SALMAN HOSEINI

Software Feature Location in Practice: Debugging Aircraft Simulation Systems

A Thesis

In The Department Of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science in Electrical and Computer Engineering at
Concordia University
Montreal, Quebec, Canada

December 2013

© Salman Hoseini, 2013

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Salman Hoseini

Entitled: Software Feature Location in Practice: Debugging Aircraft Simulation Systems

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. M. Zahangir Kabir _____Chair

Dr. Luis Rodriguez _____Examiner

Dr. Nizar Bouguila _____Examiner

Dr. Abdelwahab Hamou-Lhadj _____Supervisor

Approved by: Chair of Department or Graduate Program Director

January 2014 Faculty of Engineering and Computer Science

ABSTRACT

In this thesis, we report on a study that we have conducted at CAE, one of the largest civil aircraft simulation companies in the world, in which we have developed a feature location approach to help software engineers debug simulation scenarios. A simulation scenario consists of a set of software components, configured in a certain way. A simulation fails when it does not behave as intended. This is typically a sign of a configuration problem. To detect configuration errors, we propose FELODE (Feature Location for Debugging), an approach that uses a single trace combined with user queries. When applied to CAE systems, FELODE achieves in average a precision of 50% and a recall of up to 100%.

Acknowledgements

I want to take the opportunity to express my sincere gratitude to my supervisor Professor Dr. Abdelwahab Hamou-Lhadj who never lost his trust in me. His guidance helped me through the two years of research and whenever we faced any problem he was always a source of inspiration to continue and face the challenges.

Many thanks to the software engineers at CAE who I have had the privilege to work with. My special thanks to Martin Tapp, Gabriel Dubeau, and Patrick Desrosier who helped me throughout my research by being always present to answer my questions while they were busy with their own work.

I would also like to deeply thank CRIAQ (Consortium de recherche et d'innovation en aérospatiale du Québec), NSERC (Natural Sciences and Engineering Research Council of Canada), the Faculty of Engineering and Computer Science at Concordia University, CAE Inc., and Opal-RT for their financial support. This work would not have been possible without them.

I want to also thank my family who continued their never ending support even when I am in the other side of the world.

Lastly I want to thank my colleagues at the Software Behaviour Analysis (SBA) Research Lab who were open to discussions and spent their time helping me when I faced challenges during this research.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1. Introduction to CAE's Simulation System

Simulators play a critical role in the aircraft industry. They are used for many purposes including pilot training, aircraft design, and quality assurance. To simulate various features of an airplane, CAE, the company in which this study was conducted, is heavily invested in the development of aircraft simulation software systems. These systems are modular and component-based by design. They are composed of several software subsystems (that we refer to as *modules* throughout this thesis)–each responsible for a particular simulation function. Almost every function of an airplane is simulated through a software module.

Modules are combined to simulate complex scenarios. An example of a simulation scenario is depicted in Figure 1, where an aircraft is descending at high speed while flying at low altitude. To avoid a crash, a successful simulation is the one in which the system generates proper warnings and alarms to inform the pilot. A simulation is saved in a configuration file, which contains mainly the modules and the connections among modules.

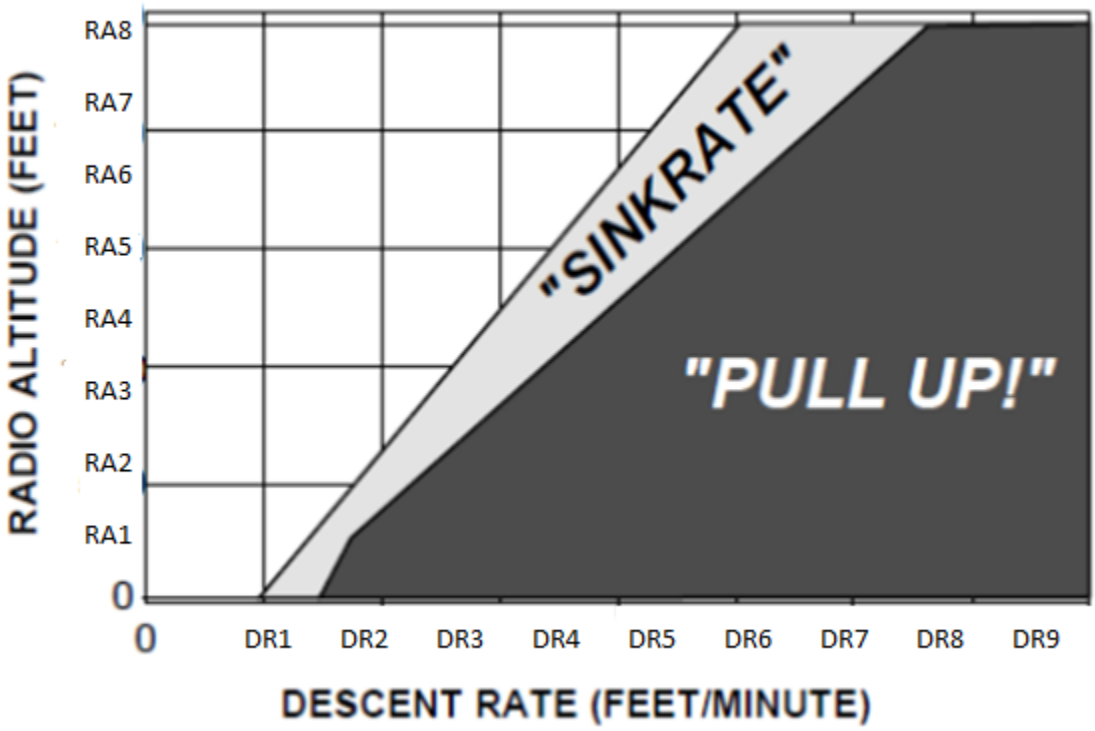

Figure 1. Taws Mode1 Envelope

At CAE, it is the responsibility of integration specialists with the help of multi-disciplinary teams (that we refer to collectively as *configuration designers*) to design and execute simulation scenarios. Configuration designers are software engineers, but not necessarily the ones involved in the development of the modules. In fact, they do not have to know much about the modules except their functionality, as well as what they take as input and provide as output.

The only way for modules to communicate with each other is through exchange of data stored in a common database. The motivation behind this design is to enforce the low coupling, high cohesion principle, hence enabling reuse of modules for the generation of

other simulation scenarios. It also makes communication among modules transparent. This is particularly important in the context of CAE so as to meet the applicable regulations on flight simulators. Through this thesis we refer to the data stored in the common database as *labels*. One can think of labels as messages exchanged among processes in a distributed architecture.

The generalized system architecture of CAE's simulation system is shown in Figure 2.

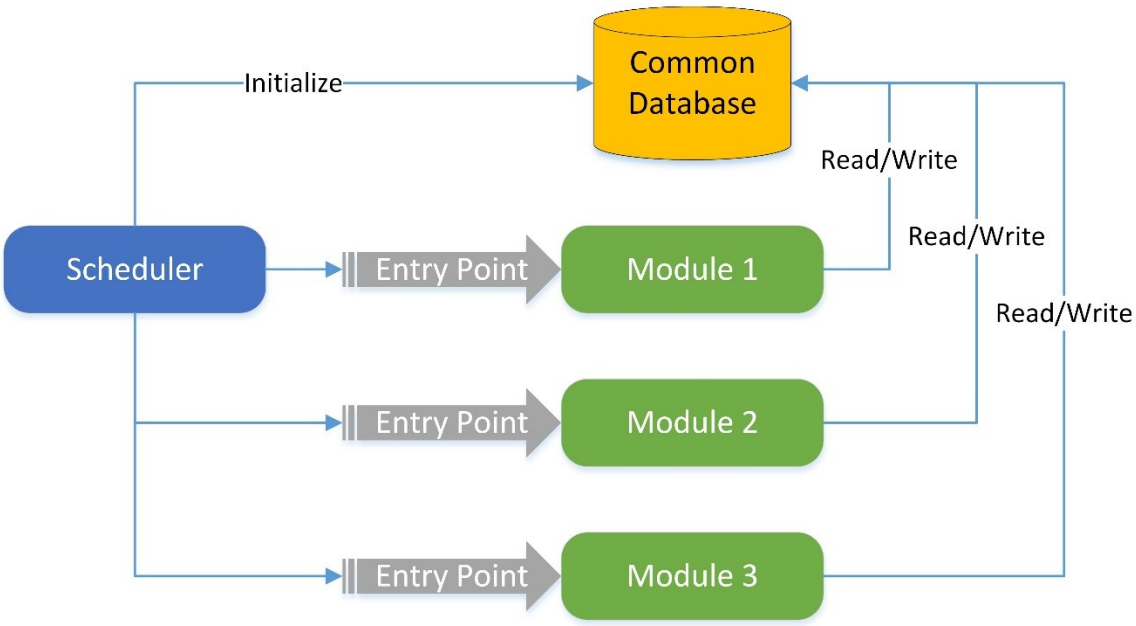

Figure 2. Generalized System Architecture

The role of the scheduler is to invoke the modules in a certain order depending on the objective of the simulation. Each module has an entry point that is used by the scheduler. The scheduler uses proprietary algorithms to synchronize the modules to meet the requirements of a given scenario. These algorithms are out of the scope of this thesis.

1.2. Problem and Motivation

At CAE, during the testing process when the simulation does not behave as intended (e.g., wrong or no warnings are output when needed), it is an indication of the presence of bugs in the software modules, or configuration errors. In this thesis, we focus on configuration errors only. Configurations problems are costly for CAE as they are found late in the integration process. Having new methods to better find the root causes helps reduce costs.

At the present time, the common approach for uncovering causes of invalid behaviour at the configuration level is by browsing configuration files searching for clues that could point out defects such as improper connection among modules. Given the large number of modules involved in a typical simulation scenario, this process is time-consuming, error-prone, and requires heavy involvement of domain experts.

In the following Section we present a sample simulation scenario and will explain the problem through exercising the scenario.

1.2.1 Simulation Scenarios

In designing a simulation scenario, the main steps are (1) determine the list of required modules, (2) enable communication among modules, and (3) execute and test the simulation.

