

Received 4 June 2022, accepted 30 July 2022, date of publication 8 August 2022, date of current version 26 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3197656

RESEARCH ARTICLE

Facilitating Reuse of Functions in Embedded Software

MD AL MARUF¹, AKRAMUL AZIM¹, (Senior Member, IEEE), AND OMAR ALAM² 

¹Department of Electrical, Computer and Software Engineering, Ontario Tech University, Oshawa, ON L1G 0C5, Canada

²Department of Computer Science, Trent University, Peterborough, ON K9L 0G2, Canada

Corresponding author: Md Al Maruf (md.maruf@ontariotechu.ca)

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) under Grant DG 2018-05494.

ABSTRACT Developing software-intensive embedded systems is a significant challenge as embedded systems have become more complex and dynamic for integrating various constraints (e.g., real-time performance, design, and functional safety). In general, software practitioners reuse code to reduce development time for implementing required functionalities. However, embedded software development still lacks code reuse on a large scale due to its intricate constraints. This paper presents an approach to identifying reusable functions and their relationships from the legacy embedded software for faster development reducing functionality testing time. A python tool is developed to automatically extract the reusable functions and present them in a feature model. The search time for finding reusable functions speeds up when they are placed as features in a feature model. We conducted experiments on three GitHub embedded software projects (elevator, infotainment, and health monitoring systems). The experimental results find that our proposed approach reduces the feature search time by 59.0%, 49.8%, and 74.5%, respectively, compared to the manual searching approach for 50 features.


INDEX TERMS Embedded software, reusability, feature model.

I. INTRODUCTION

Modern embedded software development includes a wide variety of constraints that need to be satisfied when they are configured. Networked systems for medical devices, anti-lock braking systems (ABS) in automotive applications and industrial control systems are examples of diverse embedded systems with varying software design requirements. Due to complex relationships among hardware and software components [1], it is difficult to introduce new changes to an application. Building embedded software from scratch is time-consuming, and it requires skilled developers to understand the software requirements. Therefore, there is a need for new approaches that facilitate reuse in embedded systems. Reusing legacy code has many benefits, including shorter time-to-market and improved quality.

Although code reuse is a common practice in software development, little attention has been paid to reusing code in embedded software systems. To facilitate code reuse, we need

to apprehend different constraints (e.g., resource allocation, timing, and safety) in developing embedded software. According to [2], highly configurable systems (e.g., embedded software) contain various constraints, and it requires extracting these constraints from software code to design valid configurations. This raises the need for a model that displays embedded software's integrated variants and constraints details to facilitate the functions verification and validation effort [3]. There are studies, e.g., [4], [5], [6] indicating that the constraints of embedded software are efficiently manageable and easily expressible with the help of a feature model. As prescribed by the ISO 26262 standard [7], the feature model can lift implementation-level dependencies to the level of features to understand the functional safety and feature variability of embedded software. Moreover, embedded software differs from traditional software for different characteristics such as interrupt handling using interrupt service routine (ISR), periodic timer requests, task scheduling, and language dependence. The existing studies [8], [9] show that over 80% of all embedded systems are developed using C procedural and function-oriented language. In many

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen .

cases, it is assumed that other languages are too memory or resource-intensive to be effectively deployed in bare-metal development. Furthermore, many embedded software source codes are publicly-accessible but do not have proper documentation for understanding the requirements to reuse. Therefore, developers often avoid reusing existing embedded software codes that are available in public repositories such as GitHub.

In recent years, many software engineering techniques such as programming language constructs or product line visualization have used object-oriented structures to develop software. However, embedded software largely concentrates on function-oriented design and development to perform its dedicated tasks. Existing studies show [10], [11] embedded software usually performs its specialized tasks in the form of functions that frequently interact with other functional units. For instance, the adaptive cruise control function of automotive embedded systems is distributed over multiple Electric Control Units (ECUs), and it repeatedly interacts with other functions spread across numerous ECUs [12]. Identifying these reusable functions is challenging when they appear in different locations of the code without following proper coding practices. Recent software reuse studies [13], [14] identify these functions as features for reuse in new software of interest. A feature is defined as a reusable function that satisfies any functional or non-functional requirements [15], [16] of a specific task. However, there is a lack of approaches that automatically extracts features from embedded source code. For example, recent approaches [13], [17], [18], [19], [20], [21], [22] manually extract features from the source code. It becomes a time-consuming and error-prone task while the feature details for large-scale embedded software are absent.

Therefore, this paper presents an approach to automatically identify reusable functions that pertain to implementing embedded software features. It uses a function call graph to rank the program functions and extracts calling relationships between subroutines, including function dependencies and constraints. Automatic identification of reusable functions benefits understanding software requirements and supports faster development with minimized risks than writing software from scratch. As identified functions are already tested for a legacy application, they will be less error-prone than the newly developed version.

The compact representation of the identified reusable functions assists more in apprehending the software requirements when we place them as features in a feature model. A feature model defines the features and their dependencies in a tree structure [23]. This model-based software development helps to visualize required features in a tree structure considering various aspects, including testability, code generation, traceability, reusability, testing workflow, constraints verification, and validation. The feature model allows feature-to-artifact mappings to illustrate how the system functions are implemented in terms of relationships, dependencies, behaviors, and constraints [24]. It supports reusability to improve

development time (e.g., feature search time, verification), which are major bottlenecks of releasing embedded software even after following agile methodology.

We have developed a python tool [25] that implements our approach to identify reusable functions from embedded source code and present them in a feature model using our proposed level-based model construction algorithm. It captures the application's available features and constraints to be considered for reuse. We apply our approach to different open-source C projects (vehicle's software controller [26], elevator [27], and health monitoring [27]) and evaluate it by comparing the search time needed to find a function with traditional approaches. The results show that searching for a feature in the feature model using our approach takes less time than searching it directly from the source code. The main contributions of this paper are:

- (a) Using call graph to identify reusable functions for a given embedded software program.
- (b) Constructing an embedded software feature model by creating relationships among reusable functions.

In summary, this paper contributes to the design and development of feature models from legacy code to improve embedded software's reusability. As traditional embedded software manufacturers require consolidating various features and constraints from scratch, our approach assists in reducing the development cycle by automatically extracting feature requirements from legacy software repositories. Designers and developers will be able to reuse the required functions with valid configurations and bring the product early to the market. Moreover, embedded functions' reusability can be beneficial by increasing productivity, accelerating development, and lowering operational costs.

