IEEE Copyright Notice

Published in: ***Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'19)***, February 2019

Detecting Feature-Interaction Symptoms in Automotive Software Using Lightweight Analysis

Cite as:

B. J. Muscedere, R. Hackman, D. Anbarnam, J. M. Atlee, I. J. Davis and M. W. Godfrey, "Detecting Feature-Interaction Symptoms in Automotive Software using Lightweight Analysis," *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, 2019, pp. 175-185.

BibTex:

```
@INPROCEEDINGS{8668042,
author={B. J. {Muscedere} and R. {Hackman} and D. {Anbarnam} and J. M. {Atlee} and I.
J. {Davis} and M. W. {Godfrey}},
booktitle={2019 IEEE 26th International Conference on Software Analysis, Evolution
and Reengineering (SANER)},
title={Detecting Feature-Interaction Symptoms in Automotive Software using
Lightweight Analysis},
year={2019},
volume={},
number={},
pages={175-185},
}
```

DOI: https://doi.org/10.1109/SANER.2019.8668042

Detecting Feature-Interaction Symptoms in Automotive Software using Lightweight Analysis

Bryan J. Muscedere, Robert Hackman, Davood Anbarnam, Joanne M. Atlee,
Ian J. Davis, and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{bmuscede, r2hackma, danbarnam, jmatlee, ijdavis, migod}@uwaterloo.ca

Abstract—**Modern automotive software systems are large, complex, and feature rich; they can contain over 100 million lines of code, comprising hundreds of features distributed across multiple electronic control units (ECUs), all operating in parallel and communicating over a CAN bus. Because they are safety-critical systems, the problem of possible Feature Interactions (FIs) must be addressed seriously; however, traditional detection approaches using dynamic analyses are unlikely to scale to the size of these systems. We are investigating an approach that detects static source-code patterns that are symptomatic of FIs. The tools report *Feature-Interaction warnings*, which can be investigated further by engineers to determine if they represent true FIs and if those FIs are problematic.**

In this paper, we present our preliminary toolchain for FI detection. First, we extract a collection of static "facts" from the source code, such as function calls, variable assignments, and messages between features. Next, we perform relational algebra transformations on this factbase to infer additional "facts" that represent more complicated design information about the code, such as potential information flows and data dependencies; then, the full collection of "facts" is matched against a curated set of patterns for FI symptoms. We present a set of five patterns for FIs in automotive software as well a case study in which we applied our tools to the *Autonomoose* autonomous-driving software, developed at the University of Waterloo. Our approach identified 1,444 possible FIs in this codebase, of which 10% were classified as being probable interactions worthy of further investigation.

Index Terms—**static analysis, relational algebra, feature interactions.**

I. Introduction

The complexity of automotive software continues to grow as functionality that used to be realized in electro-mechanical systems is increasingly implemented in software, and as new features are introduced to improve safety, fuel economy, driver experience, and semi-autonomous capabilities. The software in a modern-day automobile comprises more than 100 million lines of code distributed across as many as 120 electronic control units (ECUs), which communicate over a CAN bus [1]. To mitigate complexity, the software is decomposed into subsystems and **features**, each of which is a unit of functionality that can be considered, developed, and evolved independently. The downside of feature-based decomposition are unexpected **Feature Interactions** (FIs), where the actions of one feature interfere with those of one or more others, leading to conflicts, emergent functional behaviours, and possible artificial contagion to other features and systems [2] [3]. To illustrate, consider a vehicle that has two features: *Adaptive Cruise Control* (ACC), which manipulates the car's acceleration and brake actuators in order to maintain a set cruising speed, and *Lane Centering* (LC), which manipulates the car's steering to keep the vehicle in the centre of its lane. In isolation, these two features operate as expected. However, if they operate simultaneously without any coordination, ACC's impact on the car's forward speed and the LC's impact on steering angle could result in a dangerous lateral acceleration that causes the vehicle to roll over.

Feature Interactions present a real and significant safety risk to consumers and bystanders; automotive companies expend considerable effort in detecting and mitigating the risk of FIs. However, many traditional approaches for FI detection — such as code reviews and safety analyses — scale poorly as the number of features climbs into the hundreds and the number of feature combinations to analyze grows exponentially. Much previous research has focused on approaches that require models of feature behaviours [4] [5] or specifications of correct feature behaviour [6] [7]; typically, they employ dynamic analyses that have high precision and recall, but do not scale to large numbers of features.

We are exploring a static-analysis approach that extracts a lightweight model of a software system from its source code and looks for symptoms of FIs within this model. Reported instances of FI symptoms, which we call **FI warnings**, must be investigated further by engineers to determine if they represent true interactions, and if so, whether they are unintended or undesired. The ultimate goal of this work is to make a feasible, lightweight analysis that is effective in detecting potential FIs at the scale of tens-to-hundreds of features.

This paper presents our first steps in this work, in which we investigate the practicality of detecting non-trivial interactions between features, implemented as distinct C++ components that communicate via publish/subscribe primitives in the Robot Operating System (ROS) [8]. We have developed tools for exacting from C/C++ source code a collection of static "facts" about software components, such as function calls, variable assignments, and ROS messages between features. The facts are represented as tuples, and the collection of extracted facts, called a **factbase**, forms an initial model of the software. We then perform relational-algebra transformations on the factbase to infer additional "facts" that represent more complicated design information about the software, such as potential information flows and data dependencies which we add to the factbase. Next, we match this augmented model against a curated set of patterns that are symptomatic of FIs; this results in a set of **FI warnings**. The warnings must be investigated further by domain experts, who can determine whether a warning reflects a real undesired interaction that must be fixed, a real but benign interaction that can be ignored, or a false positive. Note that our initial tools are specific to the programming languages, communication primitives, and FI types that apply to automotive software; however, we hypothesize that our approach to fact extraction and FI warnings can be adapted to other application domains, and we will assess this in future works.

In this paper, we present our source-code fact-extraction tool called **Rex**; we present a set of five patterns for FIs for automotive software; and we present the results of a case study in which we assess the utility of our static-analysis approach to FI detection by applying it to the software for the *Autonomoose* autonomous driving project at the University of Waterloo [9]. Our relational queries on these facts found a total 1,444 FI warnings within the example software system; of these, 10% were classified by a developer of the system as being probable interactions worthy of further investigation.

The rest of this paper is organized as follows: Section II provides an overview of our approach to detect symptoms of FIs using static-analysis technologies. Section III introduces the *Rex* fact extraction tool. Section IV describes relational-algebraic definitions of symptoms for five different patterns of FIs. Section V presents the results of applying our tools to an automotive software system, and we discuss limitations of our approach in Section VI. Section VII reviews related work, and Section VIII concludes with plans for future work.

