IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 August 2023, accepted 1 September 2023, date of publication 11 September 2023,
date of current version 14 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3313598

RESEARCH ARTICLE

Bug Localization Model in Source Code Using Ontologies

ALISSON SOLITTO DA SILVA, ROGÉRIO EDUARDO GARCIA, AND LEONARDO CASTRO BOTEGA

Department of Mathematics and Computer Science, São Paulo State University (UNESP), Presidente Prudente, São Paulo 19060-900, Brazil

Corresponding authors: Alisson Solitto da Silva (alisson.solitto@unesp.br), Rogério Eduardo Garcia (rogerio.garcia@unesp.br), and Leonardo Castro Botega (leonardo.botega@unesp.br)

This study was supported in part by the Coordination for the Improvement of Higher Education Personnel (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) - Brazil (CAPES).

ABSTRACT The bug location process aims to identify source code artifacts associated with reported bugs. Manual bug location is burdensome for programmers who must reproduce and analyze the bug to identify the defective artifact and perform necessary maintenance. Bug locating techniques classify and identify project-specific source code artifacts, narrowing the search space. These techniques often use machine learning methods, such as textual similarity, classification algorithms, and grouping of source code files based on bug report data. This paper proposes a bug location model that leverages semantic architectural knowledge through ontologies to infer new knowledge and retrieve information from bug reports. The model's performance is evaluated on six relevant open-source projects in **C Sharp** (AutoMapper, MsBuild, EfCore, AspNetCore, MQTTnet, and NLog). Experiments utilize the evaluation metrics Top N Rank of Files (TNRF), Mean Reciprocal Rank (MRR), and Mean Average Precision (MAP). The results demonstrate the significant efficacy of the proposed model. The model contributes to relieving the manual burden on programmers and enhances bug localization accuracy and efficiency by integrating architectural semantic knowledge represented through ontologies with machine learning. The evaluation results indicate the potential of the proposed model for improving the bug-fixing process in software development.

INDEX TERMS Bug location, ontology, software engineering, software maintenance.

I. INTRODUCTION

Software development encompasses a range of methods, techniques, and tools employed throughout the software life cycle. Given the complexity and application domain, software systems are constantly evolving. Continuous improvement, corrective maintenance to address bugs in artifacts, and the need to refactor source code are integral parts of the software life cycle.

Despite adopting Software Engineering practices, bugs may still arise in software systems. Bugs can manifest themselves at various stages, from the initial software conception phase, requirements specification, diagrams, or use cases, to the deployed system [1].

Bug detection and localization are two critical phases in the software development life-cycle that substantially contribute to the final product's quality and robustness. Although closely related, these phases serve distinct functions and employ different techniques and tools.

Bug detection is the initial process of identifying errors, anomalies, or inconsistencies that may lead to incorrect or unexpected behavior in the software. This stage involves identifying syntax errors, logical mistakes, or coding standard violations through automated testing, static code analysis, or manual code review. The goal is to uncover issues during development, preventing errors from permeating the released software.

Following bug detection, the subsequent step is bug localization. This phase is focused on identifying the specific location or locations in the source code where the error occurred. Bug localization can be achieved through stack

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

tracing or algorithms correlating error reports with code sections. This process often requires manual intervention, as developers must decipher bug reports and pinpoint the source code entities requiring modification [2].

Once the source code artifact housing the bug is identified, a programmer performs the necessary maintenance [3]. The primary challenge lies in recognizing the source code artifacts associated with the defect report, often written in natural language.

There are noteworthy studies in the field of bug detection, such as [4], [5], and [6], but the focus of these works differs from what we address in this article. It is crucial to emphasize that bug localization and bug detection are distinct concepts and tackle separate issues. While bug detection concentrates on identifying the error, bug localization focuses on pinpointing the source code artifacts that originated the reported bug.

Bug Tracking Systems (BTS) provides tools to support corrective maintenance activities, enabling the reporting of inconsistencies discovered throughout the software life cycle, even during its use [7]. BTS allows for storing and managing bug reports, which describe situations where the software behaves unexpectedly. Bug reports are crucial in bug triage, as they contain valuable information for Software Engineering [8].

The bug-locating process aims to identify candidate source code artifacts related to the reported bug. This manual task is costly for programmers responsible for identifying the artifacts and performing the necessary maintenance. Various methodologies have been adopted in bug location processes, utilizing machine learning techniques, textual similarity, classification algorithms, and grouping of source code files based on data extracted from bug reports [9], [10], [11], [12], [13], [14].

There are two primary state-of-the-art approaches to bug location processes [12]. The first approach is based on similarity, where bug reports and source code files are classified, resulting in a set of source code files related to the bug report based on their level of similarity. The second approach leverages machine learning techniques and utilizes a set of training artifacts, which consist of historical bug reports and their respective faulty source code files. This approach employs algorithms to classify and group documents based on textual similarity, ultimately generating a scoring function to determine candidate source code files for correction.

Existing bug localization methodologies predominantly focus on the static analysis of source code files, often overlooking the architectural and semantic structure of the software. These methodologies employ Natural Language Processing (NLP) to establish semantic similarities between defect reports and source code artifacts. However, merely identifying semantic similarities in this context falls short; understanding the software's semantic structure (architecture) is imperative.

Frequently, a source code artifact may exhibit high similarity to a defect report, but that individual artifact might not address the issue adequately. The presumed source code artifact could inherit from an abstract class or implement an interface. Such interfaces or abstract classes may, in turn, have other inheritances that are overlooked when employing similarity or classification techniques. Relying solely on textual semantic similarity is insufficient to interpret the architectural structure of the source code.

Static methods that treat bug reports as mere collections of words often fail to grasp the architectural context of the source code artifact and to harness the rich semantic information embedded within bug reports. Domain knowledge and the application of ontologies can significantly enhance the bug localization process. Ontologies provide an integration and interoperability layer, unifying representations across domains within a semantic model [15].

When considering code semantics, NLP and machine learning techniques can be harnessed to discern patterns, intentions, and behaviors within the code. For instance, one might identify analogous code snippets, anticipate defects, or comprehend a module's functionality based on its comments and structure.

Within the source code context, an ontology can represent entities within a software system (such as classes, methods, variables, etc.) and their relationships (like inheritance, composition, method calls, etc.). That aids in understanding the system's design, architecture, and behavior.

While NLP is typically deployed on vast textual datasets for information extraction or language comprehension, ontologies cater to modeling specific domains. In these scenarios, ontologies proffer a formal and explicit structure to portray semantics and are more apt for applications where structure and relationships are paramount and necessitate clear definitions.

Although NLP boasts robust techniques for language interpretation, ontologies emerge as a superior tool for modeling source code's architecture and semantics. With its formal structure and rich semantics, the ontological representation offers a firm foundation for effective software system comprehension, maintenance, and evolution.

The formal nature of ontologies ensures clarity and precision in portraying a system's architecture. They also facilitate tracking dependencies and relationships within the code, proving beneficial for tasks like maintenance and refactoring. Additionally, ontologies promote a shared language suitable for integrating various systems or components, and by explicitly modeling semantics, they pave the way for deep comprehension and precise code interpretation, consequently reducing ambiguities.

Domain knowledge and the application of ontologies can contribute to advancing the bug location process. Ontologies provide a layer of integration and interoperability, unifying the representation of different domains within a semantic model [15].

Despite the potential benefits, the literature reveals only a limited number of models that employ ontologies in the

bug location process. Some of the presented models lack a comprehensive domain ontology. Papers by Kiefer et al. [16], and Tran and Le [17] utilize ontologies to represent the domain superficially. In other words, these ontologies only provide a taxonomy of some aspects within the domain without incorporating expressive rules and relationships.

Using ontologies in the bug location process contributes to identifying source code files that may not be explicitly related to the bug report. By employing ontologies, the semantic expressiveness of the source code is enhanced, enabling the inference of new knowledge.

This paper introduces a bug localization model that leverages ontologies to formally represent the domain knowledge of the object-oriented programming paradigm in a semantic structure. We aim to enhance the inference of new knowledge and retrieve information from bug reports by utilizing the architectural representation of source code artifacts within the ontology.

This paper proposes a bug localization model that utilizes ontologies to formally represent the domain knowledge associated with the object-oriented programming paradigm within a semantic framework. Furthermore, most research in this context predominantly focuses on projects coded in Java for bug extraction and localization. A review of related work reveals a noticeable tool gap catering to languages other than Java. Our proposal addresses this void and aims to advance these tools by incorporating another high-level language, C Sharp (C#), underscoring the presence of significant open-source projects within this community.