The remainder of this paper is organized as follows. Section II defines the problem statement and Section III explains the system model and assumptions. Section IV demonstrates the details of the proposed method. To explain the proposed approach, we provide an illustrative example in Section V. In addition, Section VI presents the experimental results and analysis. Section VII states the related work of feature modeling in software engineering. We discuss the threats to validity in Section VIII. Finally, Section IX concludes the paper.

II. PROBLEM STATEMENT

Developing embedded software from scratch becomes challenging when there is a lack of understanding and visualizing system input, output, architecture, and function configurations. Thus, the current research necessitates finding reusable functions and creating a model representing their relationships. Our paper facilitates reuse in embedded systems by:

- (a) Reusable functions identification: Find reusable functions from legacy embedded software source code.
- (b) Function relationships identification: Identify relationships of reusable functions, including constraints to design and configure them in a feature model.

In identifying the reusable functions, it reuses embedded code that is publicly available in the embedded and IoT community. The process requires extracting functions that act independently and contribute to constructing features in embedded software. Therefore, we record function calls using a call graph and map their relations to configure them correctly in the feature model.

A. GOAL

The main goal of this paper is to enable the reusability of embedded software using the feature model. To achieve this goal, our process entails dividing it into smaller sub-goals by answering the following questions.

- 1) *How to model the relationships among reusable functions?:* This paper intends to identify function relationships and their dependencies, including constraints for correct configuration. Therefore, we construct a feature model to represent the commonality and variability of embedded software functions.
- 2) *Does the feature model speed up the feature search time?:* By constructing a feature model from legacy embedded software, this paper also aims to show the benefit of using this model to improve the software development time (e.g., feature search for locating erroneous code).

III. SYSTEM MODEL AND ASSUMPTIONS

An embedded system performs periodic or aperiodic tasks satisfying different system constraints. Each task is monitored and controlled by the software components. We define a task as a set of jobs that jointly perform some actions to produce the expected output. Thus, a job is a unified piece of work or function that performs particular action of a task. These jobs are implemented based on the application requirements. Each job can be divided into multiple sub-jobs, where a sub-job is a continuation of its parent job. These jobs are repeatedly used to meet the application’s functional (e.g., anti-lock brake system) and non-functional (e.g., performance) requirements.

We assume a feature is equivalent to a job j_i that is reused more than a threshold value (≥ 1) in embedded software. However, we consider that the domain expert sets a threshold value, and it defines the minimum number of reuse of a function to be considered a feature. A function implements the functional (e.g., correct output) and non-functional requirement (e.g., constraints or quality attributes) requirements of a feature. If a feature is called inside another feature, we define it as a sub-feature of that parent feature. Thus, each feature can have multiple sub-features. The inclusion/exclusion of sub-features depends on integrating their parent features in the feature model. Here, each feature f_i is defined as: $f_i = (f_i^{tp}, f_i^{sf}, V_i[], CL_i[])$, where

- f_i^{tp} is the type of feature relation [e.g., *mandatory, or, optional, alternative, or equivalent*],
- f_i^{sf} is the list of sub-features if any,
- V_i is the list of required variables and constants, and
- CL_i is the constraint list or property of feature f_i .

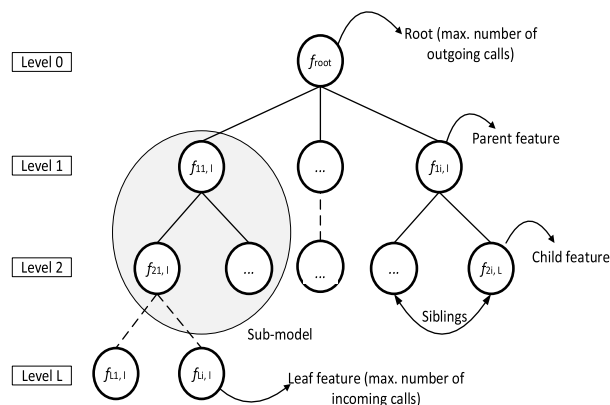


FIGURE 1. A sample representation of a feature model.

As a prerequisite to identifying reusable functions, we assume that the legacy source code of embedded software is given and the legacy code is a set of related versions of embedded software. We consider a function reusable if it has already been reused in the current code. We characterize the individual reusable function or feature as $f_i \in F(\text{set of features})$ and use regular expressions to extract the function’s name, global definitions, variables, constants, callee functions, caller functions, and the nature of the function calls (direct, transitive and recursive). The features are distributed into different levels of a tree based on our proposed level-based model construction steps discussed in subsection IV-C. Figure 1 shows an example of a feature model (FM) where the level identifies the number of incoming function calls of a feature. Therefore, the root feature has a minimum and the leaf feature has the maximum incoming calls.

IV. PROPOSED APPROACH

The proposed approach finds reusable functions and presents them in a feature model using the following steps:

- Identifying the reusable functions: It identifies the reusable functions from legacy embedded software code.
- Identifying the function relationships: It determines the function type and dependencies.
- Feature model construction: A new level-based feature model construction algorithm is described to map all the reusable functions as features in the feature model.

A. IDENTIFYING THE REUSABLE FUNCTIONS

In the reusable functions identification process, we generate a function call graph from a given legacy embedded software using GNU’s *cflow* graph generator [28]. The control flow graph extracts the functions invocation relationships and sub-routine information of the program. After that, we retrieve the number of functions, the signature of each function, the function’s definition, the location and the depth of function calling modifying *pycparser* python library [29]. A flow

graph can be very large because of a potentially long sequence of function calls (e.g., recursive function call). Thus, the depth of the function is labeled at which the flowgraph is cut off. To handle the recursive function as a typical function call, we place an extra ‘R’ symbol beside the function’s name. Our approach applies both static and dynamic analysis for building the control flow graph. The program’s variables, constants and constraints are obtained through a static analysis where function callee dependencies are retrieved using dynamic analysis. We conduct dynamic analysis using GCC compiler [30] that records the execution traces of the functions. In addition, we extract the internal, external, and static data tokens like keywords, operators, constants, and identifiers of the C program.