II. Overview

Figure 1 shows our static-analysis toolchain for detecting symptoms of Feature Interactions (FIs) in a software system. In the figure, green boxes represent files or data stores, pink boxes represent tools, and the single purple box represents the human ingenuity needed to devise the queries. Inputs to the toolchain are one or more files of C/C++ source code, which are fed to the **fact extractor**; the extractor distills important "facts" about the program entities and their relationships, creating a tuple-based model using the Tuple-Attribute (TA) language [10]. The *Grok* relational-algebra query engine [11] is used to infer additional "facts" through algebraic operations on this model, and also to pose queries. In our work, the queries, created by a human engineer, express likely symptoms of FIs in the code. Finally, the output is a collection of FI warnings that warrant further investigation using other methods such as code reviews or more sophisticated static or dynamic analyses.

Elements of this toolchain have been used in previous research projects for several tasks, including reverse engineering software architecture models [12] [13] and source-code clone detection [14]. Our research is novel in that the software under analysis is a collection of concurrent message-passing software components, rather than a single-threaded software system; this requires that the fact extractor is able to create a richer system model than in previous uses. Also, to the best of our knowledge, our application of these technologies to detect predefined FI patterns in source code is novel. The remainder of this section describes the two major tools of our toolchain, as well as the FI symptoms our work aims to detect.

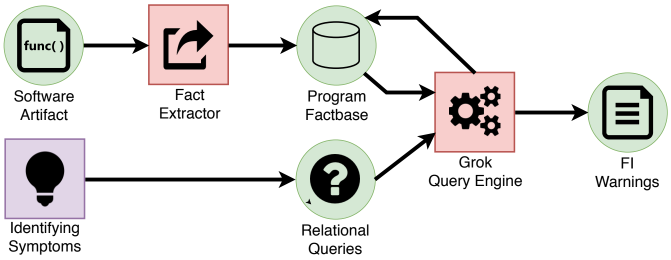


Fig. 1. Toolchain to detect symptoms of Feature Interactions.

A. Fact Extractor(s)

The first major step in our approach is to create a program model of the software system from the source code. A model is a collection of **"facts"** about the software entities and their relationships, and the model itself is called a **factbase**. We note that here, our facts are derived from static analysis, which is often imprecise and conservative; some facts that are statically "true" (e.g., *F calls G*) may not be possible at run-time (e.g., if the call is protected by a guard condition that is always false). Facts are encoded as three-tuples (triples) in the Tuple-Attribute (TA) language where the tuples define sets, relations between entities in the sets, and attributes that describe characteristics of entries in sets or relations.

A fact extractor is a custom tool, often built using compiler-like components such as scanners, parsers, and abstract syntax trees (ASTs); the reverse engineering research community has a long history of building fact extractors for various technical needs. Typically, an extractor processes the source code to generate facts of interest about the program's entities and their relationships; some extractors require full, compilable sources and access to deployment libraries and execution environments to do their job, while others are able to generate rich models from source code or object files. For large systems, extractors often mimic the design of compilers, supporting an initial per-file extraction phase akin to separate compilation followed by a merging phase similar to linking. Extractors may generate much more detailed information than is needed for typical uses; command-line options, filters, and post-processing transformations can limit the amount of detail generated, which in turn greatly reduces the amount of time needed to perform the merging/linking.

Our extractor, called **Rex** (**R**OS **Ex**tractor), processes C++ programs that run on the Robot Operating System (*ROS*) [8]; it is built using *Clang++* infrastructure [15]. *Rex* produces a

TABLE I
A SUBSET OF *Grok* RELATIONAL OPERATORS

Operator Name	Operator	Description
Union	⟨*ITEM*⟩ + ⟨*ITEM*⟩	Returns a set (relation) comprising the elements from the two operands, omitting duplicates
Intersection	⟨*ITEM*⟩ ^ ⟨*ITEM*⟩	Returns a set (relation) comprising the elements that are members of both operands
Difference	⟨*ITEM1*⟩ - ⟨*ITEM2*⟩	Returns a set (relation) comprising the elements that are in ⟨*ITEM1*⟩ but not in ⟨*ITEM2*⟩
Transitive closure	⟨*RELATION*⟩+	Returns the transitive closure of the operand ⟨*RELATION*⟩
Composition	⟨*RELATION1*⟩ o ⟨*RELATION2*⟩	Returns a relation {(x, z)| $\exists y$. (x,y)∈⟨*RELATION1*⟩ and (y,z)∈⟨*RELATION2*⟩}
Domain restriction	⟨*SET*⟩ o ⟨*RELATION*⟩	Returns the subset of ⟨*RELATION*⟩ whose domain elements are in ⟨*SET*⟩
Range restriction	⟨*RELATION*⟩ o ⟨*SET*⟩	Returns the subset of ⟨*RELATION*⟩ whose range elements are in ⟨*SET*⟩
Projection	⟨*SET*⟩ . ⟨*RELATION*⟩	Projects a set through a relation; returns set {y| $\exists x$. x∈⟨*SET*⟩ and (x,y)∈⟨*RELATION*⟩}
Projection	⟨*RELATION*⟩ . ⟨*SET*⟩	Projects a relation through a set; returns set {x| $\exists y$. (x,y)∈⟨*RELATION*⟩ and y∈⟨*SET*⟩}
Identity	`id` ⟨*SET*⟩	Returns a relation comprising an element (x,x) for each element x in ⟨*SET*⟩
Domain	`dom` ⟨*RELATION*⟩	Returns a set comprising the domain values of all elements in ⟨*RELATION*⟩
Range	`rng` ⟨*RELATION*⟩	Returns a set comprising the range values of all elements in ⟨*RELATION*⟩
Selection	⟨*RELATION*⟩[⟨*COND*⟩]	Returns the subset of ⟨*RELATION*⟩ whose elements satisfy ⟨*COND*⟩

model of the target program in TA that include both "generic" C++ facts (e.g., declarations, variable uses, function calls, message passes) as well as facts about *ROS* primitives (e.g., publishers/subscribers, features, topics, and their dependencies). The special-purpose analyses for possible FIs is performed on this TA model using *Grok* scripts; we describe *Grok* in the next section, and we discuss the special purpose *Grok* scripts for FI detection in more detail in Section IV.

B. Grok

Given a TA model (factbase) of a software system, one can infer additional "facts" about its design by performing appropriate relational-algebra operations using the *Grok* relational query engine [11]. The *Grok* environment supports basic set and relational operations (e.g., union, intersection, difference, and projection), relational composition, and transitive closure; transitive closure is particularly powerful as it allows us to infer indirect relationships from direct relationships (e.g., we can infer which functions are indirectly called by some main function by computing the transitive closure of the relation of direct function calls). Table I shows a subset of operations that *Grok* supports[1]. The results of an algebraic operation are new facts that can be added to the factbase and used in subsequent operations and queries.