Examples of modules involved in the scenario of Figure 1 include TAWS (Terrain Awareness and Warning System) and NAV (Navigation System). TAWS is a subsystem of a larger (and perhaps most important) system, called FSS (Flight Surveillance System).

TAWS generates alarms and warnings to inform the pilot of the terrain conditions (e.g., an audio sound when the terrain is too low). NAV is responsible for keeping track of the aircraft's positions using latitude, longitude, altitude, and angle in horizon.

Once the design of the simulation scenario is completed, the execution starts. For this, a different set of tools is used, among which the one related to this study is the *monitor*.

The monitor is used by configuration designers to test the simulation. It exhibits the status of each module during execution of the scenario. It also displays notification messages such as warnings and alarms. For example, monitoring the behaviour of the system under the condition shown in the dark gray area in Figure 1 will trigger the monitor to output an alarm indicating that the plane is flying at high speed and low altitude, meaning that there is a risk of a crash.

Simulation errors occur when the monitor omits to display important warnings or displays the wrong information. Many of these failures are due to configuration errors such as assigning labels to the wrong variables or even the wrong modules. One of the main reasons behind these failures is due to the way modules are connected. To debug these errors, configuration designers need to find places in the configuration files where the connections are improperly set.

Typical simulations contain hundreds if not thousands of labels; not all of them are, however, relevant to the observed failure. A technique that can automatically point out these connections will save time and effort spent on debugging complex simulations. Configuration designers can then focus on simulating new and interesting scenarios instead of fixing existing ones.

To address this issue, we propose FELODE (Feature Location for Debugging), a semi-automated approach that combines a single trace and user feedback to locate the connections among modules that are most relevant to the observed failure.

1.3. Research Contribution

The main contributions of this thesis are as follows:

- Application of feature location to an industrial system. To our knowledge, this is the first time that feature location is applied to the flight simulation domain. Also, through our review of the literature, we have not encountered studies that involve industrial systems. Existing techniques have been mainly applied to open source.

- The FELODE approach itself which relies on a two-phase process that detects *only* the components that caused the invalid behaviour. Existing feature location approaches are designed to identify *all* the components that are relevant to the traced feature no matter if they are related to the failure or not. We believe that these techniques are most suitable to feature enhancement tasks and general understanding of the feature implementation. FELODE, on the other hand, is more focused on debugging tasks.

- By locating features in configurations files, we demonstrate the applicability of feature location principles to other software engineering artefacts rather than the source code.

1.4. Thesis Outline

The rest of the thesis is structured as follows:

- In Chapter 2, we present the categories of feature location techniques (based on the terminology used in [Dit13]). We go through previous work in the area of feature location and discuss the techniques used to perform feature location. We discuss the limitations and challenges of each approach.

- In Chapter 3, we introduce our approach for locating simulation scenarios in configuration files. We first present a general overview of the approach and then discuss its components. We also discuss the challenges we faced during our analysis process.

- We show the evaluation of our approach on CAE's simulation system in Chapter 4. In Chapter 4, we apply the approach to several simulation scenarios. We introduce the scenarios we used and discuss how data in configuration files affects the scenario execution.

- In Chapter 5, we conclude the thesis with a summary of the main contributions and the future directions.

Chapter 2 Background

2.1. Feature and Feature Location

In a software system, a feature corresponds to a system functionality as defined during software specification [Dit13]. As an example, in a drawing application, drawing a rectangle or saving the content of a palette are both features of the application We can consider a software system as a collection of features which are accessible by users [Dit13]. Although in the software specification all functionalities are defined in some specific order, this is not necessarily the case in the implementation process. Many features spread all around the source code. A feature may be implemented as a simple function or as a collection of functions working together to implement the functionality. In most cases, a feature is not just a block of statements, but a group of related program elements consisting of classes, methods and variables. These program elements may not be in the same package (or namespace) or not even in the same project. As a result identifying features in a software system requires thorough investigation of the code either using static or dynamic analysis techniques [Dit13].

Feature location research consists of a set of techniques to identify the software elements that are most relevant to the implementation of a specific feature. The ideal feature location technique would be the one that detects all software components and only these components that are most relevant to the implementation of the feature. Thus, when a change in a feature is required, the developers will know that they need to modify only the detected components.

2.2. The Role of Feature Location in Software Maintenance

Software maintenance tasks fall into two main categories: “adding new functionality to the system (perfective maintenance) or removing unwanted functionality (corrective maintenance)” [Dit13]. As a result, performing maintenance tasks requires that the source code components related to the requested change first be located. The idea situation would be to use system documentation to identify these components. Unfortunately, it has been shown in practice that keeping good documentation is impractical. Most existing systems are poorly documented if documented at all.

One common activity to obtain adequate understanding of the task is to detect automatically the functionality implementations in the source code [Dit13]. This is known as locating features in the source code.

Feature location can significantly reduce the cost of program comprehension process because it provides a starting point for users who are assigned to perform the maintenance task. In other words, it gives the user the option of investigating only a small subset of the software instead of going through the overall system [Rohatgi08].

For example if a user is assigned to fix a bug in the system, he would have difficult time to first find out where to begin. This is because he does not know what parts in the source code are responsible for producing the bug. To perform the task he would have several options available; he can debug the application, but for multi-threaded applications it can be a frustrating task. The other option is to do a manual investigation through source code to find where the bug occurs, but this one is also a time consuming and error prone approach. He would have other options like asking more experienced developers but that

may not work in cases in which others are busy with their own tasks. There is clearly a need for automatically identifying these components, which is the objective of feature location techniques.

2.3. Feature Location Techniques

There are several feature location techniques (see [Dit13] for a survey). These techniques use different sources of information. Some consider only the source code of the system and by analyzing the dependencies between different parts of the source code they detect the source code components that are most relevant to the implementation of the feature under study (e.g., [Chen00, Robillard02, Robillard05a]). The second category of techniques combines the source code with documentation (e.g., [Petrenko08, Marcus04, Poshyvanyk07b]). The other set of techniques focus solely on dynamic analysis (i.e., the use of run-time information) ([Wilde92, Wong99, Eisenbrath01b]).

2.3.1. Static Analysis Techniques

Static analysis techniques focus on analyzing the source code such as control flow dependencies and data dependencies. The common approach can be described as follows: Having an element or a set of elements which are related to the feature (initial elements), the additional related elements will be detected by following the dependency flow of the initial elements. The initial elements are specified by the user either based on his/her prior knowledge or by assumptions [Chen00, Robillard05a, Saul07]. Static analysis techniques attempt to discover more elements in the source code using the information found in the structure of the source code such as program dependency graphs [Chen00, Rohatgi08, Rohatgi09].

Chen et al. proposed an approach where they used the static dependencies between program elements to obtain the feature-relevant elements [Chen00]. They introduced the concept of Abstract System Dependency Graph (ASDG) which is composed of methods and global variables of the source code. The approach starts with a node, known to be relevant to the feature. This node is either selected by the user or chosen randomly. The next node in the graph is then presented to the user. The user decides whether the visited node is relevant to the feature or not. At the end of this graph traversal, the collected path contains the program components that are most relevant to the feature. Unfortunately, this process can be quite time consuming for large systems.

Robillard and Murphy [Robillard02] proposed an approach which uses the structural dependencies between program elements. The authors argued that attempting to find elements implementing the feature at the code level may cause ambiguity. Instead, they suggested using an abstract representation of the code. They introduced a middle presentation, called the *Concern Graph*. A concern is a feature under study, and a concern graph is composed of elements implementing the concern including the relations between them.

Robillard et al. [Robillard05a] proposed a static feature location technique by analyzing the topology of structural dependencies with the objective of producing a suggestion set for the user to analyze the feature. The input of their approach is a set of program elements which consists of methods and fields which are likely to be related to the feature under study. This set is proposed by the user based on domain knowledge. The approach compares the elements of this set to the rest of the program to find more elements which are relevant to the concept. This is done by assigning two metrics to the elements that do

not belong to this set. The two metrics are Specificity and Reinforcement. Specificity measures how specific an element is to the set. Reinforcement measures how strongly an element is related to others elements in this set. Robillard's approach has the advantage of less involvement of the developer, but it is highly sensitive to the first input.

Saul et al. used structural information in the call graph to extract the feature related methods [Saul07]. In this study, the approach starts with one input that is a method called query chosen by the user as feature relevant. Their approach can be considered as an extension of the previous study by Robillard. The difference is that in Saul's approach the provided input is a single method. Then the approach creates the dependency call graph of the neighbour methods of the query. For ranking the methods, the author applies the random walk algorithm to the call graph. To efficiently extract correct methods, the approach produces a sub-graph called base graph, which is comprised of the parents, siblings, spouses and children of the query. Using the base graph, the search for related methods is narrowed down to elements in the base graph. In practice, the base graph can still be large and costly to investigate. Thus the next step in the approach is to rank the nodes to obtain more relevant results using random walk algorithm.

2.3.2. Dynamic Analysis Techniques

In dynamic analysis, the main source of information is the data collected during execution of the system. This data is typically represented as execution traces. An execution trace is the collection of events that are triggered during run time. Depending on what needs to be observed, a trace event could be a routine call, a statement, etc. The trace information is collected in a file which is called the trace file. To obtain an execution trace, the software under study needs to be instrumented. There are different

methods to instrument the system; inserting probes in the source code, binary file instrumentation, and profiling to name such techniques.

When the user defines an execution scenario in which the desired feature is exercised, the resulted trace would contain the program elements corresponding to the feature. The choice of execution scenario affects significantly the quality of results.

The important challenge of working with execution traces is the size of the trace files. Execution traces tend to be large. The execution of a scenario results in a trace that contains many program elements; not all of them are necessarily most relevant to the implementation of the desired feature. Examples include utilities [Rohatgi09].

Wilde et al. proposed an approach to locate feature specific program elements using execution information of the program [Wilde92]. Their approach, called Software Reconnaissance, uses two sets of traces. The first set is generated by exercising the feature of interest. The second set is generated by exercising different features. They compared the two set of traces and extracted the trace components that are most relevant to the feature of interest. Their approach was evaluated on small applications and showed good results.

Wong et al. proposed an approach to collect program elements at a finer level of granularity. In their study, they attempted to detect statements and basic blocks [Wong99]. The approach takes as input several test cases divided into two groups: Test cases that invoke the feature under study and the ones that do not. Similar to Wild et al. Wilde92, Their approach compares the traces generated from the two sets of test cases to detect the program elements that are most relevant to the features.