- Identify reusable functions: We identify a function if it appears more than a threshold number in a program. To visualize the reusable functions, we keep an optional threshold value: a positive integer number (e.g., $value \geq 1$). Developers can modify the threshold value and limit the function list to visualize from different abstraction levels. In this step, the process:
 - Finds the list of functions, including their parameters, variables and constants. A function is mapped to a standard form of $f_i \rightarrow f_j$, one-to-one correspondence with the source code.
 - Identifies the constraints analyzing different conditional statements such as if-else, while loop, switch condition and attribute’s properties from the Code and developer’s comments.

```

void navigate(int rotate) {
    set_direction(rotate);
    fly_normal();
}
void takeoff(float sp, float alt) {
    if(sp>0){
        navigate(rotate);
    }
    gps_tracker();
}
void landing(float distance) {
    safe_return(distance);
    gps_tracker();
}
float cal_altitude() {
    return gps_tracker();
}
void controller(int mode) {
    switch(mode) {
        case 1:
            takeoff();
            break;
        case 2:
            landing();
            break;
        default:
            fly_normal();
            break;
    }
}

```

Listing 1. C code 1.

We only retrieve the user-defined functions and skip the library functions by checking their function definitions to simplify our approach. The tool retrieves all the available functions as features by default if the developers do not set the threshold value. We show an example of C codes in Listing 1 and 2 to understand how a function is selected as feature. In these listings, let us assume that any function that appears more than once in the program is named a reusable

```

void navigate(int rotate) {
    if (rotate != NULL)
        set_direction(rotate);
    else
        fly_normal();
}
/*sp = speed, alt = altitude*/
void takeoff(float sp, float alt) {
    if (sp>0){
        navigate(rotate);
    }
    gps_tracker();
}
void landing(float distance) {
    safe_return(distance);
    gps_tracker();
}
float gps_tracker() {
    return get_coordinate();
}
float get_coordinate() {}
int main(){
    while(get_connect){
        /*use controller to change mode*/
        controller(mode);
    }
    landing(); /*upon battery charge */
}

```

Listing 2. C code 2.

feature. Since *navigate()*, and *gps_tracker()* are called more than once in this example, they are features. Similarly, if we assume *takeoff()* and *landing()* are both called more than once elsewhere in the code, they will be considered features too. Beside, we retrieve the constraints for feature *takeoff()* and *landing()*. For example, the variables *sp*, *mode*, and *rotate* are involved in validating certain constraints through if-else or switch statements to call other sub-features.

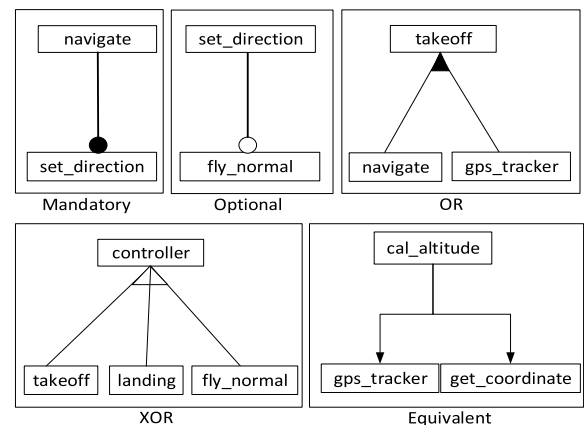


FIGURE 2. Feature types and relationships.

B. IDENTIFYING THE FUNCTION RELATIONSHIPS

In this step, the proposed approach defines different types of functions and their relationships. The process transforms the corresponding functions into features in the feature model of an embedded software. We demonstrate the relationships in Figure 2 for better visualization.

- **Mandatory:** A function is called a *mandatory* feature if and only if its parent node requires to execute it all the time. All its mandatory child features must also be included (n from n) in a parent feature. As an example, *set_direction()* and *fly_normal()* is a mandatory feature for *navigate()* in Listing 1 where *gps_tracker()*

and *safe_return()* are two mandatory sub-features under *landing()*.

- *Optional*: A function is an *optional* feature if it is selected with other available functions. It means an optional feature can frequently appear along with a particular feature, but it may appear with other features too. Any number of features can be added (m from n , $0 \leq m \leq n$) under a parent feature. For example, *fly_normal()* is an optional feature to *set_direction()* which is shown in Figure 2.
- *Alternative (XOR)*: A function is in *alternative (XOR)* feature group if it only gets selected among other functions based on certain conditions. In the case of *alternative/XOR* relationship, exactly one feature must be selected from a group of alternative features (1 from n). Thus, features *takeoff()*, *landing()*, and *fly_normal()* of Listing 1 are in *XOR* relationship (because of the switch statement).
- *OR*: A function is in *OR* relationship if its parent function selects at least one function out of multiple functions (m from n , $m > 1$). Thus, *navigate()* and *gps_tracker()* are in *OR* relationship in *takeoff()*.
- *Equivalent*: A function is equivalent to another function or combination of other functions when it shows the similar functionalities that exist in other features. Please refer to Figure 2.
- *Exclude*: If a particular function is selected from a group, other functions from the same group cannot be selected. In this relation, one function eliminates another function for selection.
- *Requires*: A *requires* relation indicates the dependency of one function (source) to another (target) function.

Moreover, we reorder the function relationships to make them more manageable for adapting to a feature model. For example, Listing 1 and 2 show that *navigate()* is a reusable feature and it is connected with other mandatory (e.g., *set_direction()*) and optional features (e.g., *fly_normal()*). Therefore, the partial feature model is as follow:

$$F' = (\textit{navigate}() \leftarrow \textit{set_direction}() \wedge \textit{fly_normal}()) \vee \textit{navigate}() \leftarrow \textit{set_direction}() \wedge \neg \textit{fly_normal}()$$

Let us consider a case where *fly_normal()* is considered as optional feature. In Listing 1, it is shown that *fly_normal()* appears with a mandatory feature *set_direction()* but with XOR relation in Listing 2. Thus, we define *fly_normal()* as an optional sub feature of *set_direction()*.

While both *navigate()* and *takeoff()* features show the possibility of reuse as combined or in an alternative to each other, we define them with a *OR* relationship. To define a *OR* relation, we assign the features under its parent feature, which is shown in the following partial model.

$$F' = \textit{takeoff}() \leftarrow (\textit{navigate}() \vee \neg \textit{navigate}()) \wedge \textit{gps_tracker}()$$

Furthermore, the dependencies among the features are simplified in tree-structured child-parent relationships. As an example, Figure 3(i) shows that feature *main()* has two mandatory features *controller()* and *landing()* where feature *landing()* is also a mandatory feature of *controller()*. Therefore, we define the tree as *main()* → *controller()* → *landing()*.

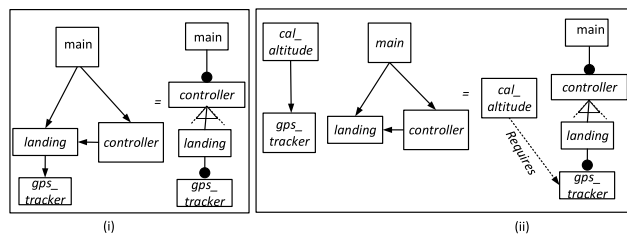


FIGURE 3. Simplification of function relationships for feature model.

