2024 IEEE International Conference on Information Reuse and Integration for Data Science (IRI) | 979-8-3503-5118-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IRI62200.2024.00058

FeaMod: Enhancing Modularity, Adaptability and Code Reuse in Embedded Software Development

Md Al Maruf*, Akramul Azim*, Nitin Auluck†, and Mansi Sahi†
* Department of Electrical, Computer and Software Engineering, Ontario Tech University, Ontario, Canada
†Department of Computer Science and Engineering, Indian Institute of Technology, Ropar, India

***Abstract*—The increasing prevalence of embedded systems in Cyber-Physical Systems (CPS) and the Internet of Things (IoT) has amplified the necessity for effective and adaptable software development practices. The challenges encountered in designing and developing these systems stem from the requirement to efficiently integrate advanced computational paradigms like machine learning and fog computing. Their inherent complexity and rigidity often limit the systems' adaptability to evolving requirements and complicate the effective management of feature dependencies, versioning, customization, and configuration in distributed environments.**

To address these challenges, we propose the FeaMod framework, integrating feature-based modularity with adaptive feature modeling for enhanced efficiency in embedded software design. Using the Bidirectional Encoder Representations from Transformers (BERT) model, FeaMod employs automated feature extraction through advanced static code analysis, facilitating the identification of computational features and requirements from existing codebases. These features are encapsulated in an adaptive feature model (AFM) that encourages code reuse and allows for dynamic configuration and system integration. By introducing a set of rules governing feature relationships, our approach ensures the adaptive nature of the model, enhancing its flexibility in response to changing system requirements, user preferences, and varying environmental conditions.

***Index Terms*—Embedded Software, Cyber-Physical Systems, Software Reuse, Feature Model**

I. Introduction

Embedded systems in Cyber-Physical Systems (CPS) and the Internet of Things (IoT) are pivotal in shaping an interconnected world, influencing sectors such as autonomous vehicles, smart cities, and precision agriculture [1]. Despite their vast potential, they encounter formidable challenges, especially in integrating advanced computational paradigms like machine learning and fog computing. The current state-of-the-art [2], [3] struggles to fully leverage these paradigms, resulting in rigid and difficult-to-adapt or reuse systems. The process of integrating new features into legacy applications further amplifies these challenges. It demands a delicate balance between innovation and preservation, requiring a deep understanding of the existing codebase to ensure compatibility and stability while promoting modernization. The need to plan for modularity, assess feasibility, and implement features incrementally calls for a structured framework to facilitate this complex process while minimizing potential risks. For instance, the inherent system complexity and rigidity of many existing embedded systems often limit their adaptability to dynamic requirements. Effective management of feature dependencies and configuration in distributed contexts [4], are additional concerns that are often inadequately addressed.

In response to this clear need for improvement in the field, we introduce FeaMod, a framework that incorporates feature-based modularity and adaptive feature modeling within the design and development of embedded software promoting code reuse. Central to our approach is the idea of automated transformation of non-modular code to a feature-based modular code and feature identification from source code and relevant comments. This transformation is made possible through advanced static code analysis, leveraging the BERT model for precise feature extraction. The inclusion of the BERT model in FeaMod enhances the accuracy of feature identification, ensuring that the extracted features are contextually relevant and comprehensive, leading to a more robust and efficient modularization process. By facilitating the identification and abstraction of computational features from legacy codebases, our approach provides an environment of software reuse and dynamic configuration while encouraging the creation of designs that are flexible, maintainable, and testable [5]. This approach, in turn, reduces development time and cuts costs, paving the way for a more efficient software development life cycle.

FeaMod's unique adaptive feature model encapsulates these computational features, allowing for dynamic configuration and system integration based on evolving requirements. In the complex world of embedded systems, adaptability is of the essence. To ensure this, we introduce a set of rules governing feature relationships within the model. This set of rules forms the backbone of our approach, allowing the model to respond flexibly to changing system requirements, user preferences, and environmental conditions. Moreover, it aids developers in a guided pathway to navigate the complex landscape of embedded system development, fostering a more resilient and adaptable system design.