Eisenbarth et al. used concept analysis to detect feature relevant program elements [Eisenbarth01b]. The authors created a concept lattice by having the features of the system as attributes and the program components as concepts. While Eisenbarth's approach is purely based on dynamic analysis, it suffers from the problem of carefully selected the execution scenarios. Their approach also requires heavy intervention from the user to navigate the concept lattice to detect manually the feature-relevant components.

Bohnet et al. [Bohnet08b] proposed an approach with which the user can have different perspectives of the execution trace. In this study, the authors implemented a visualization tool which integrates multiple views of the system. Using this tool a developer can explore the obtained trace in different levels of detail. For example, the tool offers an overview of the execution trace for the developer to obtain an initial understanding of the stages of the execution (initialization, termination etc.). One of the main features of the tool is the synchronization between all the views. One can investigate the executed code by knowing where in the trace this code is executed or in which stage of the execution. The advantage of their proposed approach is that it can manage the complexity of the unfamiliar system and also the large amount of information in the execution trace.

Hayashi et al. [Hayashi10a] used the combination of dynamic and static techniques. Their approach takes three inputs which are: a test case (in order to extract the execution information), the source code, and a user query. The approach starts with the user formulating a query. Then a score is assigned to each routine based on the similarity of terms in the query and terms in the routine, the user is asked to verify the highest ranked routines and using the static dependencies, the dependent routines will obtain higher

scores. The feedback process helps the user detect relevant routines which might have obtained a low score using the similarity measure. The idea of this iterative approach is that in the process of detecting the elements related to the feature under study, the user understands more about the feature implementation and can detect dependent elements.

2.3.3. Textual Techniques

Textual feature location techniques do not require system execution. They also do not use any information regarding data or control flow dependencies. These techniques use words and text used in the body of the source code to obtain knowledge about the implementation of different features. There are several key techniques associated with textual analysis, like pattern matching, Information Retrieval (IR) and natural language processing (NLP). Pattern matching is basically a search for terms in a body of text. IR and NLP are more advanced techniques.

The idea of textual analysis is that identifiers and comments used in the body of the source code embed the domain knowledge [Dit13]. Textual techniques extract the program elements using textual descriptions of the feature in the code. This is possible with the assumption that the feature is implemented using similar set of words used in the comments and identifiers [Dit13]. Textual analysis mostly use a query formulated by the user as input [Marcus04, Shepher'06]. A query is simply a string input by the user. It is composed of terms which describe the exercised feature. Depending on the technique used in the analysis, the query and the knowledge source (i.e. the extracted words and text from source code) will be analyzed differently. The basic idea is to find similar expressions in source code and the query, and the result is a set of retrieved program elements which have similar words in their body with the terms in the query.

The most important aspect of using textual analysis techniques is to formulate a query which can lead to the detection of valid results. To do so, the user should be able to describe the feature under study using correct terms. The effectiveness of textual feature location techniques is heavily relied on the quality of naming in the source code.

Petrenko et al. [Petrenko08] conducted a study in which they used textual information in the source code to find feature-relevant elements. In their study, they introduced the concept of ontology fragments which encapsulate the user's partial understanding of the feature under study. The approach begins with the initial ontology fragment created by the user with his initial knowledge about the feature. Based on the ontology fragment, the user formulates a query. The user can enhance the query and improve the ontology fragment. Petrenko's approach is a continuous approach, which in every step the ontology fragments becomes richer and thus gives the user more knowledge to produce better queries.

Marcus et al. used a more advanced textual technique in their study [Marcus04]. They applied IR to textual descriptions in the source code to extract domain knowledge. Also, they used Latent Semantic Indexing (LSI) to find mapping between natural language description of the feature and the relevant parts in the source code. The approach starts by collecting all the identifiers and comments in the source code and creating a corpus. The corpus is a collection of documents which each document depending on the level of granularity can be a class or a method. The terms used in the body of the element are stored in a vector. The approach then continues with user generating a query describing the feature. The query itself is transformed into a vector. The suggested program elements are collected using the comparison between the query vector and vectors in

corpus. The result is a set of program elements which had more similarity with the query in terms of words used in their body.

Poshyvanyk and Marcus in [Poshyvanyk07b] enhanced the approach introduced in [Marcus04] by adding the Formal Concept Analysis (FCA) to the process. In this approach the concept analysis matrix receives the input from LSI approach. The objects in the matrix are the program elements (methods) and the attributes are the terms used in the textual definition of the method. The approach searches for the top *n* ranked methods by LSI and then applies FCA on them.

Single Trace and Information Retrieval (SITIR) [Liu07] use a single feature executing scenario and the textual information to detect the feature relevant elements. Using IR, a corpus of documents comprised of comments and identifiers in the source code is created. SITIR starts with one execution trace from one feature specific scenario and then applies IR on the program elements appeared in the trace. For this purpose, the user inserts a query and based on the similarity between the query and textual information of elements in the trace it ranks the results to extract most relevant elements of the feature.

Revelle et al. [Revelle10] proposed an approach in which they used techniques for analyzing the structure of World Wide Web (WWW) to locate features in the source code. They used HITS [Kleinberg99] and PageRank [Brin98] as methods to filter out relevant results from the execution trace. Web analysis techniques can be used directly on the trace to locate feature relevant elements. For example, using HITS one can locate relevant methods which have many calls to other methods, making those methods relevant as well. Based on the calls of the methods to other relevant methods, HITS

provides a rank to each element in the trace. An extension to this approach is to apply the user query to the remaining methods in the trace.

Shao and Smith [Shao09] propose a combined approach which they use LSI and call graph scores to locate feature relevant elements. The approach starts with LSI, creating a corpus and assigning a score based on the user query. Then, for each of the methods in the LSI list the call graph is created. In the created graph, only the direct neighbour are considered and if one of the direct neighbours is also appeared in the LSI list, then additional score is assigned to the elements. At the end a new ranked list is created which consists of combination score of LSI and call graph.

Shepherd et al. [Shepherd07] used natural language processing and dependency graph to locate feature related elements. In their approach, they used the concept of verb and direct object as the abstract query. As in their previous study in [Shepherd06, the verb represents an action and the direct object is the object on which the action is performed. They proposed a tool called Find-Concept. The tool expands the abstract query using NLP to acquire a more complete set of results. Then using the action-oriented identifier graph model (AOIG), the tool searches for the nodes containing the terms in the query. The tool uses AOIG to detect the dependent elements to those containing the terms and then visualizes the results as a graph.

2.4. Discussion

Feature location techniques aim to facilitate the process of identifying the components that are most relevant to the implementation of a specific feature. This way, software

maintainers do not need to search the entire source to make changes to only a subset of code elements, i.e., the ones relevant to the given feature.

There have been many studies the area of feature location. Techniques vary depending whether they use dynamic analysis, static analysis, textual information, or a combination of these methods.

Most existing techniques have been applied to open source systems. Also, these techniques are not tailored to a specific maintenance task (e.g., debugging, feature enhancement, etc.).

In this thesis, we extend existing work in three ways. First, we propose a feature location approach that focuses on debugging tasks. Second, we apply our feature location to an industrial system in the domain of aircraft simulation. Finally, we show how feature location can be applied to locating configuration errors and not software bugs. To our knowledge, this is the first time that feature location is applied to an artefact rather than the source code (including the traces).

Chapter 3 Feature Location in Simulation Scenarios

3.1. Overall Approach

Figure 3 illustrates the general overview of our approach. First, we generate an execution trace by exercising the scenario of interest. We focus on traces of routine calls since labels are associated with specific routines of the modules. To obtain a correct trace of routine calls we first need to select a feature exercising the simulation scenario.

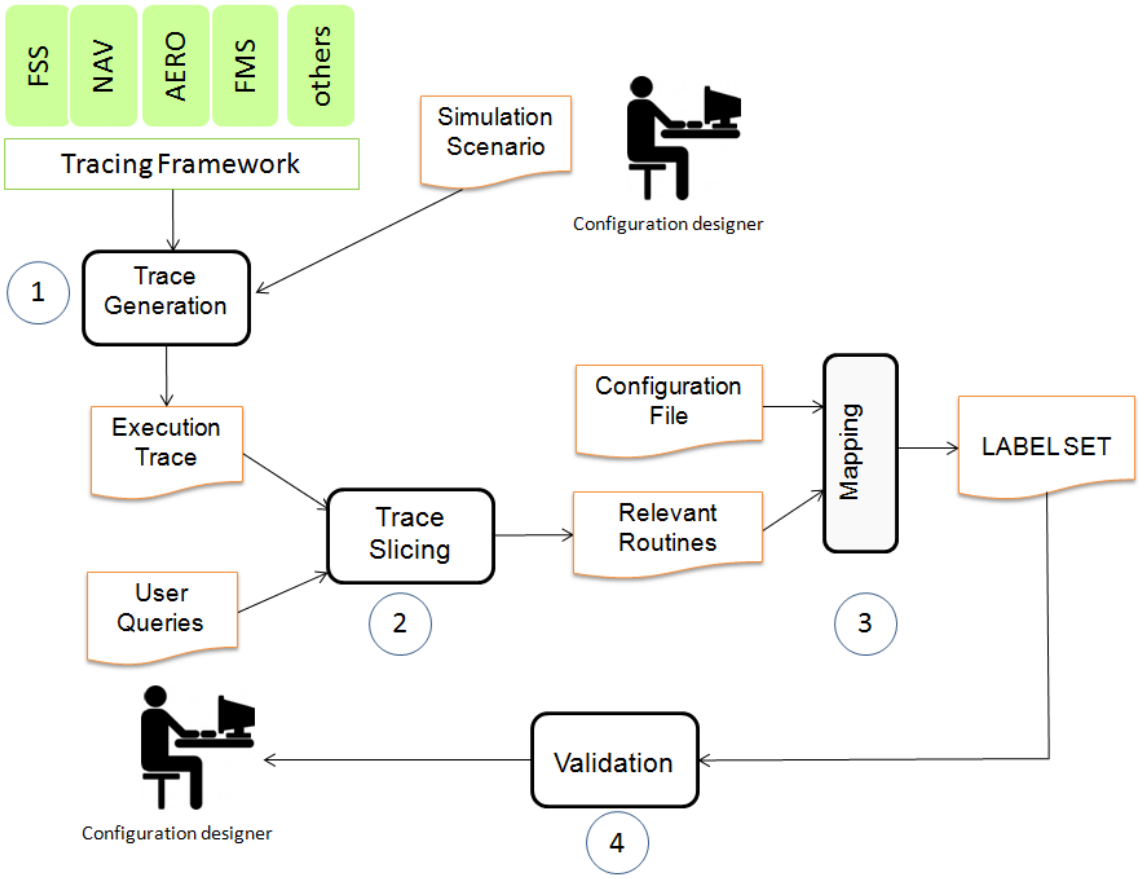

Figure 3. Overall Approach

Detecting the right routines will ultimately lead to the most relevant labels. To this end, we turn to configuration designers who are the intended users of this approach for guidance. We ask them to formulate keywords in the form of a query that can help us detect the routines, most relevant to the observed failure. We rank the routines based on how similar their names are to terms in the query text. Once we identify the most relevant routines, we map their return values (if there are any) to the labels described in the configuration files. These labels are then added to the list of candidate labels. The last step is to present the list to configuration designers for validation. We elaborate on each of this step in more detail in the following subsections.