However, a combination of multiple independent features can form another new feature. We define them as sub-features of the new feature in a sub-tree. In such a case, we assign the relation of that independent feature with the new feature using *Requires*. In Figure 3(ii), *cal_altitude()* is identified as an independent feature of a sub-tree where feature *gps_tracker()* is a part of the main tree. When the feature *cal_altitude()* calls *gps_tracker()*, it gets connected with *Requires* relationship.

C. FEATURE MODEL CONSTRUCTION

To construct the feature model, we propose a level-based algorithm that shows how the features are placed into different levels and what information is required for implementing a particular application. We implement a python tool for automating the proposed level-based feature model construction algorithm where a list of features is distributed with their associate feature types and constraints. At the beginning of this process, we determine the level of each feature using the call graph. Thus, features are mapped at each level based on the number of incoming function calls in the program. The hierarchy of function calls specifies the parent and child relationship. This relationship is identified from the caller and callee function list of the call graph. To find out the number of incoming function calls of each feature, we define a matrix named F^{in} . As an example, the matrix F^{in} represents a n -by- n matrix where n is the number of features. The degree of incoming function calls of each feature from every other feature is denoted by d_{ij} . It stores the corresponding degree of incoming function calls to feature f_i from f_j . To simplify the feature model, we set the number of incoming calls to equal one for any recursive function.

$$F^{in} = \begin{matrix} & \xrightarrow{\text{Features}} \\ & 1 & 2 & 3 & \dots & n \\ \begin{matrix} \downarrow \\ \text{Features} \end{matrix} & 1 & \left(\begin{matrix} d_{11} & d_{12} & d_{13} & \dots & d_{1n} \\ d_{21} & d_{22} & d_{23} & \dots & d_{2n} \\ d_{31} & d_{32} & d_{33} & \dots & d_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ d_{n1} & d_{n2} & d_{n3} & \dots & d_{nn} \end{matrix} \right) \end{matrix}$$

After that, we count the total incoming calls of each feature, adding all the corresponding row values using Equation 1.

$$f_i^{in} = \sum_{j=1}^n d_{ij}; \text{ where } d_{ij} = \begin{cases} x, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

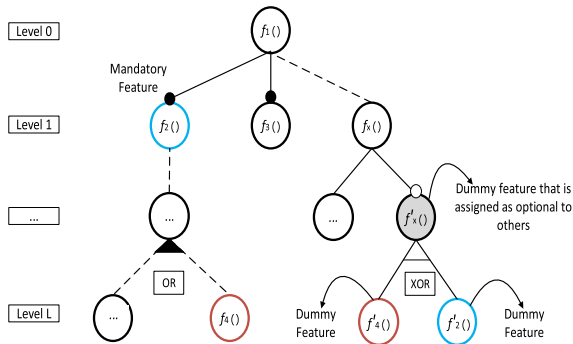


FIGURE 4. An implementation concept of feature model construction.

Finally, we use the following steps to construct the feature model in a tree structure.

- First, we assign the function as a root feature that has a minimum incoming or maximum outgoing function calls in the program and does not have any parent.
- Second, we place all the features at different levels according to the number of incoming calls. Therefore, the root feature is placed at level 0, intermediate features are between level 0 to L , and the leaf features with maximum incoming calls are placed into the last level L .
- Third, if any child feature is mandatory and located at a different level than the subsequent level of its parent, we can not include it in an *alternative (XOR)* relationship. We set the feature as a *dummy* feature for *XOR* relation under its parent feature. In Figure 4, we present an example for creating a *dummy* feature $f_2'()$ that replicates the mandatory feature $f_2()$ at Level L .
- Fourth, if any feature is in *OR* relationship, we can not put it under the *XOR* relationship in a different place. To combine it with the *XOR* relationship, we define it as a *dummy* feature that replicates the original feature. The created *dummy* feature $f_4'()$ with an *XOR* relation is shown in Figure 4.
- Fifth, if we create a *dummy* feature to build an *OR/XOR* relationship, we introduce their parent as *dummy* and make it as an *optional* feature to other features.

During the feature model construction, we traverse all the features from the lowest level L to the highest level 0. The distinct features of the lowest levels are considered leaf nodes in the feature model. We gradually decrease the level value to find the corresponding features mapped with its child features (callee function). The relation of each child feature is characterized by considering the feature types. Moreover, the *requires* and *excludes* relationships are identified from

Algorithm 1 Find the Levels of Reusable Functions and Their Parent-Child List

```

Input : Source code of embedded software
Output: Features levels including parent-child information
/* functions of embedded software */
1 featureList []= Extract functions from call graph
2 G = nx.DiGraph() /* call graph
3 fm = nx.DiGraph() /* feature model
/* finds the levels of features
4 level_list[][] = List of features at each level
5 for each feature i in range(len(featureList)) do
6   s=featureList[i];
/* finds the level of a feature
7   k=int(re.search(r' \d+', s).group())
8   level_list.append(k)
9 end
/* finds the parents of features
10 pr = [[] for x in range(len(featureList))]
11 for each feature i in range(len(featureList)) do
/* check the parent of a feature
12   nd = [x for x,y in G.nodes(data=True) if
13     y['value']==featureList[i]];
14   for j in range(len(nd)) do
15     for k in G.predecessors(nd[j]) do
16       pr[i].append(G.nodes[k]['value'])
17   end
18 end
/* finds the child of features
19 ch = [[] for x in range(len(featureList))]
20 for each feature i in range(len(featureList)) do
21   nd = [x for x,y in G.nodes(data=True) if
22     y['value']==featureList[i]];
23   for j in range(len(nd)) do
24     for k in G.successors(nd[j]) do
25       ch[i].append(G.nodes[k]['value'])
26   end
27 end
28 buildFeatureModel(featureList, level_list, ch, pr);

```

the sub-feature model relation. A feature is connected with *requires* relation if the feature is a *mandatory* feature at any level but also appears as a *mandatory* to a different level of another feature. On the other hand, the *excludes* relation identifies the feature that never appears with a specific feature that likely appears with all the common features.

The overall process of the level-based feature modeling is demonstrated in Algorithm 1 and 2. Algorithm 1 extracts the reusable functions as features through the call graph and then identifies each feature's level from the call graph. The parent and child feature list is classified considering the caller and callee relationship. After that, we add the feature relation and constraints to build the valid configurations in the feature model. Finally, Algorithm 2 builds the feature model in a tree format.