Considering the potential of advanced computational paradigms, FeaMod focuses on the integration of feature versioning and distributed execution to enhance the performance and efficiency of embedded systems. The main contributions of this paper are as follows:

- We propose FeaMod, a framework that integrates feature-based modularity and adaptive feature modeling for efficient embedded software design and code reuse, aided by the precision of the BERT model.
- We introduce a set of rules that identifies features, relations, and constraints from source code to construct AFM, ensuring adaptability and flexibility in response to changing system requirements.
- We emphasize the integration of advanced computational

979-8-3503-5118-7/24/$31.00 ©2024 IEEE
DOI 10.1109/IRI62200.2024.00058

paradigms, such as machine learning and fog computing, via the modularization of these features, leading to more efficient development, testing, and reliable systems.

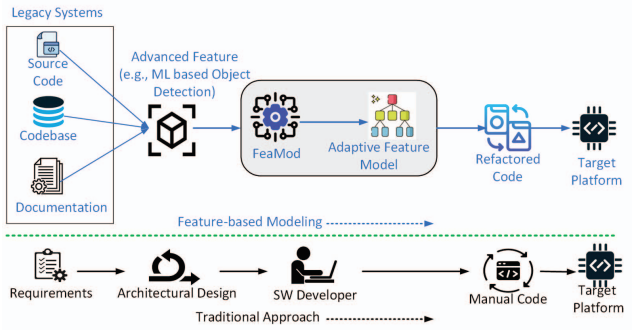

Figure 1: System Model Architecture (Feature-based vs Traditional Software Development Approach)

II. System Model and Assumptions

Our system model, shown in Figure 1, is composed of various components that interact together to understand the FeaMod framework. This *FeaMod* feature-based modeling framework is distinct from traditional software development models as it integrates existing legacy system code and employs automated feature extraction, modularity, and adaptability techniques to transform the code into a dynamic, easy-to-maintain format. In addition, reusing existing code substantially reduces both design and development efforts, providing significant efficiency gains.

Feature: Transitioning from the foundational concept of modularity, the concept of a *feature* in embedded software, denoted as a distinct unit of functionality encapsulated in a computational unit or function [6]. Features represent functional or non-functional characteristics, defining a system's behavior or quality attributes, respectively. Features, in essence, are the identifiable characteristics or capabilities of a system, often driven by user requirements that typically denote the specified criteria or constraints to which a system's feature must conform.

Adaptive Feature Model (AFM): Extending the static nature of traditional feature models, the AFM offers dynamic adaptability, allowing features to modify functionalities in real time based on various contextual variables, enhancing the system's responsiveness to user requirements and environmental conditions.

A. Assumptions

- We target non-modular Python programs used in advanced tasks such as machine learning-based object detection. Although C is prevalent in embedded systems [7], we opted for Python, given its growing relevance in advanced paradigms.
- Utilizing the CodeBERT, a derivative of the BERT model fine-tuned for code analysis, leverages its strength in understanding code context and language tasks, thereby offering a robust solution for our feature extraction processes based on its proven proficiency in contextual embeddings derived from natural language understanding.
- The codebase can be parsed into an Abstract Syntax Tree (AST) using the Python `ast` or similar module.

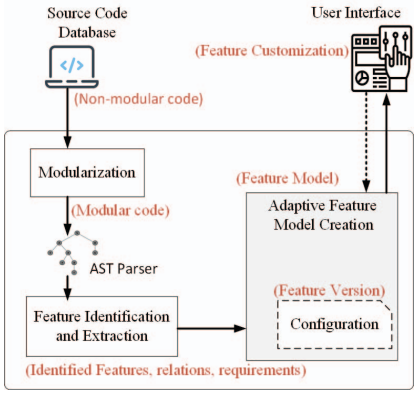

Figure 2: Proposed FeaMod Framework

III. Proposed Framework and Methodology

The FeaMod framework utilizes modular and feature-oriented programming principles to identify features and requirements from source codes of advanced computational paradigms. This adaptive methodology, comprising modularization, feature identification, and adaptive feature model creation, is delineated in the subsequent subsections and illustrated in Figure 2.

A. Modularization: Non-Modular Code to Modular Code

In the initial phase of the FeaMod process, non-modular code is segmented into discrete modules using Python's AST module, which parses the source code into an abstract syntax tree. This tree aids in identifying and separating distinct tasks or related groups of tasks into individual functions, thus preserving the original logic while enhancing the code's readability and maintainability. The approach is designed to align with the following rules defined for transforming non-modular code into modular code.

Rule 1 - Reusability: Abstract similar code segments using metrics like cosine similarity on AST representations, facilitated by clone detection techniques.

