Software Feature Location in Practice: Debugging Aircraft Simulation Systems

Salman Hoseini, Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
{sa_hosei, abdelw}@ece.concordia.ca

Patrick Desrosier, Martin Tapp
CAE Inc.
Montreal, QC, Canada
{patrick.desrosiers, martin.tapp}@cae.com

Abstract

In this paper, 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%.

Keywords: Feature Location, Trace Analysis, Debugging of Simulation Systems, Avionic Systems.

1. Introduction

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 is performed, 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 paper)─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.

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.

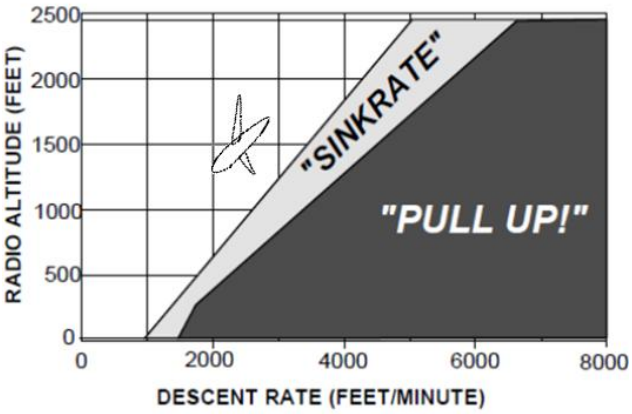

Figure 1. Example of a simulation scenario

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.

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 paper, 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.

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. The paper contributes to the current literature in the following ways. First, 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 (see [3] for a survey on feature location).

The second contribution of the paper is 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 [3]. 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. Finally, by locating features in configurations files, we demonstrate the applicability of feature location principles to other software engineering artefacts rather than the source code.

The rest of the paper is organized as follows: In Section 2, we discuss simulation scenarios in more detail, providing the reader with the necessary background to understand the content of this paper. In Section 3, we describe our approach for locating simulation scenarios in configuration files. The evaluation of the approach is the subject of Section 4. We report on lessons learned in the same section. We discuss threats to validity in Section 5, followed by related work. We conclude the paper in Section 7.

2. 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.

Modules communicate by exchanging *labels* (one can think of labels as messages exchanged among processes in a distributed architecture). CAE keeps a database of predefined labels used for different purposes. Each module receives labels through variables and transfers them to the routines that execute the required code.

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

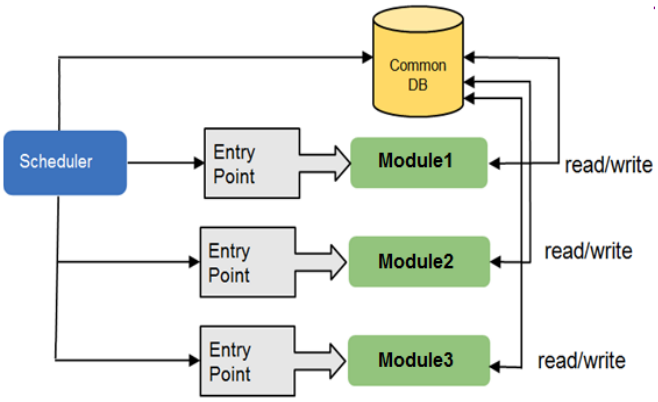

Figure 2. Generalized System Architecture

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

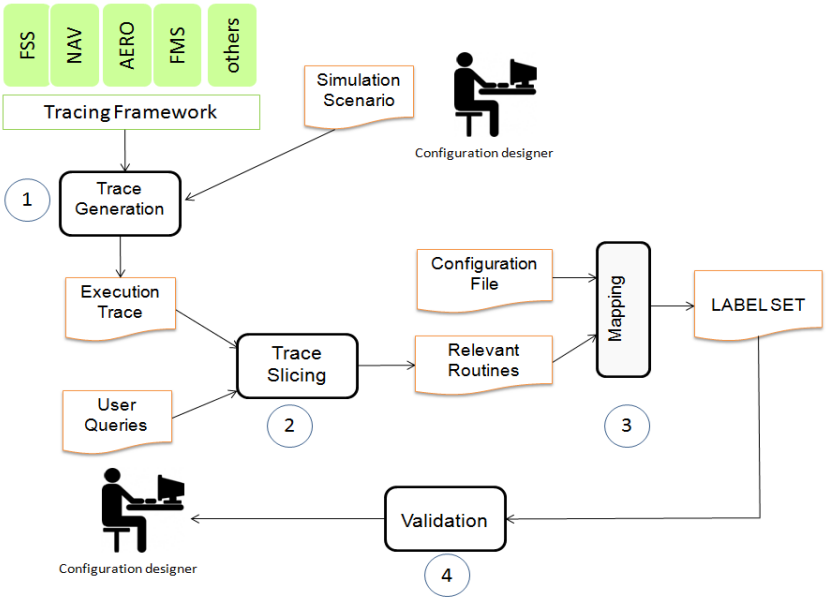

Figure 3. Overall Approach

debugging complex simulations. Configuration designers can then focus on simulating new and interesting scenarios instead of fixing existing ones.