C. Feature Interactions

A **Feature Interaction** (FI) occurs whenever one feature affects the behaviour of another [2]. How a Feature Interaction manifests itself depends on how features are represented [5]. For example, if features are expressed in logic, then interactions manifest as logical inconsistencies or unsatisfiability.

When features are expressed in code, FIs manifest in a wide variety of ways [5]:

1) nondeterminism among features' actions
2) inconsistent post-conditions of features
3) deadlock of features' executions
4) livelock of a feature's execution
5) control modification (e.g., some action by one feature causes a change in the control flow of another feature)
6) data modification (e.g., some variable assignment made by one feature causes a change to a variable assignment made by another feature)
7) resource contention, where features compete for scarce resources
8) loops in communication among features

As will be seen, this paper investigates symptoms that correspond to FI manifestations 1, 4, 5, and 8.

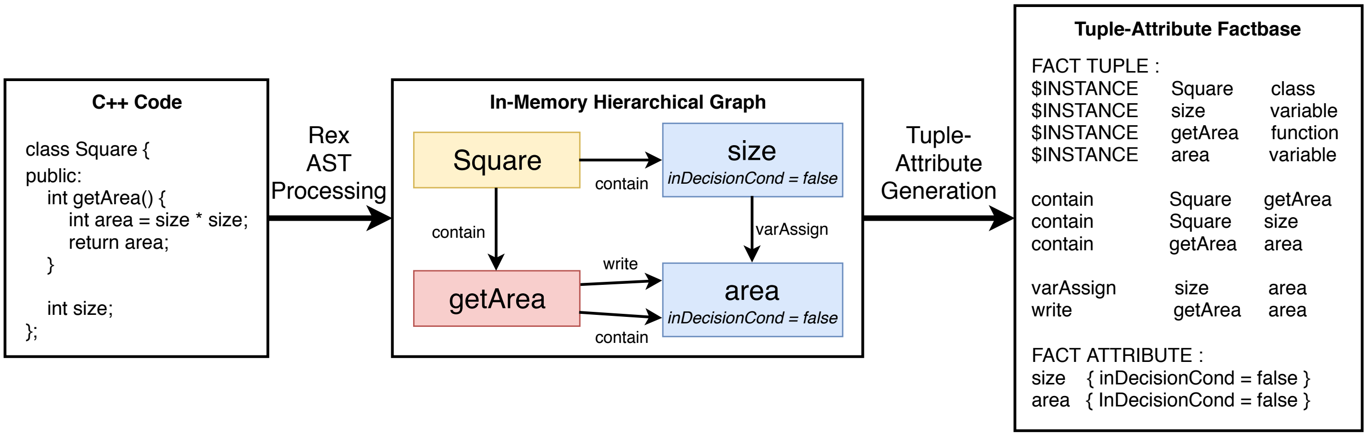
III. FACT EXTRACTION

In this section, we present our fact extractor *Rex* (**R**OS **Ex**tractor) for programs that are written in C/C++ and that use the Robot Operating System (*ROS*) framework [8] to enable communications between software components. In addition to extracting typical facts about top-level C++ entities, such as `classes`, `functions`, and function `calls`, *Rex* also extracts facts about *ROS* communication primitives, such as instances of `publish` and `subscribe`, so that we can analyze possible data flows between components. One complication in analyzing a *ROS*-based system is that it uses a unique build system, called *catkin*, which determines the set of source-code files that make up a build as well as the compiler flags associated with each file; the fact extractor needs these details to extract a faithful model (of a specific build) of the software. To explicate this information, the system-build scripts need to be modified to generate a **compilation database** comprised of the pathnames of all the source-code files that make up the build along with their respective compiler flags; this compilation data is part of the input to the fact extractor. The remainder of this section provides a brief description of our fact extractor[2].

Rex is an adaptation of the *Clang*++ open-source compiler [15], which parses C++ source-code files and generates corresponding abstract syntax trees (ASTs). Whenever *Clang*++ generates an AST node that matches information *Rex* deems to be important, *Rex* records information about that node in an in-memory hierarchical graph. Graph nodes correspond to entities and graph edges correspond to relations in the resultant TA factbase; both nodes and edges can have

[1] Additional *Grok* operators and functions can be found here [16].

[2] Detailed information about the *Rex* fact extractor can be found in [17].



```
class Square {
public:
    int getArea() {
        int area = size * size;
        return area;
    }

    int size;
};
```

```
FACT TUPLE :
$INSTANCE    Square     class
$INSTANCE    size       variable
$INSTANCE    getArea    function
$INSTANCE    area       variable

contain      Square     getArea
contain      Square     size
contain      getArea    area

varAssign    size       area
write        getArea    area

FACT ATTRIBUTE :
size  { inDecisionCond = false }
area  { InDecisionCond = false }
```

Fig. 2. An example (non-*ROS*-based) C++ program being converted into a TA factbase.

associated attributes. Figure 2 shows an example (non-*ROS*-based) C++ program, the in-memory hierarchical graph that *Rex* would produce, and the resultant TA factbase outputted by *Rex*. In this example, as *Clang*++ creates AST nodes for class `Square`, function `getArea()`, and variables `area` and `size`[3], *Rex* creates corresponding graph nodes. For each declaration in the AST, *Rex* generates a `contain` edge from the graph node of declaring function, class, or file to the graph node of the declared entity. When *Rex* sees the initialization of variable `area`, it creates a `write` edge between function `getArea()` and variable `area`. Other edges are created in the same manner.

Once the construction of the in-memory hierarchical graph is completed, information about feature entities and their associated relationships are added. The concept of a **feature** has many definitions in the literature; we define a feature to be "a coherent and identifiable bundle of system functionality that helps characterize the system from a user perspective" [18]. In *ROS*, projects are divided into packages that modularize code into divisible units that are comparable to features; thus, *Rex* adds a feature node to the in-memory graph for each *ROS* package in the project[4]. Then, to maintain the hierarchy of items, every other node is nested — by adding a `contain` edge — under its associated feature. To do this for each recorded entity, during the construction of the in-memory hierarchical graph, *Rex* creates a list of source files that refer to that entity. If an entity is used in multiple files, all files that refer to that entity are recorded. In the feature resolution phase, the compilation database is scanned to determine which files are included in the build; and the in-memory hierarchical graph is augmented with edges from the features (files) that are included in the build to their entities.