Rule 2 - Cohesion: Group code performing a single task into one module, guided by variable usage frequency, function calls, and other dependencies metrics.

Rule 3 - Encapsulation: Bundle related data and functions into units using metrics such as Data Abstraction Coupling (DAC) that focus on the count of abstract data types in a unit.

Rule 4 - Conditional Segmentation: Modularize operations in conditional constructs using cyclomatic complexity to identify targets for modularization.

Rule 5 - Loop Abstraction: Abstract repetitive tasks in loops into separate functions, targeting complex structures identified through loop depth metrics.

Rule 6 - Loose Coupling: Enhance code independence by reducing dependencies between code parts, guided by Coupling Between Object (CBO) metrics.

Rule 7 - Error Handling: Standardize error handling by encapsulating common try-except blocks, using frequency metrics to streamline common patterns.

Rule 8 - Grouped Imports: Consolidate common imports into single modules, employing frequency metrics to create logically grouped modules with descriptive names.

```
def load_model(): # Rule 1
    return cv2.dnn.readNetFromCaffe("deploy.prototxt
    ", "res10_300x300_ssd_iter_140000.caffemodel")

def load_image(image_path): # Rule 1
    return cv2.imread(image_path)

def process_image(image): # Rule 2
    return cv2.dnn.blobFromImage(image, 1.0, (300,
    300), (104.0, 177.0, 123.0))

def detect_objects(model, blob): # Rule 2
    model.setInput(blob)
    return model.forward()
# Rule 4 and Rule 5
def display_detections(image, detections):
    for i in range(detections.shape[2]):
        ...
    cv2.imshow("Output", image)

def mean_confidence(detections): # Rule 2
    return np.mean([detections[0, 0, i, 2] for i in
    range(detections.shape[2])])

def display_mean_confidence(image_path,
    mean_confidence): # Rule 7
    try:
        ...

def main(image_path): # Rule 6: Loose Coupling
    model = load_model()
    image = load_image(image_path)
    blob = process_image(image)
    detections = detect_objects(model, blob)
    ...

main("input.jpg")
```

Listing 1: Refactored Modular Object Detection Code Snippet

In the transforming, FeaMod converts a non-modular code segment into modular functions like `load_model()` and `load_image(image_path)`, as detailed in Listing 1 and [8]. By applying transformation *Rules* (1 to 8) and leveraging Python's `ast` module, we parsed the code into an Abstract Syntax Tree (AST), identified reusable segments, and encapsulated them into new functions. These functions were then reintegrated into the AST, enhancing the code's reusability and maintainability. The modified AST was converted back to source code using the `astunparse` module, resulting in distinct functions for operations such as loading a machine learning model (`load_model`), reading an image file (`load_image`), processing the image (`process_image`), and identifying objects in the processed image (`detect_objects`).

B. Feature Identification and Requirements Extraction

1) Identifying Functional Features: Every function in the codebase denotes a distinct feature, representing a unique operation or functionality within the system. Leveraging the BERT model fine-tuned with domain-specific semantics aids in extracting potential features from function names and global variables, creating a semantic space to identify functional and non-functional requirements. We apply established classifications to characterize these features systematically [6], i.e., *Mandatory*, *Optional*, *OR*, *XOR*, *Requires*, and *Excludes*. The FeaMod framework uses a set of rules to categorize features into:

- Mandatory: A feature that must be included whenever its parent feature is selected (e.g., `load_model`).
- Optional: May or may not be present depending on the conditions (e.g., `display_mean_confidence`).
- OR: One or more among the child features must be selected if the parent is included.
- XOR (Alternative): Exactly one among the child features can be chosen when the parent is selected.
- Requires: The presence of one feature necessitates the presence of another.
- Excludes: The presence of one feature prohibits the inclusion of another.