3. FELODE Approach for Locating Simulation Scenarios in Configuration Files

Figure 3 shows the steps 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. Therefore, detecting the right routines will ultimately lead to the most relevant labels. To this end, we turn to configuration designers (users of this approach) for guidance. We ask them to formulate keywords (in the form of queries) 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.

A. 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 (modules and connections).

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. The first author of the paper 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.

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.

B. Extracting Candidate Routines

In this step, we search in the trace for the routines that are most relevant to the failure. 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 modules until the detection of the failure.

B.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. This is not the first time that queries are used in feature location research (see [9, 10] 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.

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 previous section, 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.

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 similarity, we propose to use tf-idf (term frequency/inverse document frequency) [8]. tf–idf is 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.

More formally:

- $tf_{t,r}$: 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.

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

- $tf\text{-}idf_{t,d}$ is a 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 [9].

B.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 module has an *update* function that is called periodically by the scheduler to update the module's data. A new execution cycle of the module starts by a call to its update function.

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.

C. 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. This is done automatically by simply parsing the configuration file. The final list of labels is then constructed.

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

4. Case Study

We show the effectiveness of our approach, FELODE, by applying it to various simulation scenarios at CAE. The case study aims to answer the following question: Can we use trace information combined with user queries to detect labels (module connections) that are most relevant to an observed simulation failure at CAE? The answer to this question also provides insight into the application of feature location research to industrial systems.

We chose simulation scenarios that deal with flight surveillance and simulation (FSS). These are the most interesting ones because they show alarms and warnings when the aircraft is exposed to serious danger such as the possibility of a crash. A buggy scenario that goes undetected can have devastating effects.

FSS is composed of three main subsystems. The first one, introduced in the previous section, TAWS, alerts the pilot about the terrain conditions below and above the aircraft. The second one is for detecting the traffic in the flight path and alerting the pilot when there is another aircraft in the way. This subsystem is called Traffic Collision Awareness System (TCAS). The third subsystem is for implementing the weather radar (WXR) which allows the pilot to monitor weather conditions.

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.

A. Simulation Scenarios

For this case study, we selected three features of the TAWS subsystem and two scenarios involving TCAS. The scenarios are described in Table 1.

TABLE 1. SIMULATION SCENARIOS USED IN THE STUDY

#	**Subsystem**	**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.
S4	TCAS	Simulate the presence of an intruder with the intention to locate its speed.

The first scenario is TAWS Mode1 which we used as a running example in the previous sections. The other two TAWS scenarios are: TAWS Mode 4A and Mode 4B. TAWS Mode 4A is activated when the aircraft is close to the ground and is prepared for landing, but the gears are on the up position. TAWS Mode 4B is activated when the aircraft is in landing mode but the flaps are in a flight position.

For TCAS, we created two scenarios that simulate the presence of an intruder in the flight zone of the airplane. An intruder could be another plane or any object that can disturb the normal operation of the plane. It is mostly the intruder's specification that causes TCAS to activate. In the first scenario, we exercised a scenario with the intention to locate the intruder by measuring its altitude. 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). Altitude and speed are both important measures to assess whether the presence of the intruder is considered dangerous.

B. Trace Generation

To generate traces, we used the PIN framework [11], a platform independent tracing tool. PIN supports both binary and code instrumentation. We favoured binary instrumentation in this case to avoid modifying the code. Table 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 2. For example, for Scenario 1 (S1), there are 720 labels. We were told by configuration designers that complex scenarios will 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.

TABEL 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

C. Applying the approach

We asked an expert configuration designer to create queries for each scenario. For TAWS scenarios, he drafted queries that contained keywords using the monitor's warning messages. However, for TCAS scenarios, the monitor does not display explicit textual warnings. It uses sound effects, lights, and illustrations to warn the pilot. For example, it activates an alarm for relative altitude informing the pilot that an obstacle is in close range. It shows the altitude of the aircraft itself and a flashing red light indicating "traffic ahead". The configuration designer drafted queries based on his experience with TCAS scenarios. Queries are not shown in this study because of the proprietary nature of CAE systems.

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

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

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

Table 3 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.

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.

TABLE 3. PRECISION AND RECALL

N_1: Number of labels detected by the approach; N_2: Number of valid labels detected by the approach; N_3: Number of valid labels for each scenario, provided by the expert.

Scenario	N_1	N_2	N_3	Precision: (N_2/N_1)	Recall: (N_2/N_3)
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 S2, the query resulted in two seed functions with the same rank. 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 S3, 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 on the ground, intruder transporter type, etc.).

D. 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 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.

E. Lessons Learned

We demonstrated 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 [1, 2, 4, 5, 16, 17], or access to the source code [6, 7, 8, 9, 12, 14]. 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.