3.2. Scenario Selection and Trace Generation

To be aligned with the literature on feature location, we can think of a feature, in the context of CAE, as an abstract simulation that defines a particular functionality of an aircraft, whereas a simulation scenario is an instance of a feature with specific input data. The inputs are data coming from other modules which is specified in the configuration files. In other words, the configuration file is an input to the modules, through which the modules get the data about communication with other modules.

To exercise various simulation scenarios, we needed to work very closely with configuration designers at CAE. Many scenarios require special settings; most of them entail extensive knowledge of the aircraft simulation domain. We spent several months at CAE on a full-time basis interacting with configuration designers in order to understand the CAE software landscape and to become familiar with the aircraft simulation domain. This was a necessary task which helped us to analyze the simulation scenarios in detail. A simulation scenario can be exercised in two ways. The first method is to use the lunching

program, which consists of a graphical interface used by the trainers of the simulator device to create a simulated environment for the trainee. This program is mostly used for integration testing. The other way of generating simulation scenarios is to drive the desired inputs from the common database (i.e., labels) through a script file. The script defines the aircraft's specification (flight phase, altitude, angle, speed, etc.) and then lunches the simulation scenario. In this study we used both approaches to create the desired simulation scenarios.

There are various ways to collect trace information. Code instrumentation is perhaps the most popular approach. It consists of inserting probes into the source code and executing the recompiled version. The problem with this approach is that it requires modifying the source code. In the context of CAE, this turned out to be a challenging task to perform. First, we would need to have access to all the modules involved in a simulation. Many of these modules are developed by diverse development teams. In addition, the modules are written in different programming languages, which would necessitate the use of many instrumentation tools. Also, because this study targets configuration designers who do not necessarily have access to the source code, it is important to propose an instrumentation approach that is code-independent. To achieve this, we turn to binary instrumentation. This way, all what we need are executables.

We generate traces of routine calls. By routine, we mean function, procedure, or method. We also keep track of the arguments and return variables of the routines (if there are any). These variables are needed to associate labels in the configuration file to the routines that handle them.

As we mentioned before, the modules are restricted to execute in a specific time frame. Thus, the tracing framework should have low overhead on the system so the execution time would not exceed the time frame. To achieve such performance, we spent several weeks analyzing the most suitable tracing frameworks and profiling tools.

3.3. Extracting Candidate Routines

In this step, we search in the trace for the routines that are most relevant to the failure or the observed behaviour. To achieve this goal, we propose a two-phase process. First, we detect the routines that caused the monitor to issue the wrong warnings. We refer to these routines as *seed routines,* and will use them as a start point of the search process. The next phase is to detect the remaining routines that led to the failure. This process reflects the fact that a configuration error may appear way before the failure. It is therefore important to analyze all the interactions among dependent modules until the detection of the failure.

3.3.1. Detection of Seed Routines

To locate seed routines, we ask configuration designers for directions by asking them to formulate queries that can guide the search process. As we showed in the background Chapter, this is not the first time that queries are used in feature location research (see [Liu97, Marcus04] for examples). Other researchers used source code information (such as comments) combined with user input to obtain informative queries. We deliberately excluded the source code for the reasons we discussed in the trace generation subsection. With the user's query we narrow down the search space only to a sub-set of routines which are likely to be relevant to the failure based on their similarity with the query.

To minimize user intervention, configuration designers at CAE suggested to use the warning messages output by the monitor to formulate queries, as they contain keywords that can help identify the corresponding routines. These warnings are triggered by specific routines in the corresponding modules. For example, in the case of the scenario described in the introduction Section (Figure 1, the dark grey area), TAWS outputs a warning that reads "TAWS Mode1 Warning Sound", when we searched the trace, we found that the name of the corresponding routine, in the TAWS module, carries similar keywords. Thus, the query containing the terms "TAWS Mode1 Warning" is a qualified query which can detect the method triggering the warning output.

The problem is that not all observed failures are described using textual messages. The monitor uses also sound effects, lights, and graphical illustrations, just like in a real airplane. For such cases, we rely on the user's knowledge of the scenarios to formulate adequate queries.

Once a query is formulated, we compare the query keywords with terms extracted from the names of the routines invoked in the trace. By routine name, we also include the name of the class where the routine is defined.

CAE follows strict naming conventions. The camel case style is used for all identifiers, which facilitates term extraction from routines. It should be noted that by term we also include abbreviations. That is to say, we do not attempt to replace them with their original forms. This is because most abbreviations have specific meanings in the context of CAE that describe concepts in the aircraft simulation domain. We assume that configuration designers would use the same abbreviations when formulating queries. We

believe that this is a reasonable assumption given the involvement of configuration designers in the process of drafting queries. At any time, they can change the query to enter abbreviations or long forms, if needed. We suggest as a future direction to build a dictionary that maps abbreviations to their long form to further aid the term extraction process.

To measure the similarity between the terms used in query and in routines, we had several options. There are measurements like; Boolean model [Lancaster73], cosine similarity [Singhal01], Jaccard [Jaccard12] and tf-idf [Hill07] (term frequency/inverse document frequency).

In this thesis, we use tf-idf, a measure that reflects how important a word in a query is to a document in a corpus. For our purpose, we treat each distinct routine of the trace as a document. A corpus is then a set of distinct routines in the trace. The similarity between the query and each routine increases with the number of occurrences of the query terms within a routine. However, terms that are repeated frequently across the whole corpus (i.e., all the routines) are given less priority. For example, if there is a routine r_i that contains many terms of the query and that these terms are not in other routines then r_i should be given a higher rank because it is likely to be specific to the query.

The use of tf-idf is particularly suitable when measuring the similarity between a query and routine names. For example, we may have the situation where a term in the query corresponds to a class name. In such a case, all the routines (invoked in the trace) of that class will be given the same importance when only counting this term. tf-idf offsets that by using the frequency of the term in the corpus (i.e., set of routines). This reflects the

fact that some terms (e.g., class names) are more common than others such as specific terms in routine names.

To present the tf-idf more formally, we define the variables as follows.

- $tf_{t,r}$: Document frequency of term t in the query in routine r.
- idf_t - Inverse document frequency of term t in the corpus. *N* represents the number of distinct routines in the trace. And df_t is the document frequency of term t.

$$idf_t = \log \frac{N}{df_t}$$

- tf-$idf_{t,d}$ is the combined weight for term t in routine r.

$$tfidf_{t,r} = tf_{t,r} * idf_t$$

The similarity between the query *q* and the routine *r* is measured by taking into account the frequency and inverse document frequency of all the query terms with respect to the routine *r*:

$$Sim(q,r) = \sum_{t \in q} tf_{t,r} * idf_t$$

We need to select among the highly ranked routines the ones that are most relevant to the failure. One way to proceed is to define a threshold and take the routines with a rank higher than the threshold. The problem with this technique is that it is almost always challenging to find an adequate threshold that would apply to all scenarios. Besides, even

if we succeed to do this, it might not be the same threshold when applied to other systems. To address this, we simply present the ranked routines to the users and ask them to select the ones they think are most related to the query. A similar approach was used by Liu et al. in [Liu07].

3.3.2. Detection of Remaining Routines

We use seed routines to find the remaining connections among modules that led to the failure. One intuitive way to achieve this is to collect the distinct routines that appear from the root of the trace all the way to the seed routines. In the general case, this would probably be the only way to proceed. However, in the CAE context, each component in the module has an *update* function that is called periodically by the scheduler to update the module's data. A new execution cycle of the component starts by a call to its update function. For example, TAWS is a module in the simulation system, and Mode1 is a component of TAWS module. In order to receive the updated data from common database, scheduler calls the update function of Mode1 in a timely manner.

We use the update routine to slice the trace by keeping only the routines that appear on the call path between the update routine and the seed routines. This way we eliminate routines that are not relevant to the observed behaviour. Because a seed function can appear multiple times in the trace, we need to examine each path from the update function to the seed function occurrence. The resulting routines form a set which is the union of the distinct routines that appear on each path.

3.4. Extracting Labels from Configuration Files

In this step, we search for labels in a configuration file that are connected to return variables of the routines from the previous step. But not all the routines in the call path have a representation in the configuration files. Some of these methods manipulate the local variables or call other functions based on condition statements. The routines which are receiving inputs from the configuration files are specified in the configuration files. To get the mapped labels we simply needed to look for the returned variables name in the configuration files. Although not all the variables which are appeared in the configuration files are connected to labels. Some of the variables are mapped to other local variables in the module. Based on the structure of the configuration file we detect only those which are mapped to labels. This is done automatically by simply parsing the configuration file. The final list of labels is then constructed.

3.5. Validation

We verify the accuracy of the detected labels with the configuration designers. If the labels are not correct then we examine the causes by further exploring the trace. Sometimes, the cause might be due to a poor query. If so, we ask configuration designers to reformulate another (and richer) query. In fact, the query generation is the most important part of this approach. It is because of its important role in detecting the seed method. With a poor query the starting point of the analysis would be from a wrong point in the trace.