Algorithm 1: Identifying Non-Functional Features

Input: $D = \{d_1, d_2, ..., d_n\}$, a set of documents including code artifact, associated comments
Output: Potential non-functional features

for *each* d_i *in* D **do**
  Tokenize d_i into separate identifiers (tokens);
  Create a dictionary $Dict_{d_i}$ mapping each token;
  Calculate TF-IDF score for each token in $Dict_{d_i}$;
  Select the top N tokens based on the TF-IDF scores as candidate features; set is CF_{d_i};
end
Fine-tune BERT using domain-specific data (if available);
for *each* d_i *in* D **do**
  for *each token* t_j *in* CF_{d_i} **do**
    Create an embedding E_{t_j} using BERT;
  end
  Cluster the embeddings $\{E_{t_1}, E_{t_2}, ..., E_{t_n}\}$ to group semantically similar tokens, obtaining clusters $C_1, C_2, ..., C_m$;
  for *each cluster* C_k **do**
    Identify the representative token RT_{C_k} from C_k as a potential non-functional feature;
  end
end
return *Non-functional features for each document*;

2) Identifying Non-Functional Features: To identify non-functional features in software artifacts, we employ a fine-tuned BERT model, taking advantage of its deep understanding of context and semantics, as detailed in Algorithm 1. The process starts with the tokenization of each document d_i in set D, followed by forming a dictionary $Dict_{d_i}$ to map each token to its frequency in d_i. We then use TF-IDF scores to select the top N tokens as candidate features (CF_{d_i}) [9].

Next, the BERT model creates embeddings E_{t_j} for each token in CF_{d_i}, which are clustered to group semantically similar tokens. A representative token RT_{C_k} from each cluster is chosen as a potential non-functional feature.

Example: For instance, in a source code with frequent mentions of the term `security`, our method initially recognizes it as a candidate feature through high TF-IDF scoring. BERT then discerns different contexts of the term, like `data security` and `network security`, and groups them under the broader feature - `security`.

3) BERT-enhanced Requirements Extraction: Leveraging BERT's semantic analysis capabilities, we delineate a strategy for extracting both functional and non-functional requirements efficiently from software artifacts.

- Functional Requirements Extraction: These encapsulate the core functionalities that the system is designed to execute.
  - Extracting Conditional Constructs: We identify functional requirements by analyzing conditions in `if`, `while`, and `for` loops using BERT, facilitating a deep semantic understanding.
  - Function and Method Analysis: We utilize BERT to interpret function names and method signatures, helping to identify potential functionalities even in ambiguous scenarios.
- Non-functional Requirements Extraction: These involve system attributes such as performance and security.
  - Extracting Comments: BERT assists in analyzing comments semantically to pinpoint non-functional requirements effectively.
  - Documentation Analysis: Non-functional aspects are often detailed in READMEs and architectural documents.
  - Code Patterns and Libraries: Specific code patterns and libraries can signal non-functional requirements, such as security considerations.

C. Adaptive Feature Model Creation

The Adaptive Feature Model (AFM) creation facilitates dynamic reconfiguration of feature models, offering enhanced flexibility and efficiency by adapting to variations in system state and other factors. Considering a system with features denoted as f_i, we introduce the Feature Interaction Graph (FIG) $G = (V, E)$ to represent feature interactions, where where the vertices $V = f_1, f_2, \ldots, f_n$ represent features, and edges $E = e_{ij}$ represent interactions between features. For each feature f_i, we define an interaction set I_i, which includes all the features that interact with f_i. The construction of the AFM involves:

1) **Construction of the FIG:** Identifying feature interactions through code analysis and representing them in the FIG.
2) **Weight Assignment:** Quantifying feature interactions using metrics such as call frequency or parameter interactions and assigning weights to the edges in the FIG.
3) **Feature Clustering:** Utilizing graph clustering algorithms like K-means to identify groups of closely related features, which form subsystems in the final model.
4) **Feature Configurations:** Each feature f_i has an associated set of possible versions, denoted as $FC_i = \{v_{i1}, v_{i2}, \ldots, v_{im}\}$. Given the dynamic nature of software systems and ever-evolving requirements, feature versioning is a critical component in the feature model, enabling it to manage complex, changing requirements in distributed systems efficiently. A feature model configuration is considered valid if it includes all the feature configurations on which it depends.

Feature Model Attribute	Case Study
Number of functional Features	22
Number of Non-Functional Features	2
Number of Constraints	2
Number of Valid Feature Configurations	42

Table I: Feature Model Statistics for Face Detection [10]

IV. Experimental Result and Analysis

To evaluate the FeaMod framework, we utilized the publicly accessible Face Detection GitHub repository [10] as a case study. The analysis focused on the framework's proficiency in identifying features, recognizing feature relationships, and setting up configurations, coupled with assessing the adaptive feature model's construction time.