V. AN ILLUSTRATIVE EXAMPLE

To apply our proposed approach, we select a GitHub project "Software controller for vehicles" [26] written in C programming language. This GitHub project is the demo implementation of a car's software systems, and we use it to

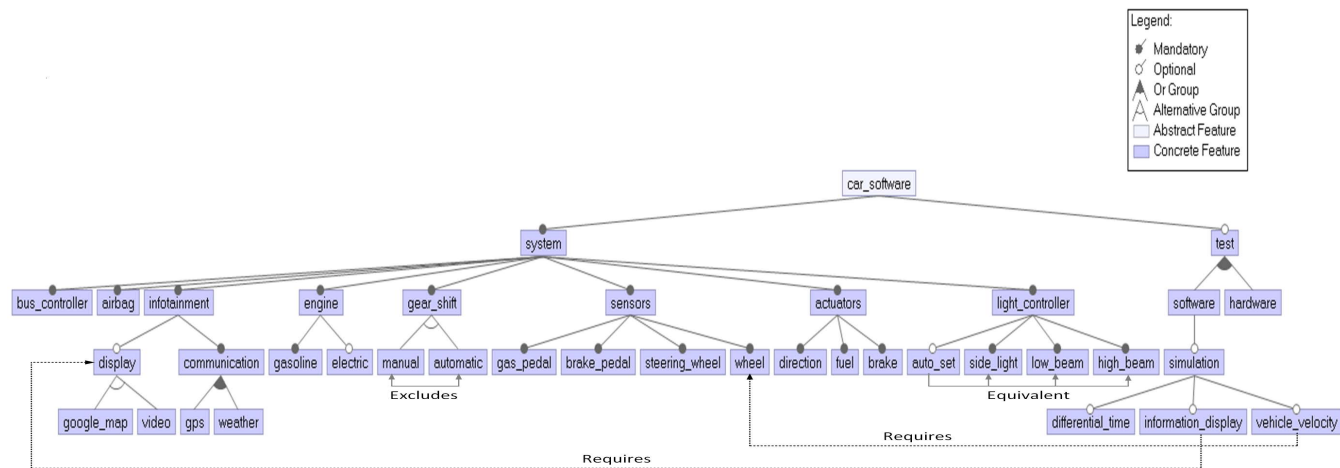


FIGURE 5. Feature model creation from GitHub project “Software controller for vehicles” [26].

Algorithm 2 Build Feature Model

```

Input : featureList [], level_list[[]], ch[], pr[], G, fm
Output: Feature model
1 Procedure buildFeatureModel (Input)
2   for each level l in range(len(level_list)) do
3     for each feature i in level l do
4       G.add_node(i,value=level_list[l][i])
5       if level (l - 1) finds child in list ch then
6         fm.add_node(level_list[l][i])
7         fm.add_edge(i,level_list[l-1][i])
8       else
9         fm.add_node(i,level_list[l][i])
10      end
11    end
12  end
13  for i in fm.edges.data() do
14    /* return relation type */
15    k=find_relation(i[0],i[1])
16    /* return constraints if any */
17    c=find_constraint(i[0],i[1])
18    fm.edges[i[0],i[1]]['relation']=k
19    fm.edges[i[0],i[1]]['constraint']=c
20  end
21  /* display feature graph */
22  drawNetwork(fm);
23 return
    
```

demonstrate our proposed method. To obtain each function’s information, we run the function flow graph generator *cflow* and extract the function list, including the depth of each call. The C keywords and library functions (e.g., malloc, fopen, printf) are ignored to simplify the process.

From the function list, we extract 38 reusable functions as features for the implementation of a car’s software systems. According to our proposed approach, initially, we find the mandatory and optional features where we retrieve 21 mandatory features and eight optional features. For example, feature ‘car_software’ has one mandatory feature ‘system’ and one optional feature ‘test’. After that, we identify the feature relationships defined in subsection IV-B and we get

four alternatives, four OR, and one equivalent feature relationships. The maximum level of each feature is extracted from the call graph and determines the potential parent and child list for each feature using Algorithm 1. After that, we apply the feature model construction actions discussed in subsection IV-C to update the placement of features in different levels if required. For example, the ‘auto_set’ feature under ‘light_controller’ calls three other features that together perform the same task. Therefore, these features are set under the equivalent features.

The output of our proposed approach for software repository [26] is shown in Figure 5. To beautify the visualization of the feature model, we use the java feature IDE, which is a feature modeling tool focusing on the java language. These features with relationships are fed into the java feature IDE tool and the final visualization is shown in Figure 5. The figure shows the association of each feature with its parent and child nodes. The parent-child relation hierarchy is displayed based on the level of each feature, where ‘car_software’ is placed at Level 0 as a root node. Besides, we find the requires and excludes relationships to understand the dependency of the features. To test the feature called ‘vehicle_velocity’, it needs requires relation with another feature ‘wheel’ that counts the wheels’ rotation. Similarly, ‘manual’ and ‘automatic’ features are mutually exclusive. One feature excludes other features. In the case of the equivalent feature, ‘auto_set’ is considered an equivalent feature to a combined three features like ‘side_light’, ‘low_beam’, and ‘high_beam’. The equivalent feature performs the same operations, but it may offer a different implementation.

A feature may have or may not have constraints. We use the C parser *pycparser* to generate the Abstract Syntax Tree (AST) and extract the code artifact along with constants, variables, as well as constraints. As an example, ‘infotainment’ systems and ‘light_controller’ are features without constraints. These features may be called periodically/apperiodically with user inputs. On the other hand,

features with constraints are ‘engine’, ‘gear_shift’, ‘airbag’, ‘brake’, etc. These features require validating certain constraints to ensure the safety of the system. The necessary test cases need to be executed before adding any feature in the model to satisfy the constraints.

Table 1 shows the list of features and their properties. The variables and constraints are retrieved from each feature’s implementation. As an example, the constraints are extracted from the conditional statement. The variable lists provide insight into understanding the inputs/outputs passed in the network’s communication messages. The features include their types and their relations that are defined in subsection IV-B. The *requires* and *excludes* feature relations can limit the possible choices of a set of features in the tree. In addition, Table 1 shows the list of sub-features. The sub-features are the list of features that are called inside of a feature to produce the expected output.

VI. EXPERIMENTAL RESULTS AND ANALYSIS

To evaluate the performance of the proposed approach, we compare the feature search time in the feature model to understand how quickly a feature can be located in the code.