We aim to enhance the inference of new insights and the retrieval of information from bug reports by leveraging the architectural representation of source code artifacts within the ontology.

The paper is organized as follows: Section II provides the theoretical foundation of the concepts necessary for developing this work. Related works and discussions about this proposal are presented in Section III. Section IV outlines the research proposal and the methodology employed for the project. The experiments conducted and the results obtained with this approach are presented in Section V. Finally, Section VI concludes the paper by summarizing the findings and discussing future work.

II. BACKGROUND

This section presents the background relevant to the development of this work, providing the theoretical basis of the relationship between using ontologies and the Software Engineering area.

A. ONTOLOGIES AND SOFTWARE MAINTENANCE

The design and use of ontologies are central technologies for creating applications that support and manipulate intelligent information. Ontologies also facilitate solving problems related to technological applications that employ databases as a formal representation of knowledge. Using ontologies to represent domain knowledge can be applied throughout the Software Engineering life cycle [18]. Reference [19] complements this perspective by explaining that computational ontologies serve as a mechanism for formally modeling the structure of a system, capturing the relevant entities and relationships within a given domain.

Difficulties arise when knowledge is not formally represented using ontologies. For instance, the fundamental premises in databases are implicit, preventing the reuse and sharing of embodied knowledge. There is a lack of a generic model that allows the construction of databases by reusing an existing knowledge model. Lastly, no technology facilitates the quick extension of databases, enabling the combination of knowledge from complementary domains with the application domain [20].

During the software maintenance process, different types of information are listed without an explicit connection [21]. When software detects a defect, it is reported in a bug tracking system (BTS), initiating its life cycle until it is resolved. Throughout this cycle, the defect undergoes an exhaustive process of discussion and analysis. Developers need to understand the source code areas affected by the defect. This interaction process typically involves searching for content in forums, software artifacts, and system specifications to understand the evolution history of a specific component and resolve the reported occurrence. In this scenario, ontologies provide a layer of integration and interoperability for data from different sources, unified in a semantic model. In addition to giving semantic expressiveness to the data, ontologies allow for the inference of new knowledge not explicitly described in the reported defect report [15].

Understanding the application domain, technologies, testing procedures, and requirements is necessary when developing software. However, much of this knowledge is not formally documented, making it challenging to comprehend the system being maintained [22] fully. Ontologies can provide several benefits in inferring new knowledge and efficiently retrieving information, facilitating the assessment of expertise relevant to the application context.

Using ontologies in Software Engineering holds the potential for advancing software construction processes. The utilization of ontologies in Software Engineering can be categorized into two scenarios. The first scenario involves using ontologies at runtime and development time. The second scenario analyzes the domain problem the software aims to solve, classifying it into infrastructure and software aspects [15].

- **Ontology-driven development (ODD)**: This approach employs ontologies at development time to describe the domain problem. The primary example is ontological languages such as RDF (Resource Description Framework) and OWL (Web Ontology Language), which enable automatic validation and verification, reducing language ambiguity. Tools based on UML can be extended to support the creation of domain vocabulary and ontologies.

- **Ontology-enabled Development (OED)**: This category is also used at development time but focuses on supporting tasks performed by software developers, such as component research, component reuse, and software maintenance.
- **Ontology-based architectures (OBA)**: OBA-based architectures utilize ontologies as a primary artifact, constituting the application's core logic and business rules.

  Ontology-enabled architectures (OEA): In this architectural scenario, ontologies are applied to support the infrastructure of a software system by adding a semantic layer over the existing syntactic layer. That is: it enables semantic information retrieval and the description of services based on a formal semantic vocabulary.

Incorporating ontologies into the Software Engineering field makes enhancing various aspects of software development and maintenance processes possible. Using ontologies facilitates the formal representation of knowledge, enables data integration from diverse sources, and supports the inference of new knowledge. Furthermore, ontologies provide a semantic layer that enhances information retrieval and assists in understanding and managing complex software systems.

B. RECOGNITION OF NAMED ENTITIES

The process of identifying and classifying proper names found in a sentence is called Named Entity Recognition (NER). NER involves determining the entities (proper names) in a text and categorizing them based on their type. The most common types identified in the NER process are person, place, and organization. Additionally, NER can be extended to identify other types of entities, such as dates, temporal data, numerical expressions, and domain-specific entities [23].

NER plays a crucial role in understanding natural language. Unlike grammatical class markup, NER assigns labels to text entities using information stored in structured knowledge sources. Machine learning techniques can be employed in this process, where models are created and trained to identify and categorize entities based on the specific domain of interest. Alternatively, heuristics in regular expressions can be used [24].

According to [25], Named Entity Recognition can be broadly divided into two categories:

- Generic: In this scenario, entities corresponding to the names of people, organizations, places, values, dates, and emails are recognized.
- Specific: Specific recognition identifies entities belonging to a particular domain, such as protein names and car brands.

There are three primary methodologies for entity recognition in these scenarios, each with varying accuracy based on the recognition objective within a generic or specific domain.

According to [26] and [25], the following approaches are commonly used for named entity recognition:

1) Rule-based recognition: This approach relies on a set of rules and patterns manually created by domain experts. It leverages syntactic, linguistic, and domain-specific knowledge.
2) Machine learning approaches: Machine learning-based approaches do not require dictionaries or rule sets to annotate entities in a text. These approaches primarily utilize models such as Support Vector Machine (SVM), Hidden Markov Model (HMM), and Conditional Random Fields (CRF) and can employ supervised and unsupervised learning techniques.
3) Dictionary of entities: This approach involves maintaining a list of named entities corresponding to the specific domain. It requires a comprehensive understanding of the domain to ensure all possible entities are included in the dictionary.

III. RELATED WORKS

Considering the proposal of this paper, which suggests a bug location model utilizing semantic knowledge from the source code domain and employing entity recognition techniques to retrieve information from bug reports, relevant works that contribute to the development of the model have been analyzed and proposed.

In this section, we present papers related to the bug location process and the utilization of ontologies to model the source code. Furthermore, we compare related works and identify existing gaps.

The related works are categorized based on their primary subjects: software maintenance with ontology support, generation of source code ontologies, and the bug location process.

A. SOFTWARE MAINTENANCE WITH THE SUPPORT OF ONTOLOGIES

In the work of Witte et al. [27], some of the problems related to software maintenance are addressed: identifying security vulnerabilities in the source code, establishing a connection between artifacts by integrating information from documentation and source code and understanding the architectural structure of the project by identifying key components and properties. Semantic Web technologies provide a unified representation for exploring, querying, and understanding related artifacts. The authors present a formal ontological representation of the source code and documentation artifacts and an automatic population of these two ontologies through source code analysis and text mining. The software ontology comprises two ontologies representing the source code and software documents. The source code ontology formally specifies the key concepts of the object-oriented programming paradigm using the Java language. The ontology population is based on JDT, a source code analyzer provided by Eclipse. The semantic infrastructure comprises Semantic Web technologies (RDF/OWL) and Racer (used as an inference engine).

Tran and Le [17] present a defect search system that utilizes ontologies to annotate defect report data semantically. This work aims to explore defect reports available in a peer-to-peer (P2P) network, discover similar defects in this environment,

TABLE 1. Comparison of the main characteristics of related works.

Related works	Bug location	Use of ontologies	Semantic modeling of source code
Witte *et al*. [27]		X	X
Kiefer *et al*. [16]		X	X
Tran and Le [17]		X	
EkramiFard and Kahani [28]		X	X
Atzeni and Atzori [29]		X	X
Aguiar *et al*. [30]		X	
Gharibi *et al*. [9]	X		
Swe and Oo [10]	X		

and unify various defect reports from different systems into a single database. The authors employ Semantic Web technologies and studies in distributed systems to perform searches in P2P networks using SPARQL queries. For defect classification, they consider the properties of defect reports that can be directly extracted for package dependency analysis, classification, and identification of components related to the reported defect. The defect classification takes into account not only the properties already described in the defect report but also keywords or groups of keywords using the term frequency (TF) method that measures the occurrence frequency of terms in the document and the inverse document frequency (IDF) that measures the importance of a term.

B. SOURCE CODE ONTOLOGY GENERATION

In their work, EkramiFard and Kahani [28] aim to utilize Semantic Web technologies for detecting security vulnerabilities in source code by converting security flaw patterns into semantic queries using SPARQL. Their semantic generator processes solutions written in the Java language, consisting of an analyzer that retrieves the source code and generates a syntax tree to produce RDF triples, along with an information retrieval module (SPARQL).