Another objective of this step is to learn about ways to improve the approach for future studies.

3.6. Summary

In this chapter, we presented our approach for detecting the parts in the system, including routines and configuration elements, relevant to a specific behavior of the simulation system. The intended result is the set of labels which are the scenario relevant inputs to the module under study.

We proposed the FELODE approach which combines dynamic analysis and textual techniques to perform feature location. The simulation scenarios, exercising the desired functionality of the simulation system are selected with the help of configuration designers. Then the trace of the simulation scenario is collected. From the collected trace, we extract the most relevant routines by comparing a user-formulated query to the name of the methods. We used tf-idf to rank routines that are most similar to words in the query. Then the user is asked to select the routines he or she thinks are related to the scenario. We use this first set of routines (i.e., the ones detected by comparing the user query) to complete the set of relevant routines. The next step is to detect in the configuration files, the labels that correspond to the final set of relevant routines.

Chapter 4 Evaluation

4.1. Target Module

In this thesis, we decided to conduct our experiments using the Flight Surveillance System (FSS) module. FSS is an important module in the simulation system. Buggy simulation scenarios can lead to catastrophic results.

FSS comprises three sub-systems: Terrain Awareness and Warning System (TAWS), Traffic and Collision Awareness System (TCAS), and Weather radar (WXR). TAWS informs the pilot about the terrain condition and generates warnings and alarms when there is a potential crash situation. TCAS is for detecting the traffic in the flight path and alerting the pilot when there is another aircraft in the way. TCAS's behaviour is related to the specification of the intruder aircraft. WXR is for monitoring the weather condition and storm characteristics.

The size of FSS subsystems are of the order of hundreds of thousands lines of code. It is worth mentioning that FSS relies on a framework that handles communications through the shared database. Understanding how FSS works necessitates also the understanding of the framework.

To apply our technique, we selected the scenarios from TAWS and TCAS sub-systems. This was due to the fact that TAWS and TCAS handle conditions which are easy to understand while being important functionality of the full flight simulator. They

communicate with other interesting modules like the navigation module and the exchanging data between these modules is simple and not detailed avionics data.

4.2. Applying FELODE

4.2.1. Scenario Selection

As we explained in the previous sections, we had two options to exercise the simulation scenarios; first using the lunch program and the second option is with script languages. We wanted to create simulation scenarios using both options. As a result, with the help of configuration designers, we created three scripts to define the specification of the aircraft and environment in different conditions. In these scripts, the starting point of the aircraft is defined and the process of its travel till the desired destination is modeled.

We also used the lunch program for the other two scenarios. Using the lunch program, we were able to define the condition of the aircraft and environment through a graphical user interface. But we could not use the lunch program for all the scenario's since we needed to have access to other modules which at the time of our research were not available to us.

As a result, we defined five scenarios, three from TAWS and two from TCAS. Table 4.1 describes the scenarios. While both TAWS and TCAS are FSS sub-systems, they both are accompanied by the dependent modules which are NAV and Terrain. Terrain is a module that has the information about the world map.

Table 4.1. Simulation Scenario Definitions

Scenario #	Sub-System	Scenario
S1	TAWS Mode1	Aircraft is descending at high speed while flying at low altitude.
S2	TAWS Mode4a	The aircraft is close to the ground and is prepared for landing, but the gears are still up.
S3	TAWS Mode4b	Aircraft is in landing mode but the flaps are in a flight position.
S4	TCAS	Simulate the presence of an intruder with the intention to locate its altitude.
S5	TCAS	Simulate the presence of an intruder with the intention to locate its speed.

We explained TAWS Mode1 in the previous sections. For this scenario, we positioned the aircraft in 900 feet altitude with the vertical speed of -3000 feet/min. Thus, we created a situation where Mode1 would be activated. TAWS Mode4a's envelope is shown in Figure 4. When the aircraft is in the grey area, this means that the aircraft is ready for landing, thus if the landing gears are not opened Mode4a must inform the pilot using appropriate alarms. In our simulation case, Mode4a generated the alarm while the user expected to have a safe flight. For this scenario, to activate Mode4a, we put the aircraft in 400 feet above the ground with the airspeed of 50 knots and while the gears positions were up.

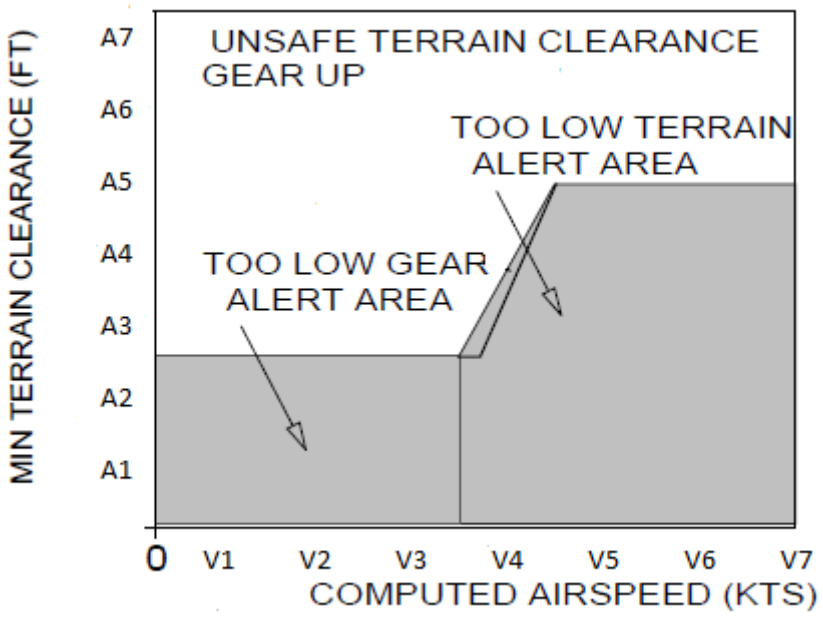

Figure 4. Taws Mode4a Envelope

Similar to Mode4a, Mode4b is for the positioning of the flaps. In general, a simulation scenario can have two phases. The first one is "in landing" phase and the second one is "in flight" phase. Based on the flight phase, the flaps should be either in landing mode or in flight mode. We exercised Mode4b's functionality when the aircraft is approaching the airport and is ready to land, but the flaps were in flight mode. Figure 5 shows the envelope of Mode4b. For this scenario, we put the aircraft in the altitude of 400 feet with airspeed of 50 knots and the flaps in flight position to activate Mode4b.

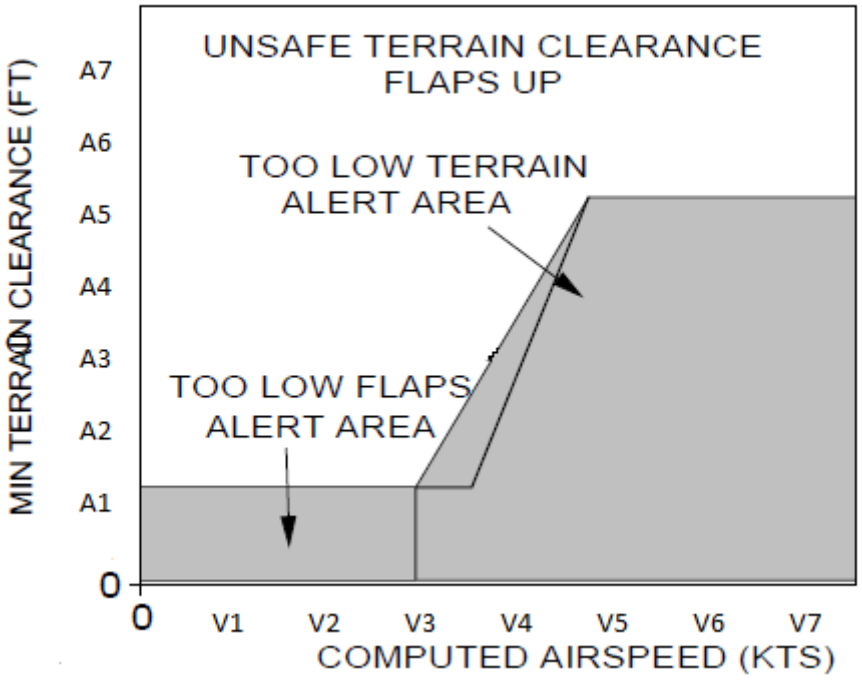

Figure 5. Taws Mode4b Envelope

TCAS functionality is heavily related to the specification of the intruders. Based on different attributes of the intruder like speed, altitude or angle TCAS activates alarms and informs the pilot about any potential danger. In order to understand TCAS's behaviour, we need to consider the intruder's behaviour as well. In the first scenario, we exercised a scenario to examine the intruder's behaviour by measuring its altitude. In this scenario, using the visual information of the radar, we spotted the intruder in self's flight zone and the intention is to check whether it is in the danger zone or not. To create this scenario, using the lunching program, we located the intruder in front of our aircraft in a way that it would pass from beneath the aircraft. For the second TCAS scenario, we were interested in detecting the intruder by measuring its relative speed (speed as a function of the

aircraft's speed). For this scenario, we again located the intruder in front of the aircraft in a way that it would pass from the right side of the aircraft. Altitude and speed are both important measures to assess whether the presence of the intruder is considered dangerous.

4.2.2. Generation of Scenario Traces

We spent several weeks investigating several tracing tools which would not affect the behaviour of the system. Many of the tools caused the scheduler to crash in process. This was due to the fact that the overhead of the selected tools was preventing the modules to communicate correctly.

Finally, we decided to use the PIN framework [PIN], a platform independent tracing tool. PIN provides several useful APIs for different purposes. It allows the users to implement their own customized tracing tool. It has a very low impact on the system if it is implemented correctly using suitable APIs which makes it a good choice for our purpose. PIN supports both binary and code instrumentation. We favoured binary instrumentation in this case to avoid modifying the code. Table 4.2 shows the size of the generated traces. We saved each scenario in a configuration file. The number of labels for each scenario is also shown in Table 4.2. For example, for Scenario S1, there are 720 labels. We were told by configuration designers that complex scenarios may result in more labels, but running such scenarios would require advanced settings and access to lab facilities within CAE for which extensive training is needed.