Exporting a completed hierarchical graph to a TA model is straightforward because a TA model can be viewed as a textual representation of a graph. In a TA factbase, identifiers (IDs) need to be declared before they are used. Thus, a hierarchical graph is converted to a TA model as follows: first all graph nodes are outputted as entities, then all graph edges are outputted as relations, and finally all attributes of nodes and edges are outputted. The tuple representations of nodes, edges, and attributes, respectively are:

```
$INSTANCE ⟨NODE_ID⟩ ⟨NODE_TYPE⟩
⟨EDGE_TYPE⟩ ⟨EDGE_SOURCE⟩ ⟨EDGE_DESTINATION⟩
⟨ID⟩ { ⟨KEY⟩ = ⟨VALUE⟩ ... }
```

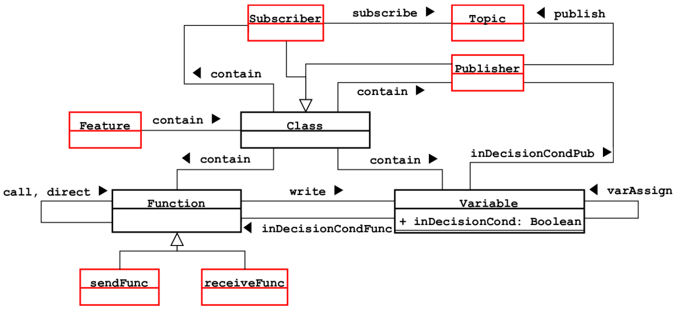


Fig. 3. Entities, relations, and attributes contained in *Rex* models.

IV. Feature-Interaction Symptoms

In this section, we present analyses for five Feature-Interaction (FI) symptoms. They cover four of the eight types of FIs described in Section II-C, and in particular they cover FIs due to *data-value manipulation* (where one feature's data affects the computations of another feature), *control-flow manipulation* (where one feature's actions affect the control flow in another feature), and *multiple-input interactions* (where a feature reacts to near-simultaneous requests from other features). While these scripts depend on the output from *Rex*, they are independent of each other; additional scripts that look for other FI symptoms can be added easily. The factbase can also be extended with additional information as needed; for

[3] The names in the example are simplified to ease exposition. In practice, *Rex* creates long identifier names that capture the entity's context (i.e., enclosing function, class, etc., up to and including filename).

[4] A feature can be added to the factbase manually with relatively little effort, should a domain expert wish to do so.

TABLE II
RELATIONSHIP AND ATTRIBUTE INFORMATION ABOUT *ROS* PROGRAMS PRODUCED BY THE *Rex* EXTRACTOR.

Relation/Attribute	Meaning
publish ⟨*publisher*⟩ ⟨*topic*⟩	⟨*publisher*⟩ publishes data to some *ROS* ⟨*topic*⟩
subscribe ⟨*topic*⟩ ⟨*subscriber*⟩	⟨*subscriber*⟩ subscribes to some *ROS* ⟨*topic*⟩ and receives data from it
mesg ⟨*function1*⟩ ⟨*function2*⟩	⟨*function1*⟩ in some feature sends a message to ⟨*function2*⟩ in another feature
call ⟨*function1*⟩ ⟨*function2*⟩	⟨*function1*⟩ calls ⟨*function2*⟩
contain ⟨*entity1*⟩ ⟨*entity2*⟩	⟨*entity1*⟩ transitively contains ⟨*entity2*⟩; for instance, a class contains a function
write ⟨*function*⟩ ⟨*variable*⟩	⟨*function*⟩ assigns data to ⟨*variable*⟩
varAssign ⟨*variable1*⟩ ⟨*variable2*⟩	⟨*variable1*⟩ assigns its data to ⟨*variable2*⟩
inDecisionCondPub ⟨*variable*⟩ ⟨*publisher*⟩	⟨*variable*⟩ is used in a control-flow statement that affects whether ⟨*publisher*⟩ publishes data
inDecisionCondFunc ⟨*variable*⟩ ⟨*function*⟩	⟨*variable*⟩ is used in a control-flow statement that affects whether ⟨*function*⟩ is called
⟨*variable*⟩ { inDecisionCond = ⟨true/false⟩ }	Denotes whether ⟨*variable*⟩ is used in the decision condition of a control-flow statement

example, information about resources can be added to detect resource contention.

Figure 3 shows a UML class diagram of the entities, relationships, and attributes that must be present in a *ROS*-based automotive TA factbase to detect each of these symptoms. UML-classes[5] are entities in our TA model, associations between classes are relationships, and fields in the classes are attributes. Red classes are *ROS*-based entities. Table II provides a detailed description of the relationships and attributes collected by *Rex*; these relations and attributes are used in the remainder of this section. Essentially, the TA model produced by *Rex* serves as input to the *Grok* analysis scripts that look for possible FIs. The remainder of this section presents and explains these scripts[6].

A. Inter-Feature Communication Loops

A **communication loop** occurs when a feature sends a message that *eventually* causes that same feature to receive a message, and can be a symptom of features that repeatedly respond to each others' messages in an infinite loop. This symptom corresponds to interaction type 8 (communication loops among features) described in Section II-C. In a direct loop, a feature sends a message directly to itself. In an indirect loop, a feature sends a message to another feature, which eventually results in the origin feature receiving a message from some feature.

Figure 4 shows the *Grok* script used to detect the communication loop symptom. In this script sendFunc is a function that sends a message to another component and receiveFunc is a function that receives a message from another component:

1) Compute the identity relation of the feature set. Store this in a relation called loopTest (line 1).
2) Take the mesg relation and lift it from the function to the feature level by joining it to the contain relation (line 3).
3) Compute direct loops by taking the intersection of the directFeat relation (all direct communications) and loopTest. Print the direct loops (lines 4–5).
4) For indirect loops, compute the union of the mesg relation and the function call relation and store the result in a relation called fullCall. Get the transitive closure of this relation (lines 7–8).
5) Restrict the resultant relation to only those tuples that start with a sendFunc in the initial feature and end with a receiveFunc in the recipient feature. Then, similar to step 2, take the indirectLoop relation and lift it from the function to feature level. Finally, take the intersection of the results with loopTest to get all indirect loops. Print the results (lines 10–13).

```
1  loopTest = id feature;
2
3  directFeat = contain o mesg o inv contain;
4  directLoop = directFeat ^ loopTest;
5  directLoop;
6
7  fullCall = mesg + call;
8  fullCall = fullCall+;
9
10 indirectLoop = sendFunc o fullCall o receiveFunc;
11 indirectLoop = contain o indirectLoop o inv contain;
12 indirectLoop = indirectLoop ^ loopTest;
13 indirectLoop;
```

Fig. 4. *Grok* script for detecting communication loops.

B. Control-Flow Interactions

A **control-flow interaction** occurs when the behaviour of one feature is altered as a result of messages received from other features. This class of FIs is the most critical to detect, as they can lead to emergent behaviours that are surprising and undesirable. This class consists of two subtypes — behaviour alteration and publish alteration — which are highly specialized, and require information about function calls, variable assignments, and control-flow statements.