A. Case Study (Face Detection)

The FeaMod, powered by the Graphviz library for visualization, employed a series of steps to generate the application's Feature Model. Initially, the code was modularized to isolate specific features and make their functions more distinct and manageable. Subsequently, FeaMod identified the key features of the application, such as *DeviceSelection*, *Open_image_capture*, *Read_Model*, *Preprocess_image*, *FaceDetectionRecognition*, *Output*, and *Video Writer*. These features were categorized as mandatory, optional, or non-functional based on their roles within the application. Two non-functional features, *Performance Metrics* and *Logging*, were identified using the TF-IDF and BERT as discussed in Algorithm 1. To validate the effectiveness of our approach in identifying the features and associated requirements, a cosine similarity analysis was performed, the results of which are depicted in Figure 3. The diagonal elements of the cosine similarity matrix, representing the similarity scores between each feature and its corresponding requirement, were notably higher. This indicates a strong correlation between the identified features and their corresponding requirements, highlighting the accuracy of the proposed method in capturing the semantic relationship between features and requirements.

In this face detection case study, the manual approach identified 26 features and 39 requirements. Our BERT-enabled approach successfully identified 22 of these features and 34 requirements, demonstrating a balanced performance with high precision and recall, as detailed in Figure 4. This approach outperforms the rule-based method, achieving F1 scores of 86.5% and 86% for feature and requirement identification, respectively, and showcasing its effectiveness in deriving insightful results through the automated analysis of source code and comments. This underlines the potential of our approach in facilitating more accurate and comprehensive feature and requirement identification in software development.