Atzeni and Atzori [29] present an ontology for representing source code in the Java language. The ontology generation process involves analyzing and downloading all project dependencies, generating a syntax tree of the source code using the Spoon library, and serializing the code into RDF triples.

Ganapathy and Sagayara [31] focus on extracting metadata from source code to facilitate component reuse in other software projects. The authors propose a framework that extracts metadata from Java programs using the QDox library and stores the metadata in an ontology using the Apache Jena Fuseki ontology repository [32].

In a similar vein to work by Atzeni and Atzori [29], Aguiar et al. [30] aimed to develop a reference ontology for representing the object-oriented programming (OOP) paradigm in the context of polyglot programming. In a more recent work [33], the authors present a semantic generator for projects written in Java and Python, validating the ontology instance with code smell detection.

C. BUG LOCATION PROCESS

Gharibi et al. [9] propose a defect localization approach considering various properties extracted from the defect report and source code files, along with the relationships of the altered artifacts based on the defect report history. In this project, the authors consider using information retrieval techniques and text categorization to improve the efficiency of the defect localization process. The analysis and pre-processing of defect reports involve five steps: token matching, VSM similarity, stack tracing, semantic similarity, and corrected defect report history. Each step generates a score for the source code artifact. Finally, these scores are combined to obtain a final ranking, ordering the relevant source code files concerning each reported defect. After processing the extracted properties from the defect report and the modification history of the source code artifacts, a score is assigned to each identified artifact, and a final classification is performed. The final classification orders the relevant source code files concerning each reported defect.

Swe and Oo [10] present a defect localization model that differs from approaches that use source code artifacts as units, which can often introduce a lot of noise in the scoring classification when the file has many lines of code. Thus, the proposed model considers using structural information from the defect report to analyze its similarity with source code artifacts and stack traces. The ranking calculation assumes the similarity of each extracted property from the source code (class name, method, and variables), assigning scores using cosine similarity. Defect reports are also scored based on their similarity, and finally, the stack trace score is assigned. The classification of potential candidate artifacts for modification is performed by combining all three scores.

D. ANALYSIS OF WORK RELATED TO THE PROPOSAL

The main characteristics of the related works are listed in Table 1. In most projects, ontologies are used for the semantic representation of artifacts and the semantic modeling of the source code. Despite most related works benefiting

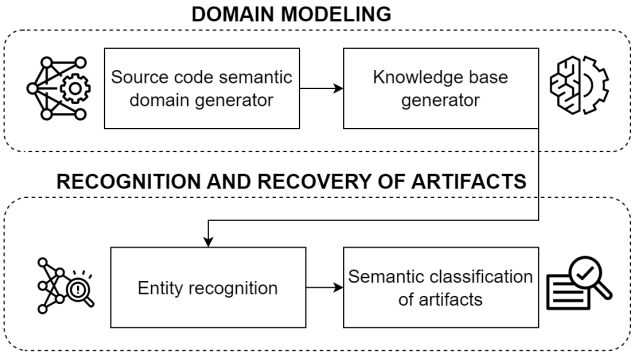


FIGURE 1. Model architecture for bug location.

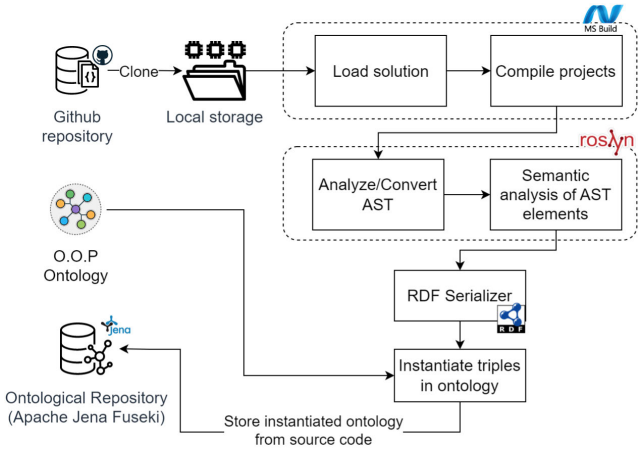


FIGURE 2. Source code semantic domain generator module.

from ontologies, the defect localization processes do not take advantage of semantic modeling.

In this scenario, the related works have contributed to identifying research gaps in defect localization models and techniques from other works that, when used together, can benefit these models.

IV. APPROACH

This section introduces a model for bug localization using the semantic domain knowledge of the source code. The aim is to automatically identify source code files that are candidates for modification based on the reported bug.

Unlike bug detection techniques, which aim to identify errors, anomalies, or inconsistencies in the code that might lead to incorrect or unexpected software behavior, such as syntax errors, logical mistakes, or violations of recommended coding standards, this model focuses on the next stage.

Once a bug has been detected and reported (typically by users, testers, or automated systems), bug localization is the next step. This process involves pinpointing the specific location or locations of artifacts within the source code where the bug occurred.

This paper proposes a model for bug localization using semantic knowledge formalized in an ontology and supported by entity recognition. In the proposed approach, the semantic representation of the source code within an ontology is employed to consider the source code's architectural and semantic structure, not merely the semantics related to terms identified in the code. This concept holds extreme relevance for identifying artifacts related to the candidates associated with the reported bug.

Figure 1 illustrates the high-level architecture of the model proposed in this paper. The model aims to automate the bug localization process, leveraging semantic web tools and techniques that have not been extensively explored in this research area. The model is comprised of four modules:

This model aims to automate the bug localization process by leveraging Semantic Web tools and techniques that have yet to be extensively explored in this research area. The model consists of four modules:

A. OWL-SHARP: SOURCE CODE SEMANTIC GENERATOR

The source code semantic generator [34] module converts source code written in the C Sharp (C#) language, following the object-oriented programming paradigm. Its objective within the bug localization model is to utilize ontological modeling to consider the semantics of project entities such as classes, methods, variables, and projects. The entities are extracted from the source code using the Roslyn compiler libraries [35], the official C Sharp language compiler. The architecture of this module is presented in Figure 2.

The ontology used to represent the source code semantics of the object-oriented programming paradigm was obtained from work by Aguiar et al. [30]. This ontology represents the semantic entities of a project at compile time and can be applied to any source code project developed using the object-oriented programming paradigm.

For this project, some modifications were made to the ontology to detail the semantics and relationships of certain entities identified during development, which are specific to the C Sharp language. The following entities have been added:

- Solution: Represents a solution that contains multiple application programs.
- Program: Represents the program to which a physical source code file belongs.
- File: Represents the physical file where the source code of a specific application module is stored.
- Mutator_Method: Represents a method that provides an interface between an object's internal data and the external world. It allows access to private instance variables of an object, also known as class properties, in C Sharp.

The source code semantic domain generator begins by retrieving all the project files from the GitHub repository and storing them in a system directory. The generator receives the file path of the solution in the system directory. It loads its projects, ".cs" extension files, and the necessary dependencies using the MsBuild library [36]. It is possible to load one or more solutions, projects, or files for analysis in this step.

```
private void Validate(ValidationContext
    context)
{
    for each (var validator in _validators)
    {
        validator(context);
    }
}
```

Listing 1. Validate method from AutoMapper project.

Upon loading the project, the compilation process starts using the MsBuild library, which validates dependencies and compiles the solutions, projects, or files.

After the compilation, the source code's abstract syntax tree (AST) structure is obtained for analysis. The C Sharp Roslyn compiler features extract the AST from each ''.cs'' file.

The Figure 3 presents the AST generated for the ''Validate'' method declaration, shown in code snippet 1 of the AutoMapper project. In Figure 3, the blue elements represent the syntax construction nodes, such as declarations, operators, expressions, etc. The green elements are the tokens that include identifiers, keywords, and special characters. The white or gray elements represent additional information about the node, such as spaces, end-of-line characters, line breaks, etc.

Next, the generator traverses all the AST elements to extract semantic information about objects, such as namespaces, entity names, types, accessibility levels, and modifiers. The Roslyn compiler libraries, specifically the Semantic API, are used for this semantic analysis. The Semantic API allows examination of the AST structure and retrieval of information such as variable types, considering dependencies on assemblies, imports, or namespaces.