Table 4.2. Trace Statistics

Scenario	File Size	Number of Routine Calls	Number of Labels in Configuration File
S1	310 MB	7,734,123	720
S2	359 MB	8,126,237	720
S3	250 MB	4,533,630	720
S4	267 MB	4,844,231	620
S5	269 MB	4,879,325	620

4.2.3. Formulating the Query

We asked one experienced configuration designers to help formulate queries for each scenario. He used the behaviour depicted in the monitor to guide the drafting of the query. In what follows, we show the behaviour of each scenario.

S1: After few seconds of execution, the monitor shows a flashing red light next to a message which says "*TAWS Mode1 Warning Sound*". The experienced configuration designer proposed to use the term in the message as the basis for the query.

S2: The monitor reads the flight phase as "*In Landing*" and as time passes the altitude is reducing. After few seconds of execution, a blinking message appears next to the message "*TAWS Mode4a*" and the blinking message reads "*Gears*".

S3: The monitor reads the flight phase as "*In Landing*". Similar to S2, after few seconds of execution, a blinking message appears next to message "*TAWS Mode4b*" and the blinking message reads "*Flaps*".

S4: For TCAS, the monitor does not give textual information. Instead it shows radar information like the ones available in real aircrafts. We observed a moving dot which got closer as time passed. At the end of the scenario it turned into a red dot and passed by the centre of the radar. During the scenario execution, a vocal message "*Pull Up!*" was triggered.

S5: Similar to S4, The monitor showed the radar and the approaching dot, but in this scenario the dot was traveling in a high speed triggered the vocal message "*Bear Left!*"

The formulated queries for our simulation scenarios are as follows:

- S1: TAWS Mode1 Warning
- S2: TAWS Mode4a Gears
- S3: TAWS Mode4b Flaps
- S4: TCAS Intruder Altitude
- S5: TCAS Intruder Speed

4.2.4. Ranking the Methods

For each generated trace, we applied the ranking method to rank the methods using the user queries.

For S1, Table 4.3 shows that "`taws::TawsMode1::warningMessage`" receives higher score since it has all three terms and the term "taws" is repeated twice. Routine "`taws::AudioHandler::warningSets`" receives a higher rank than "`taws::TawsMode1::checkSound`" since the term "warning" is more specific than "Mode1".

Table 4.3. Ranked routines for S1

Routine	Score
taws::TawsMode1::warningMessage	3.10274
taws::Malfunctions::mode1FalseWarning	2.20650
taws::AudioHandler::warningSets	1.86448
taws::Malfunctions::flightFalseWarning	1.86448
taws::TawsMode1::checkSound	1.76520
taws::TawsMode1::modeStatus	1.76520
taws::TawsMode1::auralStatus	1.76520
taws::TawsMode1::checkMode	1.76520

In Table 4.4, the routine "`taws::TawsAircraft::gearsStatus`" receives the highest rank, although it has only two of the terms in the query, but the term "gears" is more specific.

Table 4.4. Ranked routines for S2

Routine	Score
taws::TawsAircraft::gearsStatus	1.56395
taws::TawsMode4a::mode4aCaution	1.38648
taws::TawsMode4a::modeStatus	1.25924
taws::TawsMode4a::auralStatus	1.25924
taws::TawsMode4a::checStatus	1.25924
taws::TawsMode4a::isFailed	1.25924
taws::TawsMode4a::malfunctions	1.25924
taws::TawsMode4a::checkEnv	1.21287

Table 4.5. Ranked routines for S3

Routine	Score
taws::TawsAircraft::flapsStatus	1.562546
taws::TawsMode4b::mode4bCaution	1.40036
taws::TawsMode4b::modeStatus	1.38234
taws::TawsMode4b::auralStatus	1.38234
taws::TawsMode4b::modeStatus	1.38234
taws::TawsMode4b::isFailed	1.38234

taws::TawsMode4b::malfunctions	1.38234
taws::TawsPriorityManager::set	0.32188

Same as for S2, the routine "`taws::TawsAncillaries::flapsStatus`" receives the highest rank (shown in Table 4.5) since the term "flaps" is more specific.

Table 4.6. Ranked routines for S4

Routine	Score
tcas::Intruder::intruderRelativeAltitude	2.03595
tcas::Intruder::intruderAltitude	2.03595
tcas::Tcas2Track::altitudeCheck	1.36398
tcas::TcasSim::altitudeSlew	1.36398
tcas::TcasFlight::pressureAltitude	1.36398
tcas::Tcas::agl	0.99386
tcas::TcasIntruders::intruderGround	0.99386
tcas::TcasIntruders::intruderItem	0.99386

In scenario S4, as Table 4.6 shows, the weight of the term "altitude" is higher than "intruder" thus a higher score is assigned to the corresponding routines. For this scenario the two highest ranked results are "`tcas::Intruder::intruderRelativeAltitude`" and "`tcas::Intruder::intruderAltitude`" since they contain all three terms.

Table 4.7. Ranked routines for S5

Routine	Score
tcas::Intruders::groundSpeed	2.19870
tcas::Intruders::airSpeed	2.19870
tcas::Intruders::horizontalSpeed	2.19870
tcas::Intruders::verticalSpeed	2.19870
tcas::Navigation::verticalSpeed	1.50264
tcas::Navigation::airSpeed	1.50264
tcas::Intruders::intruderIntention	1.01794
tcas::Intruders::intruderItem	1.01794

Four routines in the top of Table 4.7 receive the same score since they all contain the three terms in the query.

4.2.4. Selecting the Seed Methods

For each scenario, we presented the ranked list of routines to the same configuration designer who helped us in formulating the query and asked for their feedback. Configuration designers selected the following seed routines.

- S1: One method was selected. This method triggers the warning message.
  - `taws::TawsMode1::warningMessage`
- S2: Two methods were selected. The first method is responsible for reading the gears status (up or down) and the second method triggers the warning message.
  - `taws::TawsAircraft::gearsStatus`

- `taws::TawsMode4a::mode4aCaution`

- S3: Similar to S2, two methods were selected. First one for reading the status of flaps (in flight position or in landing position) and the second one triggers the warning message.
  - `taws::TawsAircraft::flapsStatus`
  - `taws::TawsMode4b::mode4bCaution`
- S4: Two methods were selected. Both methods are for measuring intruder's altitude.
  - `tcas::Intruders::intruderRelativeAltitude`
  - `tcas::Intruders::intruderAltitude`
- S5: Four methods were selected. All four methods are for measuring intruder's speed in different criteria.
  - `tcas::Intruders::groundSpeed`
  - `tcas::Intruders::airSpeed`
  - `tcas::Intruders::horizontalSpeed`
  - `tcas::Intruders::verticalSpeed`

4.2.5. Trace Slicing and Extracting Call Paths

For each scenario we slice the corresponding execution trace from the occurrence of the seed routine, moving backward till the first encountered update method.

Both update method and seed routines appear multiple times in the trace, therefore we extracted the distinct methods from each path between update method and seed routines. We also were careful that some seed routines may share the call path. In these cases we extracted only one call path for the seed routines. The following tables show the call path for each scenario and their seed routines.

Routines presented in Table 4.8 are the methods executed before the seed routines. These methods produce results based on new values received from common database and based on those values cause the seed routine to get executed. Thus, the methods responsible for checking the values for altitude and descending rate (which are the key attributes for Mode1 feature) are likely to be in the call path.

Table 4.8. S1: Call path

Seed Routine: "`taws::TawsMode1::warningMessage`"
taws::TawsMode1::doUpdate
taws::TawsMode1::isFaild
taws::TawsMode1::checkStatus
taws::Navigation::glides
taws::Navigation::vertcalSpeed
taws::Input::aboveGroundLevel
taws::Envelope::isInEnv
taws::TawsMode1::auralStatus

taws::TawsMode1::warningMessage

Table 4.9. S2: Call Path 1

Seed Routine: "`taws::TawsAircraft::gearsStatus`"
taws::Input::update
taws::Input::computeAltitude
taws::Input::computeAboveSeaLevel
taws::Navigation::aboveSeeLevelValid
taws::Navigation::aboveSeaLevel
taws::Input::computeAboveGroundLevel
taws::Navigation::aboveGroundLevel
taws::TawsAircraft::gearsStatus

Table 4.10. S2: Call Path 2

Seed Routine: "`taws::TawsMode4a::mode4aCaution`"
taws::TawsMode4a::update
taws::TawsMode4a::checkStatus
taws::Navigation::runWayIndex
taws::Navigation::airspeed
taws::Input::aboveGroundLevel

taws::Input::flightPhase
taws::Envelope::isInEnv
taws::TawsMode4a::auralStatus
taws::TawsMode4a::mode4aCaution

For scenario S2, two call paths are extracted (Tables 4.9 and 4.10). In this scenario, we are looking for routines responsible for checking or receiving the values for altitude, air speed and the status of the gears. Mode4a functionality is closely related to the values for these attributes.

Table 4.11. S3: Call Path 1

Seed Routine: "`taws::TawsAircraft::flapsStatus`"
taws::Input::update
taws::Input::computeAltitude
taws::Input::computeAboveSeaLevel
taws::Navigation::aboveSeeLevelValid
taws::Navigation::aboveSeaLevel
taws::Input::computeAboveGroundLevel
taws::Navigation::aboveGroundLevel
taws::TawsAircraft::Shields
taws::TawsAircraft::flapsStatus

Table 4.12. S3: Call Path 2

Seed Routine: "`taws::TawsMode4a::mode4aCaution`"
taws::TawsMode4a::update
taws::TawsMode4a::checkStatus
taws::Navigation::glides
taws::Navigation::airspeed
taws::Input::aboveGroundLevel
taws::TawsMode4a::checkEnv
taws::TawsMode4a::flightPhase
taws::TawsMode4a::isInEnv
taws::TawsMode4a::auralStatus
taws::TawsMode4a::mode4aCaution

For scenario S3, similar to scenario S2, two call paths are detected (Tables 4.11 and 4.12). We are looking for the methods related to attributes: air speed, altitude and flaps positioning. Having the value of these attributes, we can pinpoint the causes of the aircraft's behaviour.