A **behaviour alteration** occurs when a message from some initial feature alters the behaviour of a recipient feature. This definition is vague about what it means for a feature to "change behaviour". We define it to be a change in control-flow: that is, a change to a variable value that is subsequently used in the decision condition of some control structure (i.e., a if, for, while, or switch statement). Thus, the pattern looks for a message to a recipient feature that leads to a direct or indirect variable assignment, where that variable is used in a control

[5] One of the boxes represents C++ classes from the target system.

[6] Detailed information about each of these scripts can be found in [17].

structure. This symptom is a type of control-modification interaction (interaction 5 described in Section II-C).

Figure 5 shows the *Grok* script to detect symptoms of behaviour alterations:

1) In a set collect all variables that are used in the decision condition of some control structure. Store the results in `controlVars` (line 1).
2) Union together the `varAssign`, `call`, `write`, and `inDecisionCondFunc` relations to form a master relation of information flows. Store the results in `masterCalls` (line 3). Compute the transitive closure of this master relation (line 4).
3) Restrict the resultant relation to those tuples that start with some `receiveFunc` and end with some control variable. Print the results (lines 6–7).

```
1 controlVars = inDecisionCond . {1};
2
3 masterCalls = varAssign + call + write +
      inDecisionCondFunc;
4 masterCalls = masterCalls+;
5
6 behAlter = receiveFunc o masterCalls o controlVars;
7 behAlter;
```

Fig. 5. *Grok* script for detecting symptoms of behaviour alterations.

A **publish alteration** occurs when a message received by some origin feature causes a recipient feature to alter its message-sending behaviour. Thus, this symptom is also a type of control-modification interaction (interaction 5). Similar to the behaviour-alteration symptom, a feature's message-sending behaviour is changed due to a change in control flow. However, detecting this symptom is more complex since a trace needs to be established from the initial received message to a publish call contained within a control flow statement; the `inDecisionCondPub` and `inDecisionCondFunc` relations are required for this symptom because the query needs to determine which "control-flow" variables are responsible for affecting whether a publish call or function call is made.

Figure 6 shows the *Grok* script to detect symptoms of publish alterations:

1) Union together the `varAssign`, `write`, `call`, `inDecisionCondFunc`, and `inDecisionCondPub` relations to form a master relation of function calls, variable assignments, and influence relations. Store the results in `masterCalls` (line 1). Compute the transitive closure of this master relation (line 2).
2) Restrict the resultant relation to those tuples that start with some `receiveFunc` and end with a message to another feature. Print the results (lines 4–5).

C. Multiple-Input Interactions

A **multiple-input interaction** occurs when a feature receives messages from two or more features, resulting in a possible race condition. There are two interaction types in this class: multiple-publisher and race condition.

```
1 masterCalls = varAssign + inDecisionCondFunc +
      inDecisionCondPub + call + write;
2 masterCalls = masterCalls+;
3
4 pubAlter = receiveFunc o masterCalls o mesg;
5 pubAlter;
```

Fig. 6. *Grok* script for detecting symptoms of publish alterations.

Multiple publishers to the same topic, or publishers to different topics that are subscribed to by the same feature, can be a problem because one publisher may send messages at a higher frequency, drowning out data sent from other publishers. This symptom corresponds to a type of livelock interaction (i.e., interaction type 4 from Section II-C. Figure 7 shows the *Grok* script to detect the multiple-publisher symptom:

1) A helper function `indegree(...)` counts the number of instances that each range element appears in a relation, and generates a new relation of the form ⟨`RANGE_ID`⟩ ⟨`COUNT`⟩. Apply `indegree(...)` to the `mesg` relation (line 1).
2) Filter out any entries where the indegree of a feature receiving messages is less than or equal to 1 (line 2).
3) Restrict the `mesg` relation to only those tuples whose range values are the features that receive messages from more than one publisher (line 4). Print the results.

```
1 cardDirect = indegree(mesg);
2 cardDirect = cardDirect [ &1 > 1 ];
3
4 multiMesg = mesg o dom cardDirect;
5 multiMesg;
```

Fig. 7. *Grok* script for detecting the multiple-publisher symptom.

A **race condition** occurs when two or more features message some recipient feature and multiple callback functions update the same variable inside that recipient. Unlike the multiple-publisher interaction, a race condition includes cases where each communicating feature has its own communication channel, as long as their messages cause updates to the same variable contained inside the recipient feature. Thus, this symptom corresponds to a type of nondeterminism (i.e., interaction type 1 from Section II-C.

Figure 8 shows the *Grok* script to detect symptoms of race conditions:

1) Restrict the relation `write` to just those tuples whose domain values are `receiveFunc`s that receive messages from other features. Store the results in `callbackWrite` (line 1).
2) Apply `indegree(...)` to the `callbackWrite` relation (line 3) and filter out any entries where the number of functions writing to a variable is less than or equal to 1 (line 4). Store the results in `cardVars`.
3) Restrict the relation `callbackWrite` to only those tuples whose range values are the variables being written

to by more than one callback function (line 6). Print the results.

```
1 callbackWrite = receiveFunc o write;
2
3 cardVars = indegree(callbackWrite);
4 cardVars = cardVars [ &1 > 1 ];
5
6 callbackWrite = callbackWrite o dom cardVars;
7 callbackWrite;
```

Fig. 8. *Grok* script for detecting the race-condition symptom.

V. Case Study

To evaluate the effectiveness of our methodology, the FI patterns we defined, and the *Rex* extractor, we tested our toolchain on automotive software called *Autonomoose* developed by the University of Waterloo Intelligent Systems Engineering (WISE) Lab. The *Autonomoose* software runs on a modified Lincoln MKZ that aims to eventually operate at level 4 of the SAE autonomous driving standard [19]. This software system consists of features, implemented as components, that each receive data from sensors and other features, perform some task, and then pass data to other features or vehicle actuators. To facilitate communication between features, *Autonomoose* uses the Robot Operating System (*ROS*) architecture. Table III shows some statistics about the *Autonomoose* codebase.

TABLE III
Source code statistics for the *Autonomoose* project as of October 2017.

Autonomoose source code statistics	
# of source code lines in C/C++ (SLOC)	74,215
# of features	14
# of functions	1,298
# of variables	5,321
# of communication channels	14

The goal of this case study is to evaluate the effectiveness and accuracy of detecting FI symptoms using our relational-algebra toolchain. We investigate three research questions:

RQ1 What is the precision/recall of entities, relationships, and attributes generated by the *Rex* fact extractor before relational manipulation?

RQ2 What FI symptoms appear in *Autonomoose*?

RQ3 How effective is each symptom for indicating potential Feature Interactions?