The RDF serialization process begins after the semantic analysis of the AST elements. This process aims to convert the extracted relationships from the source code into triples (Resource, Predicate, Object) in RDF format, following the rules described in the ontology. The resource and object represent source code entities, while the predicate represents the relationship defined in the ontological model.

The triples are instantiated in the ontology after serializing the source code into RDF triples. This step involves persisting the generated RDF triples from the previous step, following the rules defined in the ontology. Finally, the semantic domain generator of the source code stores the instantiated ontology in the Apache Jena Fuseki ontology repository [32], enabling querying in the subsequent stages of the model. Figure 4 presents a portion of the instance of the method declaration from the code snippet 1 in the AutoMapper project in the ontology.

B. KNOWLEDGE BASE GENERATOR

The knowledge base generator module aims to create a knowledge base comprising domain concepts extracted from the source code ontology and their corresponding patterns. Figure 5 presents the architecture of this module.

During the domain concept extraction phase, the source code ontology generated in the semantic domain generator module is loaded. The domain concepts are identified as classes, methods, variables, and attributes. After retrieving the terms, the concepts are decomposed according to the C Sharp language's common standards and coding conventions [37].

The widely used naming standards for entities include *PascalCase*, *CamelCase*, and *snake_case*. The decomposition process identifies the writing convention of the concept and performs the term decomposition accordingly. For example, the concept ''MemberAccessQueryMapperVisitor'' of class type extracted from the AutoMapper project is decomposed into the pattern ''Member Access Query Mapper Visitor.'' The knowledge base is not limited to concept decomposition; the original and decomposed terms are combined after decomposition.

Following the decomposition and combination processes for ontology concepts, the lemmatization process is applied to each decomposed and combined token from the previous step. This step groups different inflected forms of a word and transforms them into their root form. For instance, ''running'' and ''runs'' are transformed into ''run,'' their root form.

After completing the decomposition, combination, and lemmatization steps, the domain concepts extracted from the source code ontology and their respective patterns are persisted in a knowledge base to support the next entity recognition step. The *json* code in Listing 2 presents terms for the concept ''MemberAccessQueryMapperVisitor'' after processing through all the stages of this module.

C. ENTITY RECOGNITION

The entity recognition process begins by loading bugs for analysis and concept labeling. Bugs are extracted by categorizing issues in the project repository on the GitHub platform. Figure 6 presents the architecture of this module.

Bug reports are obtained by retrieving all pull requests and identifying issues categorized as bugs linked to the repository. The text of the pull request record is then analyzed to determine which issue it addresses. This analysis creates a dataset that includes the pull request data, the issues labeled as bugs related to the pull request, and the files modified to fix the bug.

When retrieving issues labeled as ''bugs'' from the repository, several steps related to natural language processing are performed to prepare the extracted data for entity recognition.

Certain steps are carried out during the analysis and pre-processing stage before the entity recognition process begins. The pre-processing focuses on the title and description of the bug report. The pre-processing steps include the following:

- Sentence segmentation: The text is divided into sentences and segmented at the end of each sentence, indicated by periods.

```
{
  "_id": { "$oid": "627f94b626cb5a3c0966a398" },
  "Project": 0,
  "OntologyId": "http://ooc-o/#0ACC2582AA6C6146EF24EA7EACFF7FFD10...",
  "Type": "Member_Function",
  "Name": "MemberAccessQueryMapperVisitor",
  "FilePath": "...\\QueryableExtensions\\QueryMapperVisitor.cs",
  "Pattern": [
    "member_access", "memberquery", "membermapper", "membervisitor",
    "memberaccessquery", "memberaccessmapper", "memberaccessvisitor",
    "memberaccessquerymapper", "memberaccessqueryvisitor",
    "memberaccessquerymappervisitor", "accessquery", "accessmapper",
    "accessvisitor", "accessquerymapper", "accessqueryvisitor",
    "accessquerymappervisitor", "querymapper", "queryvisitor",
    "querymappervisitor", "mappervisitor", "member_access",
    "member_query", "member_mapper", "member_visitor",
    "member_access_query", "member_access_mapper",
    "member_access_visitor", "member_access_query_mapper",
    "member_access_query_visitor",
    "member_access_query_mapper_visitor", "access_query",
    "access_mapper", "access_visitor", "access_query_mapper",
    "access_query_visitor", "access_query_mapper_visitor",
    "query_mapper", "query_visitor", "query_mapper_visitor",
    "mapper_visitor"
  ]
}
```


Listing 2. **Concept generated in the knowledge base.**

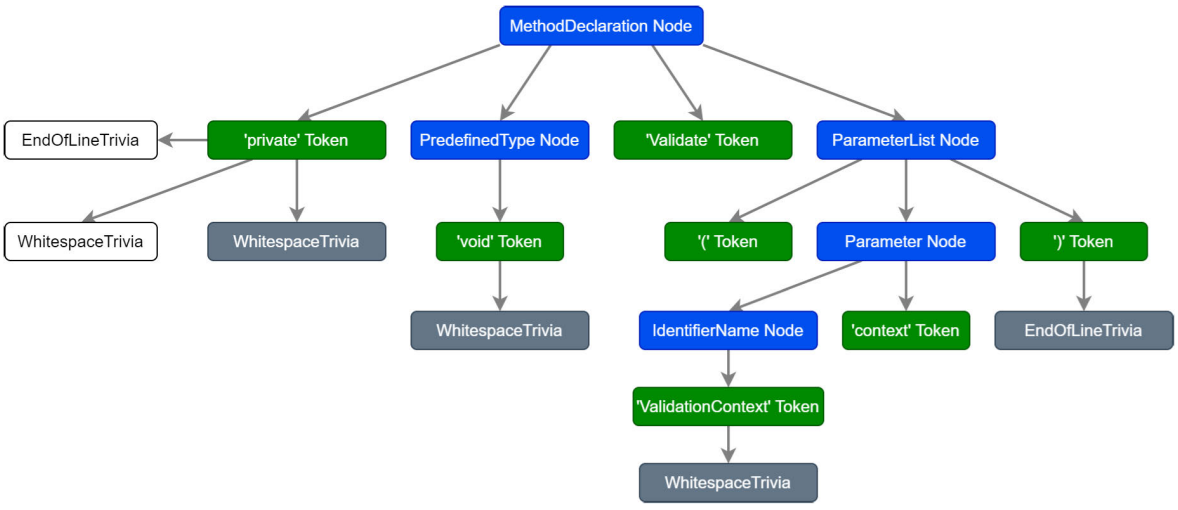


FIGURE 3. **AST related to the declaration of the *Validate* method in the AutoMapper project.**

- Word tokenization: This step involves separating the phrases into words or tokens. Typically, spaces are used as separators.
- Stemming: After tokenization, the terms are normalized by reducing words to their basic or root form. Stemming can produce root words that have no meaning.
- Lemmatization: This step groups different inflected forms of words, known as lemmas, together. The main difference between stemming and lemmatization is that lemmatization produces root words with semantic meaning.
- Identification of stop words: Irrelevant words, characters, and empty words are removed to obtain the tokens and their derivations.

The entity recognition process begins once the title and description of the bug reports have been pre-processed. Entity recognition involves labeling the concepts extracted from the bug report based on the knowledge base generated from the domain concepts extracted from the source code ontology.

In the bug location model proposed in this work, the domain knowledge of the source code and its evolution are controlled. As a result, the domain entities are fully

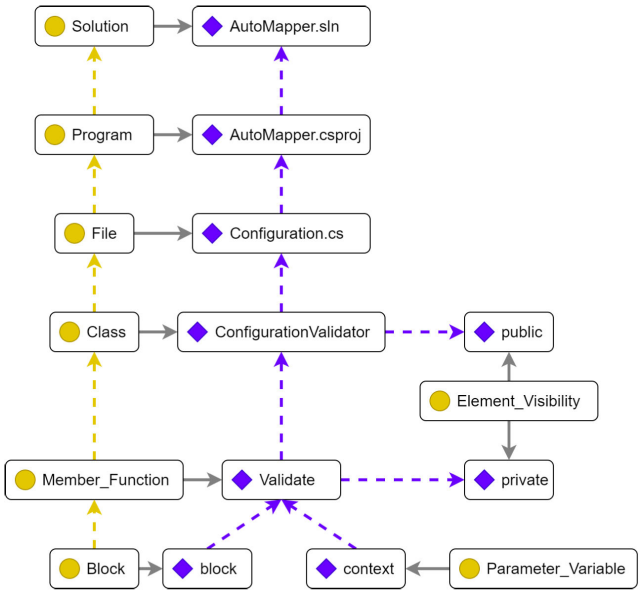


FIGURE 4. Instance in the ontology related to the declaration of the method "Validate" in the AutoMapper project.