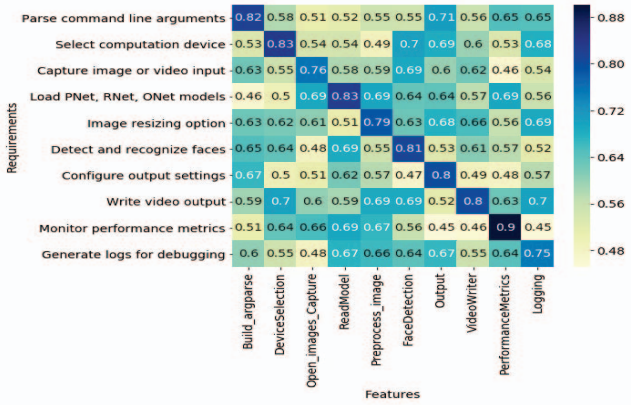

Figure 3: Features and Requirements Map for Face Detection

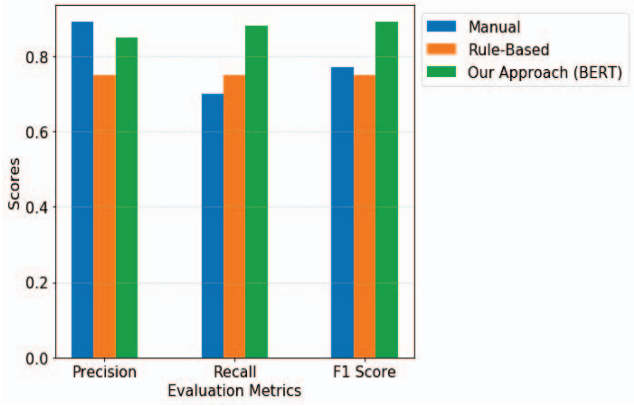

Figure 4: Comparison of BERT with Rule-based Approach

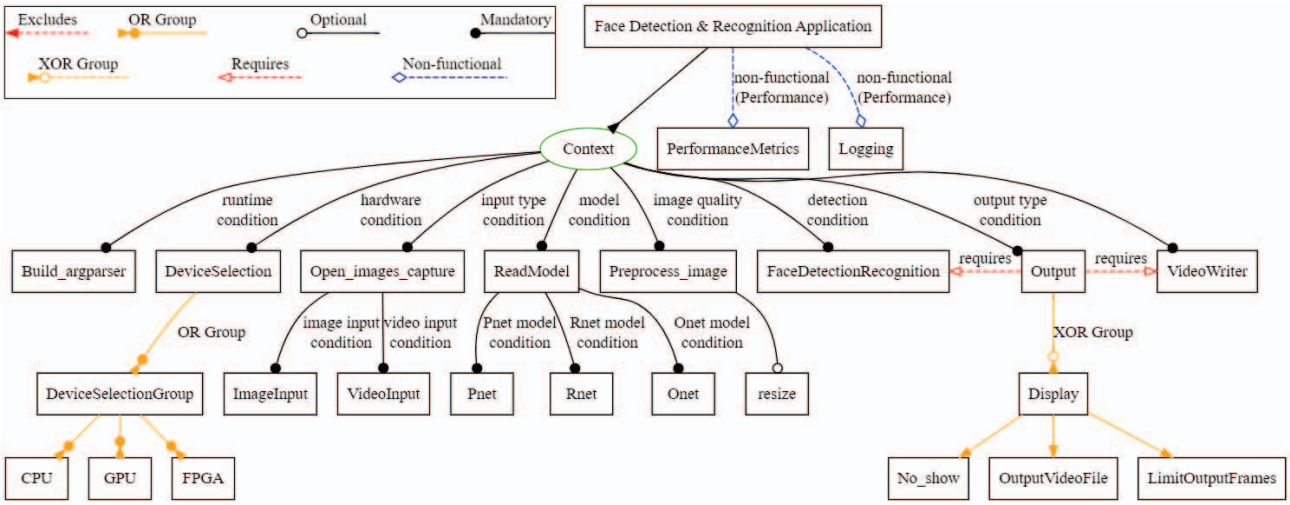

Figure 5: Adaptive Feature Model for Face Detection

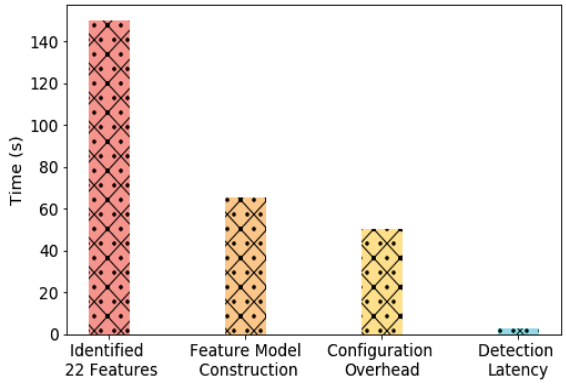

Figure 6: Execution Time Comparison for different Attributes using FeaMod

The relationship and constraints among the features were then established. For instance, *Output* was found to require *FaceDetectionRecognition* and *Video Writer* for it to function correctly. The *DeviceSelection* feature was determined to have an *OR* relationship among its sub-features: *CPU*, *GPU*, *FPGA*. The constructed adaptive feature model is shown in Figure 5. To ensure feature configuration, FeaMod checks for the existence of required features and the consistency of constraints among features, such as *OR*, *XOR*, and exclusionary relationships.

Table I delineates the breakdown of different feature model attributes of the face detection application examined in case study 1. The execution times for the various attributes of the FeaMod are illustrated in Figure 6. The most time-intensive task was the identification and extraction of features, taking approximately 150s — a reflection of the computational demands of discerning 22 features. Following this, the construction of the feature model necessitated 65s, and ensuring feature model configuration overhead consumed around 50s.

Detection latency remained minimal at a mere 2.5s. Despite the computational demands, the data manifests the potential for further optimization in performance, specifically in the realms of feature identification and model construction.

V. Related Works

A. Software Modularity and Feature Extraction

Our work stands on the foundational principles of software modularity and feature extraction, improving upon existing methodologies. Nadi *et al.* [11] made significant strides in understanding variability in large-scale systems, albeit without automated feature extraction, a gap our FeaMod framework fills. Paulius *et al.* [12] automated much of the feature modeling process using static code analysis and machine learning techniques, yet overlooked configuration considerations.

Distinctly, our approach leverages the BERT model to independently detect highly reused functions, differentiating it from the user-query-reliant method by Muller *et al.* [13]. While Jessie *et al.* [14] proposed an extended model to represent complex variability, and Ziadi *et al.* [15] focused on noise reduction and manual refinement, our approach automates the feature extraction process, enhancing precision and efficiency.

B. Adaptive Feature Modeling

The adaptive feature modeling landscape has seen various innovations. Alam *et al.* [16] adopted a concern-driven strategy that differs from our component functionality-centered perspective. Similarly, existing studies [17], [18] have explored context feature model-based adaptation logic architectures and MDRE for UML diagram generation from Java source code. Our approach distinctively utilizes the BERT model to facilitate a dynamic, adaptive, and valid feature modeling process that stands apart in accommodating evolving system requirements.