Table 4.13. S4: Call Path

Seed Routines: "tcas::Intruders::intruderAltitude" "tcas::Intruders::intruderRelativeAltitude"
tcas::ApproachingIntruders::update
tcas::Intruders::activeIntruder
tcas::Intruders::heading
tcas::ApproachingIntruders::topThreat
tcas::ApproachingIntruders::topThreatID
tcas::Intruders::isOnGround
tcas::Intruders::groundSpeed
tcas::Intruders::verticalSpeed
tcas::Intruders::horizontalSpeed
tcas::Intruders::intruderAltitude
tcas::Intruders::airSpeed
tcas::Intruders::intruderRelativeAltitude

For scenario S4, seed routines share the same call path. As a result, we created a single call path for both shown in Table 4.13. In this call path, the data related to the approaching thread is processed. The input to TCAS is from another module which has the information about the traffic in the flight path. For this scenario, the important attribute is the relative latitude of the intruder from self-aircraft. The relative altitude is the function of self-altitude and the descending or ascending rate of the intruder.

Table 4.14. S5: Call Path

Seed routines: "`tcas::Intruders::groundSpeed`" "`tcas::Intruders::airSpeed`" "`tcas::Intruders::horizontalSpeed`" "`tcas::Intruders::verticalSpeed`"
tcas:: *ApproachingIntruders*::*update*
tcas::Intruders::selectedIntruder
tcas::Intruders::activeIntruder
tcas:: ApproachingIntruders::outputs
tcas::ApproachingIntruders::heading
tcas::ApproachingIntruders::topThreat
tcas::Intruders::isOnGround
tcas::Intruders::groundSpeed
tcas::Intruders::verticalSpeed
tcas::Intruders::horizontalSpeed
tcas::Intruders::intruderAltitude
tcas::Intruders::airSpeed

In this scenario, like S4 seed routines share same call path (shown in Table 4.14). While speed, like altitude is an attribute related to the approaching intruder the call path starts with update method of "`ApproachingIntruders`" class. Speed of an aircraft is measured from different points. For example, ground speed is the measured speed from a fixed point on the ground.

4.2.6. Mapping to Configuration Files

In Tables 4.8 to 4.14, routines shown in bold have an instance in configuration files. This means that there is a mapping between the code and configuration files. Through this mapping we detect the configuration parts which are related to the observed behaviour. There is a part in the configuration file which has information about "`taws::Navigation::vertcalSpeed`". This routine was detected in the call path of "`taws::TawsMode1::warningMessage`".This part of the configuration file specifies that the element "`verticalSpeed`" of component "`Navigation`" from the module "`TAWS`" reads its value which is stored in the label "`NAV1VSpeed`". The reading method is also specified in the configuration file as "`LabelType`" which confirms that the original value is stored in a label. We have already obtained the essential information such as element name (`VerticalSpeed`) and class name in the body of detected method. To obtain the name of the label we used simple XML parsing and produced the result as a pair of element id and label name. For this case, the result would be <`Navigation::verticalSpeed, NAV1VSpeed`>.

For routine "`taws::Navigation::vertcalSpeed`" we succeeded to find the mapped label. But the other detected routine in S1, although we found the representation of it in the configuration part, we could not find the mapped label. The configuration file part for "`taws::Input::aboveGroundLevel`" is different in reading method. For this elements, the reading method is defined as "`InternalType`" which specifies that the original value for "`aboveGroundLevel`" element is calculated locally within the system. As the result it is not mapped to any label.

We did the mapping for all detected routines in the previous section and produced label sets for each of the scenarios. Tables 4.15 till 4.19 show the results that are mapped to labels in the common database.

Table 4.15. S1 label set

Navigation::verticalSpeed	**NAV1VSpeed**
Navigation::glides	NAV1Glide

Table 4.16. S2 label set

Navigation::aboveGroundLevel	**NAV1AGLevel**
Navigation::runWayIndex	NAV1RWIndex
Navigation::aboveSeeLevelValid	NAV1ASLevelV
Navigation::aboveSeaLevel	NAV1ASLevel
TawsAircraft::gearsStatus	**AC2POGear**
Navigation::airSpeed	**NAV1ASpeed**

Table 4.17. S3 label set

Navigation::aboveGroundLevel	**NAV1AGLevel**
Navigation::glides	NAV1Glide
Navigation::aboveSeeLevelValid	NAV1ASLevelV
Navigation::aboveSeaLevel	NAV1ASLevel
TawsAircraft::flapsStatus	**AC2POFlap**

Navigation::airSpeed	**NAV1ASpeed**

Table 4.18. S4 label set

tcas::Intruders::verticalSpeed	**INT1VSpeed**
tcas::Intruders::heading	INT1Heading
tcas::Intruders::activeIntruder	INT1INTActive
tcas::Intruders::groundSpeed	INT1GSpeed
tcas::Intruders::horizontalSpeed	INT1HOSpeed
tcas::Intruders::intruderAltitude	**INT1INTAltitude**
tcas::Intruders::intruderRelativeAltitude	**INT1INTMAltitude**
tcas::Intruders::airSpeed	INT1ASpeed

Table 4.19. S5 label set

tcas::Intruders::groundSpeed	**INT1GSpeed**
tcas::Intruders::heading	INT1Heading
tcas::Intruders::activeIntruder	INT1INTActive
tcas::Intruders::airSpeed	**INT1ASpeed**
tcas::Intruders::horizontalSpeed	**INT1HOSpeed**
tcas::Intruders::verticalSpeed	**INT1VSpeed**
tcas::Intruders::intruderAltitude	INT1INTAltitude

4.2.7. Evaluating Results

To evaluate the result of our approach, we needed to have the valid labels for each scenario, something to compare our results against. We asked the same expert to provide us with the most relevant labels. In Tables 4.15 to 4.19, bolded routines and labels are the validated results by configuration experts.

We used precision and recall to measure the accuracy of our approach. We define precision and recall as follows:

$$\text{Precision} = \frac{Number\ of\ valid\ labels\ detected}{Total\ number\ of\ all\ detetced\ labels}$$

$$\text{Recall} = \frac{Number\ of\ valid\ labels\ detected}{Total\ nimber\ of\ valid\ labels\ for\ the\ scenario}$$

Table 4.20 shows the results. We can observe that the approach has good recall but relatively low precision. For all scenarios (except Scenario S1), the recall is 100%. This means that we detected all valid labels. The precision, on the other hand, indicates that we detected also labels (though not too many) that were irrelevant to the failure.

Table 4.20. Precision and Recall

N1: Number of labels detected by the approach; N2: Number of valid labels detected by the approach; N3: Number of valid labels for each scenario, provided by the expert.

Scenarios	N1	N2	N3	Precision (N2/N1)	Recall (N2/N3)
S1	**2**	**1**	**2**	**50%**	**50%**
S2	**6**	**3**	**3**	**50%**	**100%**
S3	**6**	**3**	**3**	**50%**	**100%**
S4	**8**	**3**	**3**	**38%**	**100%**
S5	**7**	**4**	**4**	**57%**	**100%**

For Scenario S1, we detected two labels but only one of them is valid. The valid label holds the descending speed of the plane. In this scenario, the plane was going at -3000 feet a minute. The approach missed a label that is used to store the plane's altitude. After analysis of the trace content, we found that the corresponding function did not appear in the trace path. This was caused by the fact that the query only referred to the TAWS warning without specifying the factors that might have caused these warnings (i.e., altitude and speed). A richer query would have given better recall with the risk of further reducing precision.

For Scenario S2, the query resulted in two seed functions selected by the configuration designer. As a result, we had to include routines from two different execution paths. We detected six relevant routines. Only three of them return variables that map to the correct labels. These functions return altitude, airspeed, and flaps position. For Scenario S3, the result was similar. We detected three valid labels that represent the altitude of the aircraft, the positioning of the gears, and the caution message to the pilot about the status of the gears.

In both cases, we detected labels that were not on the list of valid labels provided by the expert. The first label represents the altitude above sea (Mode4 is concerned with the altitude above ground only). This label would have been eliminated if the query had the keyword "ground" in it. The next two labels are used for consistency checks (for example, making sure that the altitude is returned only when it is available). They might not be relevant to the failure but are needed internally to ensure that the modules are functioning properly.

For TCAS Scenario S4, we detected the altitude above sea, the relative altitude of the intruder, and the intruder's vertical speed. And for the second scenario (S5), we detected all valid labels which represent speed properties were vertical, horizontal and relative speed as well as the intruder's airspeed. But again, for both TCAS scenarios, the precision was relatively low. The additional labels that were detected return information about the intruders in the area (e.g., number of intruders, intruders heading, etc.).

4.3. Discussion

We showed the results to two configuration designers at CAE. In their opinion, there are two main factors that contributed to the significance of the study. The first one is the fact that the approach detects (in most cases) all valid labels (i.e., it has good recall). For example, using this approach, for Scenario S4 (which has the lowest precision 38%), configuration designers will need to examine, in the worst case scenario, only eight labels instead of going through the entire configuration file which contains 620 labels (see Table 4.2). The relatively low precision did not seem to be a concern because the number of detected labels was considerably smaller than the number of labels in the configuration files (in our cases, we detected at most eight labels).

The second factor has to do with the fact that our FELODE does not require static analysis of the source code or access to any other system artefacts except trace information. This is an important enabler for the adoption of this method because it fits well with the actual work environment of configuration designers. It is particularly well suited in an environment with heterogeneous software systems relying solely on software binaries. The approach is also simple to use.

Precision can be improved in two ways. First, by having configuration designers continuously refine the queries and re-execute the approach until a satisfactory set of labels is identified. The challenge with this method is to know when to stop. Another approach is to build a model that associates the behaviour exhibited by the monitor with labels in the shared database. The model can be improved overtime as new failures occur. This learning-based approach can be further combined with a query-based model for full detection power.

Finally, during this study, our ultimate objective was to detect key labels that are most relevant to the observed failure. However, after examining the results of the case study, we realized that there are also other labels that might not be the most important ones but can still contribute (perhaps at a lesser degree) to understanding the cause of the failure. For example, knowing the intruder's information for Scenario S4 and S5 might be useful to debug similar scenarios. Adding the corresponding labels to the detected labels would increase significantly precision.