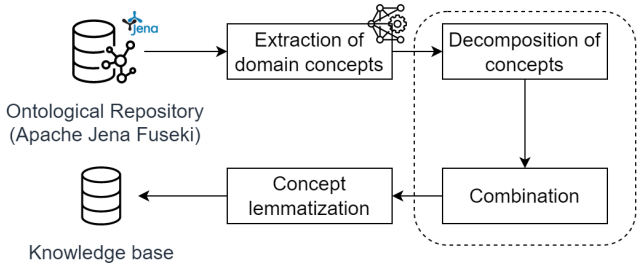


FIGURE 5. Knowledge base generator module.

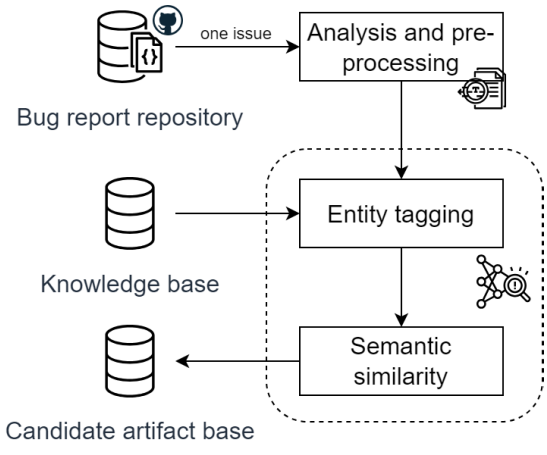


FIGURE 6. Entity recognition module.

known. Using a domain-controlled model makes employing a knowledge base and a rule-based matching mechanism for entity recognition possible, eliminating the need for advanced machine-learning techniques that rely on standard recognition. Furthermore, this approach requires a manageable amount of data for learning. In Figure 7, a snippet of the defect report is shown along with its corresponding annotation in Figure 8.

In Figure 8, the concepts identified by the entity recognition process are highlighted according to the domain knowledge present in the source code ontology of the project. The highlighted concepts are accompanied by their corresponding identifier in the ontology.

During this process, annotations of ambiguous entities may occur. The disambiguation of labeled concepts is performed in the semantic classification module. All entity annotations made in the defect report are considered in the semantic classification module of artifacts. Thus, the context in the other annotated entities is analyzed, and the term is disambiguated accordingly.

The semantic similarity component complements entity recognition by assessing the similarity of tokens with the patterns generated in the knowledge base of terms extracted from the source code ontology. Evaluating the similarity level is crucial for filtering out noise in the entity recognition process and identifying entities expressed in natural language within the bug report.

To measure the semantic similarity of labeled terms, the GloVe model incorporates words and vector representations [38]. The GloVe model employs unsupervised learning to generate word vectors by training on word-word co-occurrence matrices and using matrix factorization techniques.

For this step, the GloVe model is pre-trained with 300-dimensional vectors using the Common Crawl knowledge base [39]. As the vectors are pre-trained, the similarity is only checked for decomposed concept patterns that generate a token in natural language. Exact concepts in the domain do not require a similarity check.

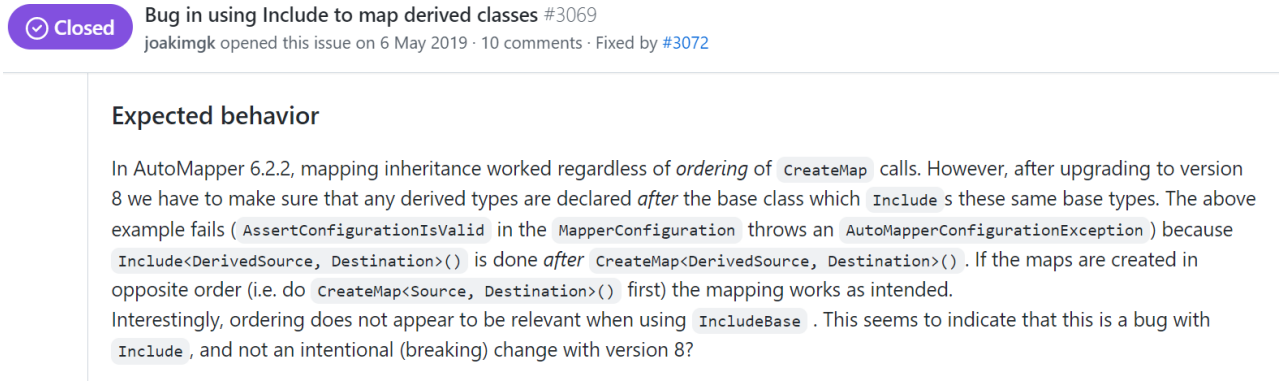
The semantic similarity assessment for tokens recognized in the bug report considers only tokens with a similarity level greater than or equal to 80% based on the knowledge base's patterns. This threshold is determined empirically by analyzing and evaluating the labeled concepts in the projects.

D. SEMANTIC CLASSIFICATION OF ARTIFACTS

Once the source code domain concepts have been located in the bug report, artifacts are identified and ranked. Before the final classification of artifacts, an ontological process is carried out to understand the relationships between the recognized concepts.

The first step in generating the ontology is to group the concepts based on the artifact they belong to. Once the artifacts are grouped, an ontology is generated for each artifact. This process considers the identified concepts, the level of semantic similarity assessment, and the semantic relationships defined in the source code's domain ontology.

Once the ontologies for each artifact have been generated, analytical methods are applied to classify each ontology based on its representativeness concerning the concepts



Closed Bug in using Include to map derived classes #3069

joakimgk opened this issue on 6 May 2019 · 10 comments · Fixed by #3072

Expected behavior

In AutoMapper 6.2.2, mapping inheritance worked regardless of *ordering* of `CreateMap` calls. However, after upgrading to version 8 we have to make sure that any derived types are declared *after* the base class which `Include`s these same base types. The above example fails (`AssertConfigurationIsValid` in the `MapperConfiguration` throws an `AutoMapperConfigurationException`) because `Include<DerivedSource, Destination>()` is done *after* `CreateMap<DerivedSource, Destination>()`. If the maps are created in opposite order (i.e. do `CreateMap<Source, Destination>()` first) the mapping works as intended.
Interestingly, ordering does not appear to be relevant when using `IncludeBase`. This seems to indicate that this is a bug with `Include`, and not an intentional (breaking) change with version 8?

FIGURE 7. Snippet of a defect report.

FIGURE 8. Example of entity recognition.

identified in the bug report. These metrics evaluate various aspects of representation within each ontology and determine their ranking based on the resulting values of all ontologies [40]. The architecture of this module is presented in Figure 9.

The first metric considered is the Class Match Measure (CMM). The CMM metric assesses the coverage of the ontology about the concepts identified in the bug report, searching for entities in the ontology that correspond to the identified concepts [40]. For each concept identified in the ontology, the semantic similarity level determines whether the ontology entity corresponds to an identical or partial concept. The data used in equations 1, 2, 3, 4, and 5 are the

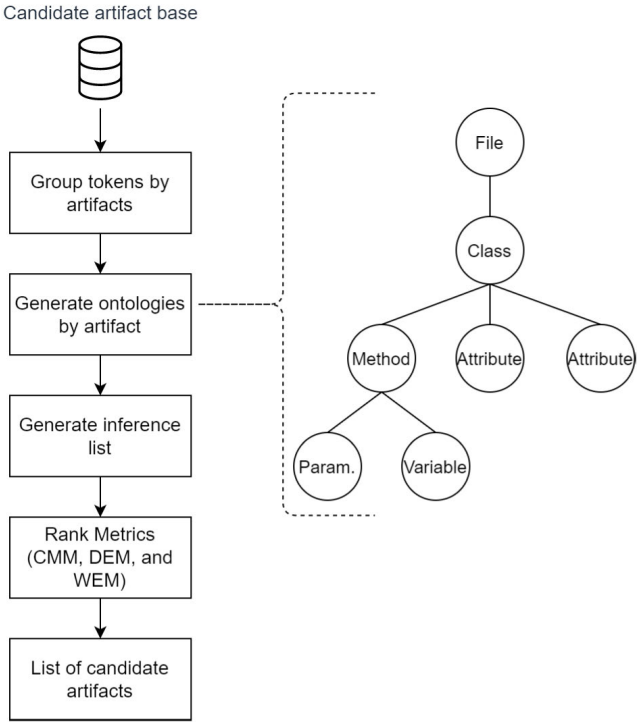


FIGURE 9. Semantic classification of artifacts module.

ontology (o), class (c), and ontology class set ($C[o]$).

$$E(o) = \sum_{c \in C[o]} SE(c) \tag{1}$$

$$SE(c) = \begin{cases} 1 : if similarity c = 1.0 \\ 0 : if similarity c < 1.0 \end{cases} \tag{2}$$

$$P(o) = \sum_{c \in C[o]} SP(c) \tag{3}$$

$$SP(c) = \begin{cases} 1 : if similarity 0.8 \geq c < 1.0 \\ 0 : if similarity c < 0.8 \end{cases} \tag{4}$$

$$CMM(o) = \alpha E(o) + \beta P(o) \tag{5}$$

For identical concepts, a semantic similarity level of 1 is considered. Concepts classified as partial have a similarity level between 0.8 and 0.9.