A. Setup

To detect potential FIs in *Autonomoose*, we incorporated *Rex* into *Autonomoose*'s build chain to generate a TA factbase of *Autonomoose*'s entities, relationships, and attributes. Specifically, we generated compilation databases that contained the required compile flags for the project so that the *Rex* extractor could successfully generate an AST for each file. Factbases for each file were extracted and linked to form a single factbase that represents all of *Autonomoose*'s features. The model was then loaded into *Grok* where each symptom script was run to look for possible FIs.

B. RQ1: Fact Extractor Validation

Because the accuracy of our approach depends on the quality of the extraction tools — i.e., both *Rex* and the compilation database — we performed a manual validation of the *Local Planner* module, the largest module in *Autonomoose*. Two members of the research team independently performed a manual fact extraction from the source code of the module, and used scripts to randomly select entities to compare against the *Rex*-generated factbase. Table IV summarizes the results; it shows the number of facts selected randomly for comparison from the manually extracted factbases, along with the corresponding precision and recall. Elements marked with an asterisk denote cases where all entities of that type were counted in the entire software project. Overall, twelve of the sixteen fact types presented in the table have a precision of 100%, and ten of the sixteen have a recall of 90% or higher; we note that because we aim to discover all *potential* FIs, we value high recall over high precision.

AST-based C++ entities, such as `features`, `classes`, `functions`, and `variables`, are collected directly from the *Clang*++ AST or compilation databases; they all have a precision and recall of 100% because *Rex* simply adds information about the associated AST nodes directly to the factbase. The exception to this is `enum classes`, which are not detected by *Rex* and comprise the two classes *Rex* did not detect. For *ROS*-based entities, such as `publish` and `subscribe` objects and *ROS* `topics`, the precision and recall was 100% except for topics, which had a recall of 95.9%. This is because *ROS* topics are created and named using strings and automatically determining the contents of these strings statically is impossible; *Rex* is able to correctly create topic entities only if topics are created using string literals.

For relations recorded in the factbase, with the exception of `call` and `contain`, *Rex* must statically interpret C++ expressions to generate information for that relation. Although all of these relations have a precision over 85%, the `varWrite` relation has a recall of only 66.3%. This is because C++ variable aliasing makes it impossible to statically determine whether a variable has actually been assigned a value. As future work, it would be beneficial to improve this relation to emphasize false positives over negatives so that there is little risk of missing important variable assignments. Relationships involving functions such as `call` and `indDecisionCondFunc` have seemingly very low recall, less than 30% for both. This is primarily due to the fact that both system libraries and core *ROS* libraries were purposely excluded from extraction. Symbols in the abstract syntax tree generated from included system header files and *ROS* libraries were not included in this analysis because execution paths that traverse into system header files are not likely to re-emerge back in the target source file.

For the sole attribute, `inDecisionCond`, *Rex* had a precision of 88.5% and recall of 84.9%. The precision and recall of this attribute are not perfect because *Rex* must extrapolate

TABLE IV
PRECISION AND RECALL OF ENTITIES, RELATIONS, AND ATTRIBUTES IN THE *Autonomoose* MODEL.

Element Type	Source Code	TA Model	Precision	Recall
`Feature` Entities*	14	14	100.0%	100.0%
`Class` Entities	54	52	100.0%	96.3%
`Function` Entities	109	109	100.0%	100.0%
`Variable` Entities	193	193	100.0%	100.0%
`Publisher` Entities*	52	52	100.0%	100.0%
`Subscriber` Entities*	46	46	100.0%	100.0%
`Topic` Entities*	74	71	100.0%	95.9%
`Publish` Relation*	60	57	100.0%	95.0%
`Subscribe` Relation*	46	43	100.0%	93.5%
`Call` Relation	106	30	100.0%	28.3%
`Contain` Relation	260	258	100.0%	99.2%
`Write` Relation	92	81	100.0%	88.0%
`VarWrite` Relation	101	67	94.2%	66.3%
`InDecisionCondPub` Relation	13	9	96.2%	69.2%
`InDecisionCondFunc` Relation	80	21	86.9%	26.3%
`InDecisionCond` Attribute	152	129	88.5%	84.9%

meaning from the AST to generate this relation. In complex control-flow statements, *Rex* is unable to determine whether a variable affects the decision condition or is simply part of the condition. As an example, in the statement `if (var = func(...))`, although `var` is part of the statement, it does not affect whether the `if` branch is taken.

C. RQ2: FI Symptoms in Autonomous

Table V shows the number of warnings generated for the *Autonomoose* project for all symptoms. For each symptom type in the table, the number of warnings are shown along with the number of features that directly and indirectly cause this symptom to occur. Overall, with exception to the multiple-publishers symptom, there is at least one warning of each symptom detected.

For communication loop symptoms, there is one direct communication warning. The lack of indirect communication loops is expected because the *Autonomoose* project is organized to pass data downstream where inputs from sensors eventually flow to vehicle actuators. It is interesting that there is a direct communication loop in one feature; this self-loop is likely used as a heartbeat message for timing or debugging purposes.

For multiple-communication symptoms, there are no warnings of the multiple-publisher symptom and eleven race condition warnings. The lack of warnings for the multiple-publisher symptom is encouraging because it shows that the developers of *Autonomoose* felt that having different features communicating on the same channel could result in errors or add unnecessary complexity.

Finally, for the control-flow symptoms, there are 1,368 warnings of behaviour-alteration and 64 warnings of publish-alteration. The number of warnings for both symptoms is high because *Autonomoose* features change their behaviour when they receive a message causing them to change state or pass data to other features. The number of reported warnings for this symptom is problematic because it is likely that many of these warnings are false positives. To combat this, future work should explore developing a method to filter warnings based on predefined patterns or should allow developers to triage warnings in order of severity.

D. RQ3: FI Symptom Validation

We validated the results by having one of the authors manually classify each reported warning into one of three categories: *impossible*, *unlikely*, and *probable*. *Impossible* warnings are those that are reported as being part of that symptom but are not; their existence could be due to errors in the TA model (as highlighted in Table IV) or are cases that are false positives. *Unlikely* warnings are those that might cause an unintended interaction but are unlikely to do so. Finally, *probable* warnings are those that are most likely to result in potential interactions, and are deemed worthy of further inspection. To provide external validation of the symptoms generated, Dr. Michal Antkiewicz, a research engineer of the *Autonomoose* project, independently classified ten randomly selected warnings from each symptom into the same three categories.

Table V shows the number of warnings classified as *impossible*, *unlikely*, and *probable* for the race condition, behaviour-alteration, and publish-alteration symptoms. For the race condition symptom, of the eleven warnings reported, eight are classified as *impossible*, one is classified as *unlikely*, and two are classified as *probable*. The *impossible* warnings are classified as such because the features participating in the race condition all write the same value to the variable in question; thus, there is no conflict about the variable's eventual value. The single *unlikely* warning is classified as such it involves multiple features that update a variable with the current time. This variable is incrementally updated and the value assigned never depends on the variable's previous value, thus assignments are unlikely to result in an unintended interaction. Finally, the two *probable* warnings are classified as such because the variables that are part of the race condition end up altering the state that the feature is in. In other words, each initial feature that sends a message to this recipient feature eventually affects the state the recipient feature is in.