Feature Search Time Analysis: To measure the time complexity of the proposed approach, we compare the traditional searching approach with the feature modeling approach for searching a feature. In the case of feature searching, the worst-case time complexity is $\mathcal{O}(n)$, where the best case is $\mathcal{O}(1)$. Here, n represents the number of features identified in embedded software. In the experiment, we collect three different GitHub projects which a) smart elevator system, b) infotainment system, and c) health monitoring system. We create an individual feature model for each application and calculate the search time to find a feature located in the deepest leaf node. The main reasons behind selecting these projects are the availability of source code and typical embedded software systems accepted by the community. Figure 6 shows the comparison of search time among these three different GitHub embedded systems projects. In all three projects, we observe that the search time for finding a selected feature using the feature model is much lower than the manual search in the codebase. The manual search reads the codebase sequentially, starting from the first line to the end. In Figure 6, the x-axis represents the number of features and the y-axis represents the required search time for finding a feature over a varying number of features.

Moreover, the experiment result states that the searching times vary for changing the number of features. Although the search time for a small embedded system application does not differ much, our approach outperforms the manual search approach for a large-scale project with a large number of features. In our approach, the feature search times for three different projects (smart elevator system [27], infotainment system [31], and health monitoring system [32]) get reduced by 59.0%, 49.8%, and 74.5% for 50, 25, and 50 number of features, respectively.

VII. RELATED WORK

Although many techniques are used mainly for identifying reusable units and requirements for developing an application from legacy code, the requirements are provided ahead as inputs or manually specified by users [16], [36]. A few approaches have focused on semi-automated feature extraction. However, they do not consider embedded software’s constraints and language dependency (e.g., written in C). Stefan Fischer *et al.* [37] presents an Eclipse-based framework to support the practice of “clone and own” by automatically extracting the reusable features from java applications. The clone and own process has three steps which are extraction, composition and compilation. The authors proposed automated extraction and composition to help the developer to point out the potential artifacts from previously developed products and map the relationship among those artifacts as well as store those information in a database for the composition of a new product. Moreover, for the compilation stage developer gets a hint from this tool if there is a lack of connection between two features.

In [18], a three-step process is introduced in the paper to identify features from the source code of product variants by using reverse engineering. In the first step, the product model is reverse engineered from the source code to reduce the noise which is created because of the different implementations of the same feature. Then each product model divides into a set of small pieces. In the second step, an algorithm is implemented to identify identical features. In the final step, manually eliminate the non-relevant candidates and add the missing features, if any. In another work, Omar Alam *et al.* propose [38] a concern-driven software development where they create a feature model considering the variability of the interfaces of the concerns (e.g., unit of modularization) instead of focusing on the functional definition of any component.

Jessie *et al.* [39] highlight the limitations of basic feature models also called boolean feature models which only represent boolean features. That means it only identifies if a characteristic is present or not in a software system. The authors proposed an extension of the existing boolean variability model that considers multivalued attributes or UML-like cardinalities to support variability modeling in complex product lines. Complex variability extraction algorithms are based on two mathematical frameworks: formal concept analysis and pattern structures. Finally, comparison matrices are implemented to represent the complex descriptions of software system families. The authors claim that their method is free from scalability issues and extracts all the potential relationships. It also contains numerous unexpected connections that require filtering. To compare our proposed approach with the existing related works, we present a comparison table in Table 2 specifying the research idea, input, output, techniques, language dependency, and application domain. We find most of the related works focus on object-oriented software (e.g., Java), but Muller *et al.* [13] shows how to extract features from C source code against a user given

TABLE 1. Extracted features and their properties for GitHub project “Software Controller for Vehicles” [26].

F_id	Feature	Type / Relation	Requires / Excludes	Variables and Constraints	Sub-features
1	System	Mandatory		V:{vehicle_status}	{bus_controller, airbag, infotainment, engine, sensors,actuators, engine, gear_shift, light_controller}
2	Test	Optional			{software, hardware}
3	bus_controller	Mandatory		V:{gas_pedal_pos, brake_pedal_pos, Comm_bus_message }	
4	airbag	Mandatory		V:{crash_threshold, inflate_speed} CL:{crash_threshold=10mph, inflate_speed=100mph}	
5	infotainment	Mandatory		V:{ICE_WARNING_DELAY}	{display, communication}
6	engine	Mandatory		V:{Temp_refresh_interval} CL:{Temp_refresh_interval=3000ul}	{gasolin, electric}
7	gear_shift	Mandatory		V:{eLimitedSpeed, minRPMs, redline} CL:{ Gear(0, 0, 0, true, eLimitedSpeed, minRPMs, redline), Gear(1, 35, 0, false, eLimitedSpeed, minRPMs, redline), Gear(2, 65, 8, false, eLimitedSpeed, minRPMs, redline), Gear(3, 95, 30, false, eLimitedSpeed, minRPMs, redline), Gear(4, 110, 45, false, eLimitedSpeed, minRPMs, redline), Gear(5, 125, 60, false, eLimitedSpeed, minRPMs, redline), Gear(6, 155, 85, false, eLimitedSpeed, minRPMs, redline), Gear(-1, -30, 0, false, eLimitedSpeed, minRPMs, redline) }	{manual, automatic}
10	actuators	Mandatory		V:{brake_actuator_pos, fuel_actuator_pos, direction_actuator_pos} CL: {event_action(time, period)}	{direction, fuel, brake, }
11	sensors	Mandatory		V: {vehicle_wheel_rotation, detection_time} CL: {collect_data()}	{gas_pedal, brake_pedal, steering_wheel, wheel_sensor}
12	light_controller	Mandatory		V:{light_status, light_pos}	{sidelight, low_beam, high_beam, auto_set}
13	gas_pedal	Mandatory		V:{gas_pedal_pos} CL: {change_speed(speed)}	
14	brake_pedal	Mandatory		V:{brake_pedal_pos}	
15	steering_wheel	Mandatory		V:{steering_wheel_pos}	
16	wheel	Mandatory		V:{avgWheelPeriod, wheelMode, wheel_angle} CL: {lockUpValue(), , wheel_rotation_count()}	
17	manual	Alternative	{Excludes: automatic}		
18	automatic	Alternative	{Excludes: manual}		
19	direction	Mandatory			
20	fuel	Mandatory		V:{fuel_actuator_pos}	
21	brake	Mandatory		V: {ADC_brake_value, brake_mode} CL: {EMERGENCY_STOP=1000, MIN_BRAKE=10, MIN_PERIOD=60, ADC_FREQUENCY=50}	
22	gasoline	Mandatory			
23	electric	Optional			
24	sidelight	Mandatory			
25	low_beam	Mandatory			
26	high_beam	Mandatory			
27	auto_set	Equivalent	{Requires: sidelight, low_beam, high_beam}		
28	vehicle_velocity	Optional	{Requires: wheel}	V:{wheel_rotation, distance}	
29	communication	Mandatory		V:{comm_type}	{gps, weather}
38	weather	OR		V:{location, weather_status}	

query input (e.g., related term) employing natural language processing. However, our approach just depends on source code and automatically identifies the highly reused functions.