The values of α and β are assigned 0.6 and 0.4, respectively, to weigh the importance of exact or partial terms. These values are determined based on the study conducted by [40], favoring the precision of terms for calculating the CMM metric.

The next metric is the Density Measure (DEM) metric. The density metric assesses the level of detail in representing an identified concept. It also considers related entities, such as attributes and variables, when determining a class-type entity. The density measurement aims to evaluate the representative density and the level of detail of the ontology knowledge. This metric is adapted from the DEM metric for the current project's usage scenario [40]. The data used in equations 6 and 7 are the ontology (o), class (c), w_i weights, and $S = \{S_1, S_2, S_3, S_4\} = \{class[o], methods[o], attributes[o], variables[o]\}$.

$$dem(c) = \sum_{i=1}^{4} w_i |S_i| \tag{6}$$

$$DEM(o) = \frac{\sum_{i=1}^{n} dem(c)}{E(o) + P(o)} \tag{7}$$

The third metric aims to assess the weight of the ontology using the Weight Measure (WEM) without considering the relations with the previous metric (DEM). The WEM calculation is based on the DEM metric [40]. The data used in equations 8 and 9 are the ontology (o), class (c), w_i weights, and ontology entity types $S = \{S_1, S_2, S_3, S_4\} = \{class[o], methods[o], attributes[o], variables[o]\}$.

$$wem(c) = \sum_{i=1}^{4} w_i |S_i| \tag{8}$$

$$WEM(o) = \sum_{i=1}^{n} wem(c) \tag{9}$$

Finally, the total score is calculated after calculating the CMM, DEM, and WEM metrics for the ontologies generated from identifying concepts in the bug report and the grouping of artifacts. The score is computed by considering all the values from the previous metrics and their respective weights, which are defined empirically. The ranking of the ontologies is determined to classify the relevant artifacts based on the concepts identified in the bug report. The data used in equation 10 include the set of ontologies (O), the specific ontology (o), w_i weights, and the results obtained from the previous metrics $M = \{M[1], M[2], M[3]\} = \{CMM, DEM, WEM\}$.

$$Score(o \in O) = \sum_{i=1}^{3} w_i \frac{M[i]}{max_{1 \leq j \leq |O|} M[j]} \tag{10}$$

V. EXPERIMENTS AND RESULTS

This section evaluates the performance of the developed model, providing details about the dataset used for validation, the metrics employed to measure performance, and the results obtained through model execution. The evaluation metrics used are based on Information Retrieval, as described in the following subsections.

A. DATASET

To assess the proposed approach in this paper, we constructed a dataset comprising bug reports from various open-source projects available on GitHub. This bug report suite comprises over 650 bug reports from 6 different projects. Defect reports are obtained from the project repositories available on GitHub and include reports from various users. Figure 10 illustrates the architecture specifically designed to automate the batch data acquisition process for conducting the experiment.

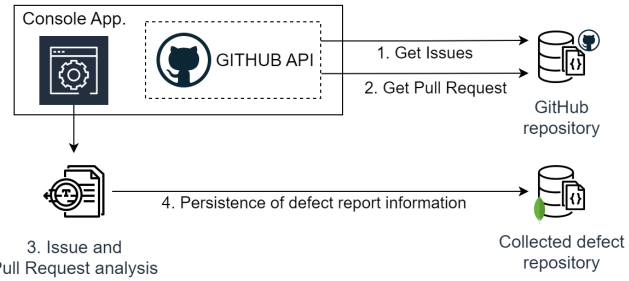


FIGURE 10. Solution for obtaining the dataset for the experiment.

The dataset encompasses bug reports extracted from diverse open-source projects by utilizing issue categorization within the project repositories on the GitHub platform. To obtain bug reports, we used the GitHub API to retrieve all pull requests and issues categorized as ''bugs'' linked to the respective repositories. After recovering the set of issues and pull requests, a heuristic was employed to analyze the text of the pull request records and identify the related issue that each pull request resolves. This process resulted in creating a dataset containing pull request data, issues labeled with bugs related to the pull requests, and the modified files that address the bugs.

The dataset does not include any filters and is thus representative of bug reports, encompassing various forms of user-inputted information, such as open-text descriptions, images, source code snippets, stack traces, etc.

Some pull requests and bug reports contain changes and fixes related to multiple bugs. Developers sometimes bundle fixes for several issues together to save time and effort. It is essential to clarify that the model is limited to locating and classifying the artifacts connected with the bug report's description. If the bug report is related to more than one artifact, these artifacts will be considered as originating from that specific bug report.

The following projects were considered in the dataset generation to evaluate the proposed approach. Only files with the ''.cs'' extension modified in the bug reports were considered. Reports containing test project files and files not present in the source code domain ontology were excluded from the dataset.

- AutoMapper: AutoMapper is a library designed to address object mapping challenges [41].
- MsBuild: The Microsoft Build Engine, commonly known as MSBuild, is a platform for building applications. It utilizes an XML schema for project files to control the software build process [36].
- EfCore: EF Core is an object database mapper (ORM) for .NET. It supports LINQ queries, change tracking, updates, and schema migrations [42].
- AspNetCore: ASP.NET Core is an open-source, cross-platform framework for developing modern cloud-based internet-connected applications, including web apps, IoT apps, and backends [43].
- MQTTnet: MQTTnet is a high-performance .NET library for MQTT-based communication. It provides an MQTT client and an MQTT server (broker) that supports MQTT protocol up to version 5 [44].
- NLog: NLog is a logging platform for .NET offering advanced log management and routing capabilities [45].

B. EVALUATION METRICS

Three evaluation metrics are considered for assessing the performance of the bug location model and ranking the list of candidate artifacts for each bug report. Higher values indicate better performance.

- Top N Rank of Files (TNRF): This metric evaluates the position of the candidate artifact in the bug report correction. Files are sorted based on their position in the returned list of artifacts, considering N positions (N = 1, 5, 10). For each bug report, if the top N artifacts include at least one file related to the bug fix, the artifact is considered to be located [9].
- Mean Reciprocal Rank (MRR): This metric measures the overall effectiveness of retrieving candidate artifacts from a set of bug reports. The metric calculates the reciprocal rank of the first relevant candidate artifact for the bug fix in each bug report [46].

$$MRR = \frac{1}{|BR|}\sum_{i=1}^{|BR|}\frac{1}{first_i} \tag{11}$$

- Mean Average Precision (MAP): The average precision metric is a calculation used in information retrieval to evaluate the performance of all candidate files relevant to the bug report. In the bug location process, more than one artifact may be relevant for fixing the bug, so this metric considers all candidate artifacts pertinent to the bug report [47].

$$MAP = \sum_{i=1}^{|BR|}\frac{AvgP(i)}{|BR|} \tag{12}$$

$$AvgP = \frac{\sum_{k\in SPrec@k}}{m} \tag{13}$$

$$Prec@k = \frac{relevant\_artifact\_in\_position\_k}{k} \tag{14}$$

C. EXPERIMENTAL RESULTS

In the initial evaluation, we also examined the metrics of the semantic domain generator module of the source code. Table 2 presents the metrics obtained when processing the project source code in the dataset used for the experiment. The ontologies were generated considering only files with the ''.cs'' extension and excluding files related to testing projects.

For all projects retrieved from GitHub, the source code semantic domain generator module successfully processed all project files and completed the processing of all classes, methods, attributes, parameters, inheritance, etc., extracted

TABLE 2. Source code semantic domain generator metrics.