C. Threats to Validity

This section discusses the potential threats to the validity of our research and suggests areas for future expansion.

- *Generalizability of Results:* The results obtained from the case study on GitHub repositories may not be generalizable to other types of embedded software.
- *Scope of the Programming Languages:* While the current implementation of the FeaMod framework is tailored for Python, the underlying principles and techniques are applicable to any modular programming language with functional or procedural syntax structures. This adaptability suggests the potential for extending the framework to other languages commonly used in embedded systems.
- *Bias in Feature Extraction:* The reliance on CodeBERT for feature extraction might introduce biases based on the training data and model limitations.

VI. Conclusion

We introduced the FeaMod framework to facilitate code reuse and dynamic adaptability in embedded software development, leveraging advanced computational paradigms. It uniquely combines feature-based modularity with adaptive feature modeling, identifying features and their interrelationships from source code. Future work will aim to refine and broaden the framework's applications.

References

[1] Rui Tong, Quan Jiang, Zuqi Zou, Tao Hu, and Tianhao Li. Embedded system vehicle based on multi-sensor fusion. *IEEE Access*, 2023.

[2] Vidya Kamath and A Renuka. Deep learning based object detection for resource constrained devices-systematic review, future trends and challenges ahead. *Neurocomputing*, 2023.

[3] Jongmin Lee, Michael Stanley, Andreas Spanias, and Cihan Tepedelenlioglu. Integrating machine learning in embedded sensor systems for internet-of-things applications. In *2016 IEEE international symposium on signal processing and information technology (ISSPIT)*, pages 290–294. IEEE, 2016.

[4] Alberto Ballesteros, Manuel Barranco, Julián Proenza, Luís Almeida, Francisco Pozo, and Pere Palmer-Rodríguez. An infrastructure for enabling dynamic fault tolerance in highly-reliable adaptive distributed embedded systems based on switched ethernet. *Sensors*, 22(18):7099, 2022.

[5] Jane DA Sandim Eleutério, Breno BN de França, Cecilia MF Rubira, and Rogério de Lemos. Realising variability in dynamic software product lines. In *Software Engineering for Variability Intensive Systems*, pages 195–223. Auerbach Publications, 2019.

[6] Md Al Maruf, Akramul Azim, and Omar Alam. Facilitating reuse of functions in embedded software. *IEEE Access*, 10:88595–88605, 2022.

[7] Bruce Powel Douglass. *Design patterns for embedded systems in C: an embedded software engineering toolkit*. Elsevier, 2010.

[8] Code modularization. https://figshare.com/s/cf7c7e5ada6f0a7789787/t, 2023.

[9] Md Al Maruf and Akramul Azim. Automated features and requirements identification for improving cps software reuse using topic modeling. In *Proceedings of the ACM/IEEE 14th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2023)*, pages 262–263, 2023.

[10] OpenVINO Toolkit. Face detection mtcnn python* demo. https://github.com/openvinotoolkit/open_model_zoo/tree/master/demos/face_detection_mtcnn_demo, 2023. Accessed: 2023-07-12.

[11] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th international conference on software engineering*, pages 140–151, 2014.

[12] Robertas Damaševičius, Paulius Paškevičius, Eimutis Karčiauskas, and Romas Marcinkevičius. Automatic extraction of features and generation of feature models from java programs. *Information Technology and Control*, 41(4):376–384, 2012.

[13] Patrick Müller, Krishna Narasimhan, and Mira Mezini. Fex: Assisted identification of domain features from c programs. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 170–180. IEEE, 2021.

[14] Jessie Carbonnel, Marianne Huchard, and Clémentine Nebut. Towards the extraction of variability information to assist variability modelling of complex product lines. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 113–120, 2018.

[15] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. Feature identification from the source code of product variants. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 417–422. IEEE, 2012.

[16] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-oriented software design. In *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16*, pages 604–621. Springer, 2013.

[17] Martin Pfannemuller, Christian Krupitzer, Markus Weckesser, and Christian Becker. A dynamic software product line approach for adaptation planning in autonomic computing systems. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 247–254. IEEE, 2017.

[18] Umair Sabir, Farooque Azam, Sami Ul Haq, Muhammad Waseem Anwar, Wasi Haider Butt, and Anam Amjad. A model driven reverse engineering framework for generating high level uml models from java source code. *IEEE access*, 7:158931–158950, 2019.