (SF = 0.07) and a portion (defined by **C.10**) to the regenerative braking force (RF = 0.28). The 3-way interaction occurs due to the conflict where features **B** and **C** both attempt to set SF to different values, clearly SF = 0.35 and SF = 0.07 cannot be true simultaneously.

Table 2 shows how *Thoosa* identifies that feature **B** fails due to a feature interaction, along with the remainder of the run-time results. That is, when a known counterexample is used to initialize the system variables, the detection code generated by *Thoosa* is able to detect the known feature interaction. Note, the top-level goal is still satisfied because *either* goal **B** or goal **C** is required, and goal **C** is still satisfied. Therefore, despite the feature interaction, the system still succeeds, but not as intended. In this case, the system designer may record this feature interaction, but adaptation is not strictly necessary due to the top-level satisfaction of the system. Importantly, FIs that occur while the overall system is still satisfied expose latent properties that are less likely to be detected based only on a static analysis of the overall system behavior since the overall system is still satisfied. These errors should be addressed to avoid their untimely manifestation in other run-time scenarios. For example, updates may include strengthening pre-conditions , weakening post-conditions , or revising the system requirements entirely. Other strategies are proposed in Section 3.4.

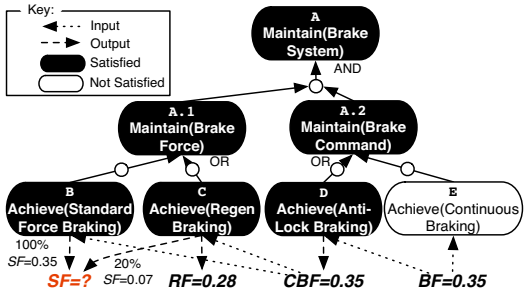

Figure 4: 3-Way FI in Figure 1 Caused by B or C

Run-Time Analysis of Feature **C** *for Failure Due to FI.* A counterexample with a similar type of failure is the interaction failure due to feature **C**: a different conflict between **B** and **C** attempting to set *SF* to conflicting values. The variable values applied to the run-time system, as shown in Table 1b, result in the detection of a failure of feature **C** due to the interaction.

Table 2 shows how *Thoosa* identifies that feature **C** fails due to a feature interaction, along with the remainder of the run-time results. Note, the top-level goal is still satisfied because *either* goal **B** or goal **C** is required, and goal **B** is still satisfied. Therefore, despite the feature interaction the system still succeeds, but not as intended. This is similar to the top-level satisfaction shown again, when feature **B** fails due to a feature interaction. Again, no adaptation is necessarily required, however recording the feature interaction is one of several possibilities proposed in Section 3.4.

5 RELATED WORK

Significant work has been done in the area of detecting, avoiding, and resolving feature interactions. This section overviews and compares related work limited to those that address run-time detection, the focus of this work. More comprehensive surveys on feature interactions have been presented elsewhere [1, 2, 7].

Often run-time detection techniques are paired with an effort at run-time resolution, including techniques such as prioritization [9] and negotiation [23] between features. Recently, methods to resolve interactions based on the conflicting variables have been presented to reduce the number of resolutions [6, 36], though dependencies between variables (e.g., speed and acceleration) are not considered for interactions. This method does not provide guarantees that the resolution provides desired behavior for the system. Therefore, while resolution at run-time may provide an acceptable solution, it is not guaranteed. *Thoosa* operates at the requirements level and identifies which features fail due to the feature interaction and allows for mitigations to be created that adapt from an interacting set of features while still satisfying the top-level goal.

Table 1: Two Separate Scenarios where FIs Occur

(a) Feature B as Cause

Variable	Value
SF	0.07
CBF	0.35
PBF	0.0
RF	0.28
BF	0.35
SS	*true*

(b) Feature C as Cause

Variable	Value
SF	0.35
CBF	0.35
PBF	0.0
RF	0.28
BF	0.35
SS	*true*

Table 2: Run-Time Results for Scenarios of Features B and C

Identifier	In Code	Scenario B Result	Scenario C Result
Top-Level (Goal **A**)	topLevelSat	Satisfied	Satisfied
Feature **B** Fails	B_fails	True	False
Feature **C** Fails	C_fails	False	True
Feature **D** Fails	D_fails	False	False
Feature **E** Fails	D_fails	False	False
Pre-Conditions for **B**	B_precondition	True	True
Pre-Conditions for **C**	C_precondition	True	True
Pre-Conditions for **D**	D_precondition	True	True
Pre-Conditions for **E**	E_precondition	False	False

Typically, the design-time detection of feature interactions focuses on identifying minimal sets of features that are necessary for a feature interaction to take place [3, 4, 18, 25]. Often, these methods are limited to pair-wise detection [8, 18] or other small feature sets [19], where only interactions between two features can be detected to reduce computational cost. *Thoosa*, however, detects n-way feature interactions at run-time, and only inspects a single scenario (i.e., set of variables and features based on the current state of the system and environment) at a time.