VIII. THREATS TO VALIDITY

To understand the scope of our work, we report threats to the validity of our proposed approach.

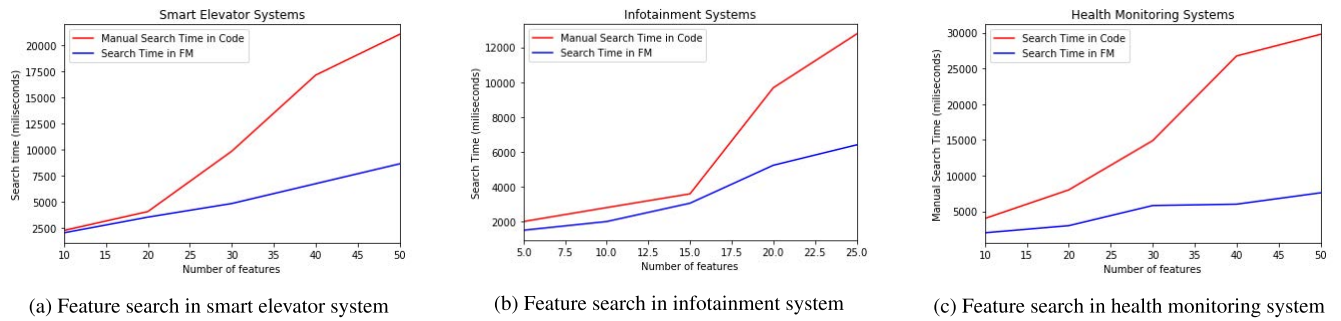


FIGURE 6. Search time comparison for finding a feature (deepest leaf node) among different GitHub embedded systems projects.

TABLE 2. Comparison of different reverse engineering approaches for identifying features.

Paper	Idea	Techniques	Inputs	Output	Language	Tools Used	Use Code Repository?	Case Study
J. Martinez et al. [33]	Extracting features from models for SPL	Model-driven (MoVa2PL)	Set of model variants	Feature, Feature model	Object Oriented (e.g., Java)	ArgoUML	No	Entertainment system
P. Muller et al. [13]	Domain features identification	Natural Language Processing (TDM)	Source code, query input (e.g., Term)	Features	C	Fex, grep	Yes	Tools written in C (e.g., inotify-tools)
S. Umair et al. [34]	Generating high level UML models	Model Driven (MDRE), Text-to-Model (T2M)	Source code	UML model	Object Oriented (e.g., Java)	UML generator	No	ATM, Amadeus Hospitality
Y. Tang et al. [35]	Feature Mining from legacy software	Conditional Probability	Feature model, Source code	AST	Object Oriented (e.g., Java)	Loong	Yes	Prevayler2, MobileMedia
Our approach	Extracts highly reusable features	Function Call graph, Static Analysis	Source code	Reusable function, Feature Model	C	cflow, pycparser	Yes	Software controller for vehicles, Elevator system, Health monitoring

External validity (Can we generalize the results for all embedded software?): We target only the embedded software written in C. In addition, the proposed method uses a function call graph as its principal to find reusable functions. Therefore, this approach is applicable to all function-oriented embedded software.

Internal validity (Does this approach identify all the necessary features?): The proposed approach identifies only those functions reused more than once in the code. Hence, there can be potential functions that are not reused in the code, but they could be a candidate for future reuse. These functions are skipped by our approach to identify only the highly reusable functions. However, our approach has the option to set the threshold value to a minimum for considering all the available functions in the code.

IX. CONCLUSION

Developing embedded software requires extensive testing to ensure the quality of the software by satisfying different constraints. Reusing legacy embedded software can improve the development process of finding reusable units. Therefore, this paper presents a new technique for faster development by identifying reusable functions from legacy software code. In addition, it shows the relationships placing the reusable functions as features in a feature model to visualize the application requirements. This work contributes toward the reverse-engineering of reusable function identification from large-scale embedded systems projects. The experimental results show that the generated feature model can drastically reduce the search time compared to traditional approaches.

REFERENCES

- [1] J. Polaczek and J. Sosnowski, "Exploring the software repositories of embedded systems: An industrial experience," *Inf. Softw. Technol.*, vol. 131, Mar. 2021, Art. no. 106489.
- [2] S. Nadi, T. Berger, C. Kästner, and K. Czarniecki, "Mining configuration constraints: Static analyses and empirical results," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 140–151.
- [3] H. Guissouma, C. P. Hohl, H. Stoll, and E. Sax, "Variability-aware process extension for updating cyber physical systems over the air," in *Proc. 9th Medit. Conf. Embedded Comput. (MECO)*, Jun. 2020, pp. 1–8.
- [4] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarniecki, "The variability model of the Linux kernel," *VaMoS*, vol. 10, no. 10, pp. 45–51, 2010.
- [5] J. White and D. C. Schmidt, "Automated configuration of component-based distributed real-time and embedded systems from feature models," *IFAC Proc. Volumes*, vol. 41, no. 2, pp. 6891–6896, 2008.
- [6] D. Streitferdt, P. Sochos, C. Heller, and I. Philippow, "Configuring embedded system families using feature models," in *Proc. Net. ObjectDays*, 2005, pp. 339–350.
- [7] T. Groß, J. Moore, and J. Lee, "Planning software architecture and modeling patterns for ISO 26262 compliance," in *Proc. Embedded World Conf. Germany: Nuremberg Exhibition Centre*, 2020.
- [8] C. Ünsalan, H. D. Gürhan, and M. E. Yücel, *Embedded System Design With ARM Cortex-M Microcontrollers: Applications with C, C++ and MicroPython*. Cham, Switzerland: Springer, 2022.
- [9] B. P. Douglass, *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*. Amsterdam, The Netherlands: Elsevier, 2010.
- [10] J. J. Labrosse, "Operating systems," in *Software Engineering for Embedded Systems*. Elsevier, 2019, pp. 153–206.
- [11] V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yılmaz, "Testing embedded software: A survey of the literature," *Inf. Softw. Technol.*, vol. 104, pp. 14–45, Dec. 2018.
- [12] G. Xie, G. Zeng, Z. Li, R. Li, and K. Li, "Adaptive dynamic scheduling on multifunctional mixed-criticality automotive cyber-physical systems," *IEEE Trans. Veh. Technol.*, vol. 66, no. 8, pp. 6676–6692, Aug. 2017.
- [13] P. Müller, K. Narasimhan, and M. Mezini, "FEX: Assisted identification of domain features from C programs," in *Proc. IEEE 21st Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2021, pp. 170–180.