TABLE V

REPORTED SYMPTOM WARNINGS AND THEIR ASSOCIATED SEVERITY IN THE *Autonomoose* PROJECT.

Symptom Type	**Total Results**	**Direct Results**	**Indirect Results**	**Impossible**	**Unlikely**	**Probable**
Communication Loop	1	1	0	0.0%	0.0%	100%
Multiple-Publishers	0	0	0	—	—	—
Race Condition	11	3	1	72.7%	9.1%	18.2%
Behaviour-Alteration	1,368	11	16	74.6%	18.6%	8.9%
Publish-Alteration	64	7	15	18.7%	42.2%	39.1%

When all eleven warnings were presented to the *Autonomoose* developer, their classifications were the same as ours.

Of the 1,368 behaviour-alteration warnings detected, 1,021 are classified as *impossible*, 225 are classified as *unlikely*, and 122 are classified as *probable*. Of the 1,021 *impossible* warnings, 17% result from the use of *ROS*-based logging functions for debugging purposes and do not impact the feature itself. The other *impossible* warnings occur from variable assignments to local variables that simply change the local function's behaviour. For the 255 *unlikely* warnings, the majority are classified as such because they cause a local variable to change its value which goes on to *eventually* cause a global variable to change its value. For the 122 *probable* warnings, these cases involve a global variable changing its value which then affects how the entire feature operates. Although many of these warnings are intended interactions, their presence warrants a further look at the source code.

When ten random warnings were shown to the *Autonomoose* developer, they were all deemed to be uninteresting because it is expected that a feature sending a message to another feature will affect the variable assignments inside the recipient feature. Although the developer was shown only a tiny fraction of the warnings, their feedback suggests that it could be useful to rank reported warnings according to the "length" of the communication path. Such a ranking would highlight warnings where a feature action has an effect in another feature with which it does not directly communicate.

With 1,368 reported warnings but only 10 random instances shown to developers, this conclusion reinforces the need to further refine this symptom or introduce a filtering mechanism to attempt to reduce the number of reported results. As future work, this can be done by implementing a triage system for developers where warnings are sorted by "severity" by using the length of the information flow path in the warning. The motivation behind this is that warnings with shorter paths are likely to be known to developers and would not be as useful as warnings with long, complex paths through multiple variables and functions.

Of the 65 publish-alteration warnings, 12 are classified as *impossible*, 27 as *unlikely*, and 25 as *probable*. Similar to the behaviour-alteration warnings, the 12 *impossible* publish-alteration warnings are classified as such because they involve cases where debug information is published to the *Autonomoose* developers when certain messages are received. Because these do not alter how the feature behaves, they can be ignored. The 27 *unlikely* warnings have no invoking feature. In other words, although the recipient feature publishes to another feature when it receives a message, there is no feature that sends the initial message to that recipient feature. Instead, the recipient feature receives messages from vehicle sensors rather than other features. Future work could improve the publish-alteration symptom to filter out these warnings. Finally, *probable* warnings represent publish calls that result when a recipient feature receives a message from another feature and sends important data to other features. This could include route information or processed sensor data.

When ten random warnings were shown to *Autonomoose* developers, they expressed the most interest in warnings consisting of a long trace from start to finish. This meant that if a recipient feature received a message from some origin feature, interesting cases would involve a long chain of function `calls` and variable assignments that would *eventually* cause a message to be sent to a third feature. In these cases, it would be likely for a developer to miss these Feature Interactions. Although our classifications of the ten warnings differed, of the ten shown to *Autonomoose* developers, six were categorized as *probable* and four as *unlikely*. Four of the six warnings they classified as *probable* were classified as *unlikely* by us; since each of these four instances involved a long, complex trace, the developers deemed this useful to investigate further.

VI. LIMITATIONS

The primary limitation of this work comes from the use of static analysis to analyze software artifacts. Although static analysis is effective at detecting built-in language constructs, such as `functions` or `class`es, it has difficulty determining non-AST-based relationships between entities such as variable assignments, which results in imperfect precision and recall of the extracted entities and relations. Because this approach aims to detect *potential* Feature Interactions (FIs), developed extractors should aim to generate models with high recall. This emphasis sometimes impacts the precision of the extraction or analysis, such as with behaviour-alteration symptoms. In future work, we plan to study the causes of imprecision and devise ways to mitigate these causes.

A limitation that is specific to statically analyzing *ROS* code is the linking of features' names for shared communication channels. When a publisher or subscriber object is created in a *ROS* package, the constructor for that object requires a programmer to specify the name of the communication channel as a string. The problem is that topic names can be passed to this constructor as string variables or literals. Statically determining the contents of a variable is generally impossible; in such cases, *Rex* simply takes the variable name or string-literal value and uses that as the communication channel name. Therefore, if two features that communicate via the same channel use different variable names for the respective publisher/subscriber constructor, *Rex* will be unable to link these features' communications.

A. Threats to Validity

There are several threats to the validity of our results. One internal threat comes from the subjectivity of manually calculating the precision and recall of the *Rex*-generated TA models and from classifying symptom instances. As these tasks were manually done by individuals on our team, bias might be introduced into the results. We aimed to mitigate this bias by having domain experts for the *Autonomoose* project examine the symptoms that were deemed the most severe and classify them.

Second, the results we present are not necessarily applicable to other software systems or non-*ROS*-based automotive systems. In particular, the extraction of facts related to inter-component communications may be more complex for automotive software built on the *AUTOSAR* framework. Future work will aim to generalize our research to *AUTOSAR*-based software, other message-passing communication APIs, and other application domains.

A related threat to generality is that features themselves are not generally statically identifiable in source code. In our current work, we assume that features are implemented as distinct components, which are identifiable. In other systems, we may need other means to identify features (e.g., code that is guarded by feature-specific conditional-compilation directives, or an input mapping of features to functions).

VII. Related Work

There is a rich body of literature on detecting potential Feature Interactions among features, services, or components, that is summarized elsewhere [5] [3] [20] [21]. Approaches to detecting FIs depend heavily on how features or components are represented [5], and many approaches require some specification of feature properties or expected feature behaviour, against which to compare actual system behavior. Moreover, approaches based on dynamic analyses and verification are computationally expensive and do not scale to programs with lots of features. [21].