Project	Time	File	Classes	Methods	Triples
AutoMapper	7.26	76	162	1,450	39,417
MsBuild	20.28	693	904	12,428	362,841
EfCore	31.35	1,704	1,950	19,576	602,378
AspNetCore	60.84	4,576	5,231	33,971	1,024,056
MQTTnet	2.7	238	243	2,081	55,020
NLog	7.92	409	501	5,627	157,029

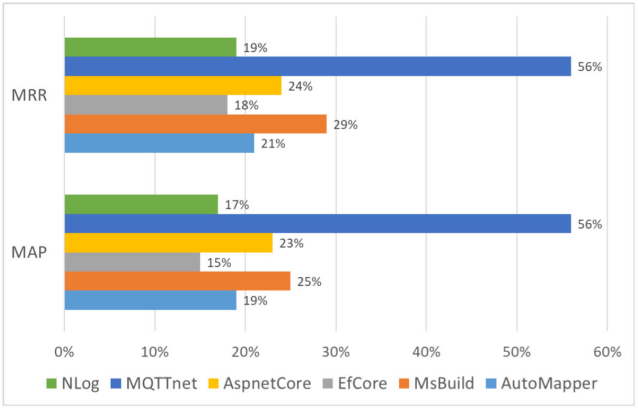


FIGURE 11. Performance TNRF.

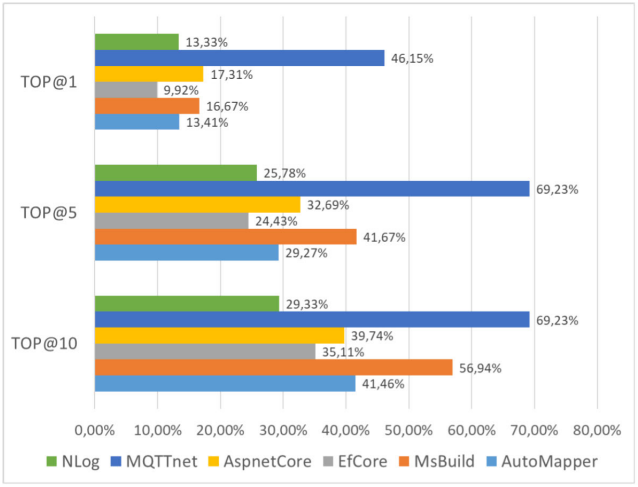


FIGURE 12. Performance MRR and MAP.

from the Abstract Syntax Tree (AST) and present in the source code artifacts.

The time metrics in Table 2 are expressed in seconds. The number of triples represents the semantic relationships in each project and includes all relationships present in the source code, not just classes and methods.

We evaluated the quality of the artifact list concerning the reported bugs. For this purpose, we used the same projects applied to the processing of the semantic domain generator module of the source code, as listed in Table 2.

In Table 3, the results of the metrics described in Section V-B are presented. For the TNRF metric, it can be observed that although the TOP@1 classification obtains an average percentage of approximately 20% for the six projects in the experiment, the TOP@5 and TOP@10 classifications achieved excellent results in identifying candidate artifacts related to the reported defects. It is worth noting that the defect reports used in the experiment were not filtered or classified for the investigation, meaning they were considered in their original form as described in real project scenarios. The defect reports on GitHub can contain images, code snippets, stack traces, or any other element, as the text on the GitHub platform is free-form. Figure 11 provides an overview of the results for each project for the TOP@1, TOP@5, and TOP@10 metrics. Figure 12 presents an overview of the MRR and MAP metrics results.

The MQTTnet project has 13 defect reports in its dataset for the experiment. Despite having a low number of defect reports, the project was included in the investigation to analyze the influence of the MRR and MAP metrics for the relevant files found in the first position.

The MRR and MAP metrics consider the position in ranking candidate artifacts in their formulas. Therefore, the results of MRR and MAP are also influenced by the TOP@1 TNRF metric. This relevance of the TOP@1 classification can be observed in the MQTTnet project, where the percentages of the TOP@5 and TOP@10 classifications are higher than the TOP@1. Still, the values of MRR and MAP are higher when compared to the other projects.

Despite the importance of the TOP@1 metric, some situations may need to be revised to analyze results when considering only the TOP@1 metric as a relevant evaluation for the experiment. For example, when a defect report fix involves changes in three different source code artifacts, one of the artifacts is found in the first position during the classification process. In contrast, the others are ranked near or beyond the TOP@10. In this case, the MRR and MAP metrics may show a higher result than in an example where the three modified source code artifacts are found in the 3rd, 4th, and 5th positions.

TABLE 3. Identification and classification performance of candidate artifacts.

Project	TOP@1	TOP@5	TOP@10	MRR	MAP
AutoMapper	11 (13.41%)	24 (29.27%)	34 (41.46%)	21%	19%
MsBuild	12 (16.67%)	30 (41.67%)	41 (56.94%)	29%	25%
EfCore	13 (9.92%)	32 (24.43%)	46 (35.11%)	18%	15%
AspnetCore	27 (17.31%)	51 (32.69%)	62 (39.74%)	24%	23%
MQTTnet	6 (46.15%)	9 (69.23%)	9 (69.23%)	56%	56%
NLog	30 (13.33%)	58 (25.78%)	66 (29.33%)	19%	17%

The MRR and MAP metrics are relevant. However, in the analysis of the experiments, we can see that the TOP@5 and TOP@10 metrics achieved significant results for the classification process of candidate artifacts for modification. The MsBuild and AspnetCore projects show substantial results in the TOP@5 and TOP@10 metrics. Both projects serve as bases for application development and have implementations that utilize advanced language features, making it challenging to identify candidate artifacts for defect reports.

D. LIMITATIONS

A limitation of this experiment is the need to compare the proposed technique and existing techniques for bug localization. This point is necessary to clarify the unique context of this research. The technique introduced in this paper is the first in the state-of-the-art to engage with open-source projects developed in C Sharp. In contrast, existing approaches predominantly focus on projects developed in Java.

Research works [28], [29], [31], [33] have showcased tools for the semantic generation of source code for projects coded solely in the Java programming language. As a result, the contribution has remained restricted, and using ontologies to support Software Engineering needs more tools for extracting metadata from source code in other high-level programming languages. The semantic source code domain generator presented in the bug localization methodology of this paper also contributes to the evolution of semantic code generation tools, opening a new horizon for projects coded in the C Sharp language.

This divergence in programming languages presents substantial challenges for a direct comparison. The underlying structure and architectural principles between C Sharp and Java and the particular projects utilized further complicate the matter. For these reasons, conducting a direct and meaningful comparison would necessitate overcoming significant barriers related to the alignment and normalization of the datasets and criteria between the two programming languages. That could lead to inconclusive or misleading results.

Moreover, the unique application of C Sharp in this research fills a gap in the existing literature, offering fresh insights and possibilities within the field of bug localization. The absence of a direct comparison with Java-based techniques should not detract from the contributions of this work but rather highlight the novelty and potential for further exploration and comparison in future studies.

VI. CONCLUDING REMARKS AND FUTURE WORK

The ontology plays a crucial role in the bug localization model proposed in this paper. Bug localization using ontologies facilitates interoperability, standardization, organization, and reuse of domain information extracted from the application source code.

Using an ontology to consider the architectural and semantic structure of the source code is important for capturing artifacts not identified in the defect report. These artifacts in projects can include class inheritance, interfaces, methods that modify a virtual implementation, and inherited property elements.

In addition to discovering artifacts not identified in the defect report, the semantic structure of the source code enables the disambiguation of terms by analyzing the context of other annotated entities.

Leveraging ontologies in the bug localization process makes it possible to perform entity recognition without relying on deep machine learning algorithms.

The development of this work is expected to contribute to the advancement of the defect localization process by employing an ontology to semantically represent the source code domain, thereby enabling the exploration of new approaches for the semantic representation and identification of concepts from the source code domain in defect reports. This model can benefit The corrective maintenance process as it automates the search for source code artifacts that are potential candidates for bug fixes by narrowing down the search space within the project's source code.

All research in this scenario uses projects coded in Java to extract and validate semantic code generators. The lack of tools that work with languages other than Java is evident in all the related works. Our proposal also contributes to the evolution of these tools by adding another high-level language (C Sharp), highlighting the existence of important open-source source code projects in this community.

While the lack of direct comparison with existing Java-based techniques might be perceived as a limitation, it is a deliberate choice rooted in the distinct nature of the research subject. The findings and contributions of this study stand as an independent advancement in the field, paving the way for subsequent investigations that leverage this foundation to craft more comprehensive comparative analyses.