- [14] G. K. Michelon, L. Linsbauer, W. K. G. Assunção, S. Fischer, and A. Egyed, "A hybrid feature location technique for re-engineering single systems into software product lines," in *Proc. 15th Int. Work. Conf. Variability Model. Software-Intensive Syst.*, Feb. 2021, pp. 1–9.
- [15] J. Krüger, W. Mahmood, and T. Berger, "Promote-pl: A round-trip engineering process model for adopting and evolving product lines," in *Proc. 24th ACM Conf. Syst. Softw. Product Line*, Oct. 2020, pp. 1–12.
- [16] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 210–224, Mar. 2003.
- [17] N. Islam and A. Azim, "Feature characterization for CPS software reuse," in *Proc. 10th ACM/IEEE Int. Conf. Cyber-Physical Syst.*, Apr. 2019, pp. 314–315.
- [18] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, "Feature identification from the source code of product variants," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2012, pp. 417–422.
- [19] A. Burger and S. Gruner, "FINALIST2: Feature identification, localization, and tracing tool," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2018, pp. 532–537.
- [20] R. Williams, T. Ren, L. De Carli, L. Lu, and G. Smith, "Guided feature identification and removal for resource-constrained firmware," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, pp. 1–25, Apr. 2021.
- [21] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, "Reverse engineering architectural feature models," in *Proc. Eur. Conf. Softw. Archit.* Berlin, Germany: Springer, 2011, pp. 220–235.
- [22] S. Greiner, M. Nieke, and C. Seidl, "Towards trace-based synchronization of variability annotations in evolving model-driven product lines," in *Proc. 16th Int. Work. Conf. Variability Modeling Software-Intensive Syst.*, Feb. 2022, pp. 1–10.
- [23] L. Li, Y. Zheng, M. Yang, J. Leng, Z. Cheng, Y. Xie, P. Jiang, and Y. Ma, "A survey of feature modeling methods: Historical evolution and new development," *Robot. Comput.-Integr. Manuf.*, vol. 61, Feb. 2020, Art. no. 101851.
- [24] D. Hinterreiter, K. Feichtinger, L. Linsbauer, H. Prähofer, and P. Grünbacher, "Supporting feature model evolution by lifting code-level dependencies: A research preview," in *Proc. Int. Workshop Conf. Requirements Eng., Found. Softw. Quality*. Cham, Switzerland: Springer, 2019, pp. 169–175.
- [25] M. A. Maruf. (2022). *Automatic Feature Extractor*. Github Repository. [Online]. Available: <https://figshare.com/s/d4bf19bc3818f1421ab2>
- [26] U. T. P. Coser. (2020). *Software Controller for Vehicles*. Github Repository. [Online]. Available: <https://github.com/PCoser/Software-Programing-in-C>
- [27] J. Liang. (2016). *Elevator Simulation*. Github Repository. [Online]. Available: <https://github.com/JustinLiang/ElevatorSimulation>
- [28] P. Wu, J. Wang, and B. Tian, "Software homology detection with software motifs based on function-call graph," *IEEE Access*, vol. 6, pp. 19007–19017, 2018.
- [29] E. Bendersky. *Pycparser*. Github Repository 2020. Accessed: Jun. 1, 2022. [Online]. Available: <https://github.com/eliben/pycparser>
- [30] R. M. Stallman, "GNU compiler collection internals," *Free Softw. Found.*, vol. 4, no. 2, Dec. 2003.
- [31] Y. Malinov. *Car infotainment*. Github Repository 2015. Accessed: May 20, 2022. [Online]. Available: <https://github.com/YMalinov/car-infotainment.git>
- [32] V. Thakur. *Health Monitoring System*. Github Repository 2020. Accessed: May 20, 2022. [Online]. Available: <https://github.com/vikrant-developer/health-monitoring-system>
- [33] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. Le Traon, "Automating the extraction of model-based software product lines from model variants (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 396–406.
- [34] U. Sabir, F. Azam, S. U. Haq, M. W. Anwar, W. H. Butt, and A. Amjad, "A model driven reverse engineering framework for generating high level UML models from Java source code," *IEEE Access*, vol. 7, pp. 158931–158950, 2019.
- [35] Y. Tang and H. Leung, "StiCProb: A novel feature mining approach using conditional probability," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2017, pp. 45–55.
- [36] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-up adoption of software product lines: A generic and extensible approach," in *Proc. 19th Int. Conf. Softw. Product Line*, Jul. 2015, pp. 101–110.
- [37] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "The ECCO tool: Extraction and composition for clone- and-own," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 665–668.
- [38] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *Proc. Int. Conf. Model Driven Eng. Lang. Syst.* Berlin, Germany: Springer, 2013, pp. 604–621.
- [39] J. Carbonnel, M. Huchard, and C. Nebut, "Towards the extraction of variability information to assist variability modelling of complex product lines," in *Proc. 12th Int. Workshop Variability Modeling Software-Intensive Syst.*, Feb. 2018, pp. 113–120.



MD AL MARUF is currently pursuing the Ph.D. degree with the Department of Electrical, Computer and Software Engineering, Ontario Tech University, ON, Canada. His research interests include real-time embedded systems, software engineering, software reuse, task scheduling, and machine learning.



AKRAMUL AZIM (Senior Member, IEEE) is currently an Associate Professor in software engineering with Ontario Tech University, Oshawa, ON, Canada. His research interests include real-time embedded software, safety-critical software, machine learning, software engineering, cognitive computing, and intelligent transportation systems. He is also a Professional Engineer in Ontario.



OMAR ALAM is currently an Associate Professor with the Department of Computer Science, Trent University. His research interest includes software engineering. In particular, he is interested in model-driven engineering, aspect-oriented modeling and software reuse. He has published in premier venues in model-driven engineering and software engineering, such as MODELS, SPE, SLE, ICSR, and SAM. He served as a reviewer and a program committee member for various journals and conferences in the field of model-driven engineering and software engineering.

...