5. Threats to Validity

We describe threats to validity in three categories: internal validity, construct validity, and external validity [18].

A threat to internal validity exists in the implementation of our approach. We have mitigated this threat by manually verifying the outputs. We have also used smaller simulation scenarios when testing the approach. We worked closely with configuration designers at CAE to verify our results.

A threat to construct validity exists in the use of user queries. It is possible to have queries that do not quite

reflect the invalid behaviour. This is most likely when the monitor does not display explicit warning messages. We have mitigated this threat by working with experienced users. We acknowledge, however, that we need to work towards defining a set of representative queries that can also benefit novices. This could be one direction for future work.

A threat to external validity exists in generalizing the results of this study to other systems, perhaps from another domain. We believe that the two-phase method of FELODE can be reused in other contexts. For example, one can use messages displayed on a GUI during a crash to locate seed routines.

6. Related Work

Feature location techniques can be grouped based on whether they use dynamic analysis, static analysis, or a combination of both. Despite the noticeable increase of attention to feature location research, we have not encountered any study that applies to industrial systems. Existing techniques have been mainly applied to open source systems.

Wilde et al. proposed to use multiple traces to locate the components that are most relevant to the traced feature [16, 17]. They used a set of feature-relevant and a set of non-feature relevant traces and compared them. The components that appeared in the first set and not in the other one were considered the most relevant ones [16, 17]. Eisenberg et al. [4] and Eisenbarth et al. [5] also used multiple traces but instead of comparing them, they used concept analysis to detect feature-related components by exploring the concept lattice. Antoniol et al. [1, 2] proposed a method where the number of traces was reduced to two (one exercising the feature and the other one irrelevant to the feature). They argued that two traces should be sufficient for feature location. Although these studies have been shown to provide good results, they require more than one trace. In the context of CAE, this means setting up more than one simulation scenario. This is simply impractical given the amount of work required.

Rohatgi et al. [14, 15] proposed to combine dynamic and static analyses. They used a single feature-trace and the component dependency graph as the sources of information for their feature location approach. First the distinct classes are extracted from the trace and then an impact score is assigned to each class. The impact score is calculated using the component dependency graph. The idea is that feature-specific classes are the ones that are called less by difference components of the system. Thus the classes with the least impact are likely to be relevant to the feature under study.

Single Trace and Information Retrieval (SITIR) is a feature location approach proposed by Liu et al. [9]. SITIR starts with a feature-trace. It then applies Information Retrieval techniques to trace components. It collects a corpus of textual information using the trace routines. Users can then insert a query and based on the similarity between the terms used in the query and the corresponding texts in the corpus, it ranks the results to extract the semantically most similar elements to the feature.

Hayashi et al. [7] used the combination of dynamic and static techniques. Their approach takes as input, 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 the terms in the query and the 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.

Hill et al. introduced, Dora [8], an approach which uses static and textual analysis to find feature-relevant elements. The first input of the approach is a query formulated by the user. Dora measures similarity between the query and the methods of the source code using term frequency - inverse document frequency metric (which we used in this study as well). The methods with the highest tf-idf score are marked in the program's call graph. Dora then explores the neighbors of the marked methods and assigns a relevance score to the neighbors and in the process detecting the feature relevant methods. Dora scores all the methods in the source code and then refers to only few of them as feature relevant.

There are two main differences between FELODE and the approaches presented in the above studies. First, we do not use source code information in formulating queries. This is particularly important in a heterogeneous environment such as CAE. The second difference is that FELODE uses a two-phase mechanism by first locating seed routines based on observed failures and then collecting the remaining routines. We believe that this approach is more suitable to debugging tasks. The above studies attempt to locate all the routines in one step. This would result in more routines than needed to find the cause of defects. These approaches are more useful for feature enhancement, where a general understanding of the feature implementation is necessary.

7. Conclusion and Future Work

In this study, we presented a feature location approach, called, FELODE, for locating simulation scenarios in configuration files. The study was performed at CAE. When applied to five simulation scenarios, we achieved in average 50% precision and 90% 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.

To build on this work, 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.

Acknowledgment:

This work is supported partially by CRIAQ (Consortium de recherche et d'innovation en aérospatiale du Québec), NSERC (Natural Science and Engineering Council of Canada), CAE Inc., and Opal-RT inc.

We would like to thank configuration designers at CAE for their input, active participation in the study, and valuable feedback.

8. References

[1]. 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.

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

[3]. 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.

[4]. 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.

[5]. 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.

[6]. 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.

[7]. 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.

[8]. 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.

[9]. 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.

[10]. 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.

[11]. 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.

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

[13]. 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.

[14]. 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.

[15]. 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.

[16]. 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.

[17]. 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.

[18]. Wohlin C., et al., Experimentation in Software Engineering: An Introduction. Norwell, USA: Kluwer Academic Publisher, 2000.