The works closest to ours are those that extract information from software artifacts to detect interactions amongst components. JITANA [22] analyzes Android applications to determine whether inter-component calls (ICC) exist between separate applications. Although legitimate communications exist, these results can be investigated by other means to identify malicious or invalid communications between programs. Purandare et al. [23] present an approach to extract information from *ROS* projects to detect dependencies amongst components, and to determine how specific changes to code segments affect different components. DEvA [24] uses static analysis to detect event anomalies in event-based systems; these anomalies are defined as two events that access the same fields where at least one event performs a *write*. VarXplorer [25] analyzes configurable software, looking for discrepancies in data flows and control flows among features in different configurations. Some of these works (e.g., [23]) have created extraction technologies comparable to ours. However, their approaches each detect a single type of interaction symptom (e.g., inter-component communications [22] [23], potential write conflicts [24]) and are hard-coded, whereas our approach to expressing symptoms of interactions is programmable, making it easier to express multiple types of interaction symptoms.

There has been other work that has used a form of relational algebra to verify software. Kozen [26] use a type of algebra called Kleene Algebra with Tests (KAT) to automatically verify the correctness of programs. Although the author proves that KAT is effective in verifying safety policies, it is very labour intensive. Generating a KAT encoding of a code fragment needs to be done manually and assertions need to be injected into the representative KAT encoding to test correctness properties. As such, applying KAT to a large-scale automotive system would be difficult.

VIII. Conclusion and Future Work

The aim of our work is to develop techniques that complement current approaches to detecting Feature Interactions (FIs), and that we intend to be scalable to large software systems. We have presented a static-analysis toolchain that can detect potential FIs in a message-passing, automotive software system. Specifically, (1) we presented a tool called *Rex* that is capable of generating queryable models of *ROS*-based projects, (2) we identified five symptoms of potential FIs that could be expressed as relational-algebra queries on these models; and (3) we conducted a case study on the *Autonomoose* project to test the effectiveness of *Rex* and the symptom definitions.

We see two major areas of future work. First, symptom definitions need to be refined and expanded to detect more types of FIs, and to filter extraneous results to reduce the number of reported false positives. Based on our analysis of false positives from our case study, we have initial ideas about how to mitigate against some instances; and in other cases, we might be able to rank warnings in order of usefulness to the engineer.

Second, we need to expand our extractors to operate on other types of software, such as automotive software that uses the *AUTOSAR* platform [27] and controller area network (CAN) bus [28] communications. We also plan to evaluate the tools on large open-source software systems, in order to assess the generality of this work to other application domains and to assess its scalability to large systems.

Acknowledgment

The authors would like to acknowledge Dr. Michal Antkiewicz and the other developers from the *Autonomoose* project for providing us with access to the *Autonomoose* source code and for their time spent assisting us with symptom validation.

References

[1] C. Ebert and J. Favaro, "Automotive Software," *IEEE Software*, vol. 34, no. 3, pp. 33–39, May-Jun. 2017.

[2] E. Cameron, N. Griffeth, Y. Lin, and H. Velthuijsen, ""Definitions of Services, Features, and Feature Interactions"," December 1992, bellcore Memorandum for Discussion, presented at the International Workshop on Feature Interactions in Telecommunications Software Systems.

[3] D. O. Keck and P. J. Kuehn, "The feature and service interaction problem in telecommunications systems: A survey," *IEEE Transactions on Software Engineering*, vol. 24, no. 10, pp. 779–796, Oct. 1998.

[4] A. L. J. Dominguez, "Feature interaction detection in the automotive domain," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 2008, pp. 521–524.

[5] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks*, vol. 41 (1), pp. 115–141, 2003.

[6] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of Feature Interactions Using Feature-aware Verification," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11, 2011, pp. 372–375.

[7] S. Apel, A. Von Rhein, T. ThüM, and C. KäStner, "Feature-interaction Detection Based on Feature-based Specifications," *Comput. Netw.*, vol. 57, no. 12, pp. 2399–2409, Aug. 2013.

[8] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[9] R. Mckenzie and J. Mcphee, "Research and educational programs for connected and autonomous vehicles at the university of waterloo," *Mechanical Engineering Magazine Select Articles*, vol. 139, no. 12, pp. S21–S23, 2017.

[10] R. C. Holt, "An introduction to TA: The tuple-attribute language," *University of Toronto, Toronto*, vol. 24, 1997.

[11] ——, "Introduction to the Grok programming language," *University of Waterloo*, 2002.

[12] ——, "WCRE 1998 most influential paper: Grokking software architecture," in *2008 15th Working Conference on Reverse Engineering*, Oct 2008, pp. 5–14.

[13] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a case study: Its extracted software architecture," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 555–563. [Online]. Available: http://doi.acm.org/10.1145/302405.302691

[14] I. J. Davis and M. W. Godfrey, "From whence it came: Detecting source code clones by analyzing assembler," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 242–246.

[15] C. O.-S. Compiler, "Language compatibility." [Online]. Available: https://clang.llvm.org/compatibility.html

[16] Waterloo Software Architecture Group. (2006) Grokdoc. [Online]. Available: http://www.swag.uwaterloo.ca/jgrok/grokdoc/index.html

[17] B. Muscedere, "Detecting automotive feature-interaction hotspots using relational algebra," Master's thesis, University of Waterloo, 2018.

[18] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, "A conceptual basis for feature engineering," *Journal of Systems and Software*, vol. 49, no. 1, pp. 3 – 15, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412129900062X

[19] S. O.-R. A. V. S. Committee *et al.*, "Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems," *SAE Standard J3016*, pp. 01–16, 2014.

[20] S. Apel, J. M. Atlee, L. Baresi, and P. Zave, "Feature Interactions: The Next Generation (Dagstuhl Seminar 14281)," *Dagstuhl Reports*, vol. 4, no. 7, pp. 1–24, 2014.

[21] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, no. 1, Jun. 2014.

[22] Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, and J. Dinh, "An efficient, robust, and scalable approach for analyzing interacting android apps," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 324–334.

[23] R. Purandare, J. Darsie, S. Elbaum, and M. B. Dwyer, "Extracting conditional component dependence for distributed robotic systems," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012, pp. 1533–1540.

[24] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic, "Detecting event anomalies in event-based systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 25–37. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786836

[25] L. R. Soares, J. Meinicke, S. Nadi, C. Kästner, and E. S. de Almeida, "Varxplorer: Lightweight process for dynamic analysis of feature interactions," in *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VAMOS 2018, 2018.

[26] D. Kozen, "Kleene algebra with tests and the static analysis of programs," Cornell University, Tech. Rep., 2003.

[27] F. Kirschke-Biller *et al.*, "Autosar–a worldwide standard current developments, roll-out and outlook," in *15th International VDI Congress Electronic Systems for Vehicles, Baden-Baden, Germany*, 2011.

[28] *CAN Specification*, 2nd ed., Bosch Semiconductors and Sensors, sep 1991.