Thoosa differs from existing feature interaction detection methods by *combining* these key elements:

- it operates at the requirements level,
- it detects n-way feature interactions,
- it identifies which features can fail due to an interaction,
- it triggers an adaptation or logging of the failure, and
- it only inspects a single scenario (i.e., set of variables and features) at run time, reducing computational cost.

6 CONCLUSION

In this paper, we have described *Thoosa*, a run-time approach for detecting unwanted n-way feature interactions and determining features that they can cause to fail. *Thoosa* is an important practical assessment of feature interactions at run time when the design-time computational costs of n-way interactions are infeasible. Unlike previous n-way feature interaction detection approaches that detect feature interactions at run time, *Thoosa* also identifies the features that can fail due to the feature interaction that is detected. *Thoosa* reduces the effort to log errors or adapt run-time behavior by identifying which features fail in an interaction rather than simply detecting that an interaction has occurred. We have demonstrated *Thoosa* on a braking system goal model comprising several features to realize an automotive braking system.

Future research will explore how specific mitigation strategies can be applied to support run-time adaptation. Additionally, we will continue to explore how *Thoosa* can be extended to address additional assurance challenges posed by computing-based systems as they interact with the environment. For example, we will explore how to detect FIs in the face of uncertainty due to environmental conditions as they impact system conditions. Additionally, while *Thoosa* detects FIs at run time, we will also investigate design-time FI detection and mitigation strategies. We will also explore the use of RELAXed goals [35], whose satisfaction is evaluated according to fuzzy logic expressions.

ACKNOWLEDGMENT

This work has been supported in part by grants from NSF (CNS-1305358 and DBI-0939454), Ford Motor Company, and General Motors Research; and the research has also been sponsored by Air Force Research Laboratory (AFRL) under agreement number FA8750-16-2-0284. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL), the U.S. Government, National Science Foundation, Ford, GM, Michigan State University, or other research sponsors.

REFERENCES

[1] Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave. 2014. Feature Interactions: The Next Generation (Dagstuhl Seminar 14281). *Dagstuhl Reports* 4, 7 (2014), 1–24. https://doi.org/10.4230/DagRep.4.7.1

[2] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*. ACM, 1–8.

[3] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander Von Rhein, and Dirk Beyer. 2011. Detection of feature interactions using feature-aware verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 372–375.

[4] Sven Apel, Alexander Von Rhein, Thomas Thüm, and Christian Kästner. 2013. Feature-interaction detection based on feature-based specifications. *Computer Networks* 57, 12 (2013), 2399–2409.

[5] Silky Arora, Prahladavaradan Sampath, and S Ramesh. 2012. Resolving uncertainty in automotive feature interactions. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*. IEEE, 21–30.

[6] Cecylia Bocovich and Joanne M. Atlee. 2014. Variable-specific resolutions for feature interactions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 553–563. https://doi.org/10.1145/2635868.2635927

[7] Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41, 1 (2003), 115–141.

[8] Muffy Calder and Alice Miller. 2006. Feature interaction detection by pairwise analysis of LTL properties-a case study. *Formal Methods in System Design* 28, 3 (2006), 213–261.

[9] A Chavan, L Yang, K Ramachandran, and WH Leung. 2007. Resolving Feature Interaction with Precedence Lists in the Feature Language Extensions.. In *ICFI*. 114–128.

[10] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. 2009. On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications*. Springer, 363–379.

[11] Kevin Colson, Robin Dupuis, Lionel Montrieux, Zhenjiang Hu, Sebastián Uchitel, and Pierre-Yves Schobbens. 2016. Reusable self-adaptation through bidirectional programming. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2016 IEEE/ACM 11th International Symposium on*. IEEE, 4–15.

[12] Michael A Cusumano. 2011. Reflections on the Toyota debacle. *Commun. ACM* 54, 1 (2011), 33–35.