4.3.1. Lessons Learned

We demonstrated through this study that feature location techniques can help in debugging tasks in an industrial setting. However, each environment will likely necessitate a tailor-made approach. We could not directly apply existing techniques because they required either multiple traces for each scenario [Antoniol05, Antoniol06, Eisenberg05, Eisenbarth01b, Wilde92, Wilde95], or access to the source code [Chen00, Hayashi10a, Hill07, liu07, Rajlich04, Rohatgi08]. Both solutions were quickly rejected and found impractical in the context of CAE. Generating multiple traces means exercising many simulation scenarios. We discussed the limitations of using source code

analysis in the previous sections. It was important to design a light-weight solution that is simple to use and implement. But most importantly, a solution that does not require significant changes to the work habits of the configuration designers.

In the beginning of the study, we investigated fully automated solutions. However, after conducting the experiments, we realized that the user input was critical to reducing the complexity of finding the most relevant routines in the trace. We believe that any future work should integrate user feedback as a key element. Furthermore, the approach should be tailored to varying levels of experience and domain knowledge of the users. To reduce user intervention, we can invest in building models that capture essential knowledge needed for the approach. For example, there should be a way to save queries and enrich them overtime for further use. We believe that the effort spent on managing this knowledge will pay off in the future by increasing the detection accuracy of the approach.

Finally, we found that input from CAE software engineers was critical to the design choices we made. For example, the two-phase approach for extracting routines from a trace was suggested by a CAE configuration designer. Also, guidance from CAE engineers greatly facilitated our efforts to relate terms in the query to terms in routine names.

Chapter 5 Conclusion

5.1. Research Contributions

In this study, we introduced a novel approach for locating features in configuration files used at CAE to understand the behaviour of simulation scenarios.

Our approach uses dynamic information stored in trace files and user queries to obtain the relevant elements to the scenario under study. Dynamic analysis narrows down the search space that would have been unnecessarily complex if static analysis is used. Combined with user queries, our approach is capable of detecting scenario-relevant routines and configuration labels.

To our knowledge, this is the first time that a feature location technique is applied to the avionic domain. We applied FEOLDE on two sub-modules of CAE and five different scenarios. We achieved in average 50% precision and up to 100% recall. We argued that the precision can be further improved by (a) having richer queries, and (b) considering labels that are not most relevant but still contribute to the understanding of the failure. One key finding of this study is that feature location techniques, once customized depending on the context, are applicable to solving real industrial problems.

Future research should focus on conducting additional experiments with more simulation scenarios. In this experiment, we isolated some of the modules to analyze the behaviour. We also need to study the performance of FELODE, especially when applied to complex scenarios which require processing extremely large traces.

We need to gain more comprehensive knowledge of (a) the variables defining a simulation scenario failure, and (b) relationship among modules. This would help configuration designers to draft richer queries which will ultimately lead to better trace slicing techniques. We also need to build a knowledge base where queries are saved and improved over time. This knowledge-directed approach can further enhance the detection accuracy.

Chapter 6 References

[Antoniol05] Antoniol, G. and Guéhéneuc, Y., (2005), "Feature Identification: A Novel Approach and a Case Study," *In Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM'05),* Budapest, Hungary, pp. 357-366, 2005.

[Antoniol06] Antoniol, G. and Guéhéneuc, Y. G., (2006), "Feature Identification: An Epidemiological Metaphor," *IEEE Transactions on Software Engineering, 32(9),* pp. 627-641, 2006.

[Bohnet08b] Bohnet, J., Voigt, S., and Dollner, J., (2008b), "Locating and Understanding Features of Complex Software Systems by Synchronizing Time-, Collaboration- and Code-Focused Views on Execution Traces", in Proceedings of 16th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, The Netherlands, June 10-13, pp. 268-271, 2008.

[Brin98] Brin, S. and Page, L., (1998), "The Anatomy of a Large-Scale Hypertextual Web Search Engine", in Proceedings of 7th International Conference on World Wide Web, Brisbane, Australia, pp. 107-117, 1998.

[Chen00] Chen, K. and Rajlich, V., (2000), "Case Study of Feature Location Using Dependence Graph," *In Proceedings of 8th IEEE*

International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, pp. 241-249, 2000.

[Dit13] Dit, B., Revelle, M., Gethers, M. and Poshyvanyk, D. (2013), "Feature location in source code: a taxonomy and survey," *Wiley Journal on Software Evolution and Practice, 25(1),* pp 53-95, 2013.

[Eisenberg05] Eisenberg, A. D. and De Volder, K., (2005), "Dynamic Feature Traces: Finding Features in Unfamiliar Code," *In Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM'05),* Budapest, Hungary, pp. 337-346, 2005.

[Eisenbarth01b] Eisenbarth, T., Koschke, R., and Simon, D., (2001b), "Derivation of Feature Component Maps by means of Concept Analysis," *In Proc. of European Conference on Software Maintenance and Reengineering (CSMR'01),* pp. 176-179, 2001.

[Hayashi10a] Hayashi, S., Sekine, K., and Saeki, M., (2010a), "iFL: An interactive environment for understanding feature implementations," *In Proc. of the 26th IEEE International Conference on Software Maintenance (ICSM'10),* Timisoara, Romania, pp. 1-5, 2010.

[Hill07] Hill, E., Pollock, L., and Vijay-Shanker, K., (2007), "Exploring the Neighborhood with Dora to Expedite Software Maintenance," *In*

Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), pp. 14-23, 2007.

[Jaccard12] Jaccard P. (1912): The distribution of the flora in the alpine zone. New Phytologist 11(2), pp 37-50, 1912.

[Kleinberg99] Kleinberg, J. M., (1999), "Authoritative Sources in a Hyperlinked Environment", *Journal of the ACM*, vol. 46, no. 5, pp. 604-632.

[Lancaster73] Lancaster, F. Wilfrid. and Fayen, Emily Gallup. *Information retrieval: on-line [by] F. W. Lancaster and E. G. Fayen* Melville Pub. Co Los Angeles 1973.

[Liu07] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., (2007), "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," *In Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07),* Atlanta, Georgia, USA, pp. 234-243, 2007.

[Marcus04] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., (2004), "An Information Retrieval Approach to Concept Location in Source Code," *In Proc. of 11th IEEE Working Conference on Reverse Engineering (WCRE'04),* Delft, The Netherlands, pp. 214-223, 2004.

[Petrenko08] Petrenko, M., Rajlich, V., and Vanciu, R., (2008), "Partial Domain Comprehension in Software Evolution and Maintenance", in

International Conference on Program Comprehension, Washington, USA, pp. 13-22, 2008.

[PIN] PIN - "Pin - A Dynamic Binary Instrumentation Tool." Intel Corporation, URL: http://software.intel.com/en-us/articles/pin-a dynamic-binary-instrumentation-tool, Retrieved on July 12, 2012.

[Rajlich04] Rajlich, V. and Gosavi, P., (2004), "Incremental Change in Object-Oriented Programming," *In IEEE Software,* pp. 2-9, 2004.

[Revelle10] Revelle, M., Dit, B., and Poshyvanyk, D., (2010), "Using Data Fusion and Web Mining to Support Feature Location in Software", in Proceedings of 18th IEEE International Conference on Program Comprehension (ICPC'10), Braga, Portugal, June 30 - July 2, pp. 14-23, 2010.

[Robillard02] Robillard, M. P. and Murphy, G. C., (2002), "Concern Graphs: Finding and describing concerns using structural program dependencies", in Proceedings of International conference on software engineering, pp. 406–416, 2002.

[Robillard05a] Robillard, M., (2005a), "Automatic Generation of Suggestions for Program Investigation," *In Proc. of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering,* Lisbon, Portugal, pp. 11 – 20, 2005.

[Rohatgi08] Rohatgi, A., Hamou-Lhadj, A., and Rilling, J., (2008), "An Approach for Mapping Features to Code Based on Static and Dynamic Analysis," *In Proc. of the 16th IEEE International Conference on Program Comprehension (ICPC'08),* Amsterdam, The Netherlands, pp. 236-241, 2008.

[Rohatgi09] Rohatgi, A., Hamou-Lhadj, A., and Rilling, J., (2009), "An Approach for Solving the Feature Location Problem by Measuring the Component Modification Impact," *IET Software, 3(4),* pp. 292-311, 2009.

[Saul07] Saul, M. Z., Filkov, V., Devanbu, P., and Bird, C., (2007), "Recommending Random Walks", in Proceedings of 11th European Software Engineering Conference held jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'07), Dubrovnik, Croatia, pp. 15-24, 2007.

[Shao09] Shao, P. and Smith, R. K., (2009), "Feature location by IR modules and call graph", in Proceedings of ACM Annual Southeast Regional Conference, Clemson, South Carolina. Article No. 70, 2009.

[Shepherd06] Shepherd, D., Pollock, L., and Vijay-Shanker, K., (2006), "Towards Supporting On-Demand Virtual Remodularization Using Program Graphs", in Proceedings of International Conference on

Aspect-Oriented Software Development (AOSD'06), Bonn, Germany, pp. 3-14, 2006.

[Shepherd07] Shepherd, D., Fry, Z., Gibson, E., Pollock, L., and Vijay-Shanker, K., (2007), "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns", in Proceedings of 6th International Conference on Aspect Oriented Software Development (AOSD'07), pp. 212-224, 2007.

[Singhal01] Singhal, Amit (2001). "Modern Information Retrieval: A Brief Overview". Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 24 (4), pp 35–43, 2001.

[Wilde92] Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D., (1992), "Locating User Functionality in Old Code," *In Proc. of IEEE International Conference on Software Maintenance (ICSM'92),* Orlando, FL, USA, pp. 200-205, 1992.

[Wilde95] Wilde, N. and Scully, M., (1995), "Software Reconnaissance: Mapping Program Features to Code," *In Proc. of the Wiley Journal of Software Maintenance: Research and Practice, 7(1),* pp. 49-62, 1995.

[Wong99] Wong, W. E., Gokhale, S. S., Horgan, J. R., and Trivedi, K. S., (1999), "Locating program features using execution slices", in Proceedings of IEEE Symposium on Application-Specific Systems

and Software Engineering and Technology (ASSET'99), March 24-27, pp. 194-203, 1999.