[13] Byron DeVries. 2017. *Using Formal Analysis and Search-Based Techniques to Address the Assurance of Cyber-Physical Systems at the Requirements Level*. Ph.D. Dissertation. Michigan State University.

[14] Byron DeVries and Betty H. C. Cheng. 2016. Automatic detection of incomplete requirements via symbolic analysis. In *19th International Conference on Model Driven Engineering Languages and Systems, MODELS 2016, Proceedings, Saint-Malo, France, October 2-7*. ACM, 385–395.

[15] Byron DeVries and Betty H. C. Cheng. 2017. Automatic Detection of Incomplete Requirements Using Symbolic Analysis and Evolutionary Computation. In *International Symposium on Search Based Software Engineering*. Springer, 49–64.

[16] Byron DeVries and Betty H. C. Cheng. 2017. Detecting Unintended N-Way Feature Interactions and their Causes for Autonomous Systems. *Technical Report* (2017). http://www.cse.msu.edu/%7Echengb/FI-Techreport-2017.pdf

[17] Byron DeVries and Betty H. C. Cheng. 2017. Using Models at Run Time to Detect Incomplete and Inconsistent Requirements. In *Proceedings of the 21th International Workshop on Models@run.time co-located with 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, Texas, September 18*.

[18] Alma L Juarez Dominguez. 2012. *Detection of feature interactions in automotive active safety features*. Ph.D. Dissertation. University of Waterloo.

[19] Alessandro Fantechi, Stefania Gnesi, and Laura Semini. 2017. Optimizing Feature Interaction Detection. In *Critical Systems: Formal Methods and Automated Verification*. Springer, 201–216.

[20] Erik M Fredericks, Byron DeVries, and Betty H. C. Cheng. 2014. AutoRELAX: automatically RELAXing a goal model to address uncertainty. *Empirical Software Engineering* 19, 5 (2014), 1466–1501.

[21] Erik M Fredericks, Byron DeVries, and Betty H. C. Cheng. 2014. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 17–26.

[22] Heather J Goldsby, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Danny Hughes. 2008. Goal-based modeling of dynamically adaptive system requirements. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*. IEEE, 36–45.

[23] Nancy D Griffeth and Hugo Velthuijsen. 1994. The negotiating agents approach to runtime feature interaction resolution.. In *FIW*. 217–235.

[24] Jonathan D. Hay and Joanne M. Atlee. 2000. Composing Features and Resolving Interactions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications (SIGSOFT '00/FSE-8)*. ACM, New York, NY, USA, 110–119. https://doi.org/10.1145/355045.355061

[25] Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan Gomaa. 2007. Model composition in product lines and feature interaction detection using critical pair analysis. In *Model Driven Engineering Languages and Systems*. Springer, 151–165.

[26] Jeffrey O Kephart and David M Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[27] Phil Koopman. 2014. A case study of *Toyota* unintended acceleration and software safety. *Presentation. Sept* (2014).

[28] Emmanuel Letier and Axel van Lamsweerde. 2002. Deriving operational software specifications from system goals. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, 119–128.

[29] Henry Muccini, Mohammad Sharaf, and Danny Weyns. 2016. Self-adaptation for cyber-physical systems: a systematic literature review. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 75–81.

[30] Ashutosh Pandey, Ivan Ruchkin, Bradley Schmerl, and Javier Cámara. 2017. Towards a formal framework for hybrid planning in self-adaptation. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 109–115.

[31] Vítor E Silva Souza, Alexei Lapouchnian, William N Robinson, and John Mylopoulos. 2011. Awareness requirements for adaptive systems. In *Proceedings of the 6th international symposium on Software engineering for adaptive and self-managing systems*. ACM, 60–69.

[32] Vítor E Silva Souza, Alexei Lapouchnian, William N Robinson, and John Mylopoulos. 2013. Awareness requirements. In *Software Engineering for Self-Adaptive Systems II*. Springer, 133–161.

[33] Axel van Lamsweerde et al. 2009. *Requirements engineering: from system goals to UML models to software specifications*. Wiley.

[34] Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. 2017. Quality-aware runtime adaptation in complex event processing. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 140–151.

[35] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. 2009. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*. IEEE, 79–88.

[36] M Hadi Zibaeenejad, Chi Zhang, and Joanne M Atlee. 2017. Continuous variable-specific resolutions of feature interactions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*. ACM, 408–418.