For future work, the proposed bug localization model, which incorporates entity recognition and ontologies, can be

enhanced by incorporating additional data inputs for information retrieval. Including the analysis of similar bugs as a parameter in the model can help identify which artifacts were affected in the version history of the source code. This approach can improve the accuracy of candidate source code artifacts and their ranking positions.

Moreover, the support of ontologies in generating the semantic domain of the source code also opens avenues for research in areas such as code smells detection, design pattern localization, component reuse, and vulnerability identification. Additionally, it enables the inference of new knowledge from the source code domain.

REFERENCES

[1] M. Delamaro, M. Jino, and J. Maldonado, *Introduction to Software Testing (Introdução ao Teste de Software)*. Brasília, Brasil: Elsevier, 2013.

[2] S. Gujral, G. Sharma, S. Sharma, and Diksha, "Classifying bug severity using dictionary based approach," in *Proc. Int. Conf. Futuristic Trends Comput. Anal. Knowl. Manage. (ABLAZE)*, Feb. 2015, pp. 599–602.

[3] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Aug. 2009, pp. 111–120.

[4] J. Winkler, A. Agarwal, C. Tung, D. R. Ugalde, Y. J. Jung, and J. C. Davis, "A replication of 'deepbugs: A learning approach to name-based bug detection,'" in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, p. 1604.

[5] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2016, pp. 708–719.

[6] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *ACM Program. Lang.*, vol. 3, pp. 1–30, Oct. 2019.

[7] T. Zhang and B. Lee, "A bug rule based technique with feedback for classifying bug reports," in *Proc. IEEE 11th Int. Conf. Comput. Inf. Technol.*, Aug. 2011, pp. 336–343.

[8] M. F. Zibran, "On the effectiveness of labeled latent Dirichlet allocation in automatic bug-report categorization," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2016, pp. 713–715.

[9] R. Gharibi, A. H. Rasekh, M. H. Sadreddini, and S. M. Fakhrahmad, "Leveraging textual properties of bug reports to localize relevant source files," *Inf. Process. Manage.*, vol. 54, no. 6, pp. 1058–1076, Nov. 2018.

[10] K. E. E. Swe and H. M. Oo, "Bug localization approach using source code structure with different structure fields," in *Proc. IEEE 16th Int. Conf. Softw. Eng. Res., Manage. Appl. (SERA)*, Jun. 2018, pp. 159–164.

[11] D. Chen, B. Li, C. Zhou, and X. Zhu, "Automatically identifying bug entities and relations for bug analysis," in *Proc. IEEE 1st Int. Workshop Intell. Bug Fixing (IBF)*, Feb. 2019, pp. 39–43.

[12] P. Loyola, K. Gajananan, and F. Satoh, "Bug localization by learning to rank and represent bug inducing changes," in *Proc. 27th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2018, pp. 657–665.

[13] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension (ICPC)*, May 2017, pp. 218–229.

[14] M. M. Rahman and C. K. Roy, "Improving IR-based bug localization with context-aware query reformulation," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Oct. 2018, pp. 621–632.

[15] H.-J. Happel and S. Seedorf, "Applications of ontologies in software engineering," in *Proc. Workshop Sematic Web Enabled Softw. Eng. (SWESE) (ISWC)*, 2006, pp. 5–9.

[16] C. Kiefer, A. Bernstein, and J. Tappolet, "Analyzing software with iSPARQL," in *Proc. Int. Workshop Semantic Web Enabled So Ware Eng. (SWESE)*, 2007, pp. 1–15.

[17] H. M. Tran and S. T. Le, "Software bug ontology supporting semantic bug search on peer-to-peer networks," *New Gener. Comput.*, vol. 32, no. 2, pp. 145–162, Apr. 2014.

[18] M. Uschold and M. Gruninger, "Ontologies: Principles, methods and applications," *Knowl. Eng. Rev.*, vol. 11, no. 2, pp. 93–136, Jun. 1996.

[19] N. Guarino, D. Oberle, and S. Staab, "What is an ontology?" in *Handbook Ontologies*. Berlin, Germany: Springer, 2009, pp. 1–17.

[20] S. Isotani and I. I. Bittencourt, *Dados Abertos Conectados: Em busca da Web do Conhecimento*. Brasília, Brasil: Novatec Editora, 2015.

[21] R. d. A. Falbo and G. H. Travassos, "A integração de conhecimento em um amblente de desenvolvimento de software," in *Proc. II Congreso Argentino de Ciencias de la Computación*, 1996, pp. 328–329.

[22] D. Gasevic, N. Kaviani, and M. Milanović, "Ontologies and software engineering," in *Handbook on Ontologies*, S. Staab and R. Studer, Eds. Berlin, Germany: Springer, May 2009, pp. 593–615, doi: 10.1007/978-3-540-92673-3_27.

[23] V. Keselj, *Speech and Language Processing*, D. Jurafsky and J. H. Martin, Eds. Stanford, CA, USA: Stanford Univ. Univ. Colorado at Boulder, 2009.

[24] W. Khan, A. Daud, J. A. Nasir, and T. Amjad, "A survey on the state-of-the-art machine learning models in the context of NLP," *Kuwait J. Sci.*, vol. 43, no. 4, pp. 98–100, 2016.

[25] G. K. Palshikar, "Techniques for named entity recognition: A survey," in *Bioinformatics: Concepts, Methodologies, Tools, and Applications*. Hershey, PA, USA: IGI Global, 2013, pp. 400–426.

[26] H.-J. Song, B.-C. Jo, C.-Y. Park, J.-D. Kim, and Y.-S. Kim, "Comparison of named entity recognition methodologies in biomedical documents," *Biomed. Eng. OnLine*, vol. 17, no. S2, pp. 1–14, Nov. 2018.

[27] R. Witte, Y. Zhang, and J. Rilling, "Empowering software maintainers with semantic web technologies," in *Proc. Eur. Semantic Web Conf.* Cham, Switzerland: Springer, 2007, pp. 37–52.

[28] A. EkramiFard and M. Kahani, "Providing a source code security analysis model using semantic Web techniques," in *Proc. Int. Congr. Technol., Commun. Knowl. (ICTCK)*, Nov. 2015, pp. 33–37.

[29] M. Atzeni and M. Atzori, "CodeOntology: RDF-ization of source code," in *Proc. Int. Semantic Web Conf.* Cham, Switzerland: Springer, 2017, pp. 20–28.

[30] C. Z. d. Aguiar, R. d. Almeida Falbo, and V. E. S. Souza, "OOC-O: A reference ontology on object-oriented code," in *Proc. Int. Conf. Conceptual Modeling*. Cham, Switzerland: Springer, 2019, pp. 13–27.

[31] G. Ganapathy and S. Sagayaraj, "To generate the ontology from Java source code," *Int. J. Adv. Comput. Sci. Appl.*, vol. 2, no. 2, p. 146, 2011.

[32] A. Jena. (2021). *Apache Jena Fuseki*. Accessed: May 9, 2021. [Online]. Available: https://jena.apache.org/documentation/fuseki2/

[33] C. Z. D. Aguiar, F. Zanetti, and V. E. S. Souza, "Source code interoperability based on ontology," in *Proc. 17th Brazilian Symp. Inf. Syst.*, Jun. 2021, pp. 1–8.

[34] A. S. Da Silva, R. E. Garcia, and L. C. Botega, "OWL-sharp: Source code semantic generator," in *Proc. 18th Iberian Conf. Inf. Syst. Technol. (CISTI)*, Jun. 2023, pp. 1–6.

[35] Microsoft. (2021). *Roslyn*. Accessed: May 9, 2021. [Online]. Available: https://github.com/dotnet/roslyn

[36] Microsoft. (2022). *Msbuild*. Accessed: May 2, 2022. [Online]. Available: https://github.com/dotnet/msbuild

[37] Microsoft. (2022). *Csharp Coding Conventions*. Accessed: May 2, 2022. [Online]. Available: https://docs.microsoft.com/pt-br/dotnet/csharp/fundamentals/coding-style/coding-conventions#naming-conventions

[38] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1532–1543.

[39] GloVe. (2022). *Common Crawl*. Accessed: May 2, 2022. [Online]. Available: https://nlp.stanford.edu/projects/glove/

[40] H. Alani, C. Brewster, and N. Shadbolt, "Ranking ontologies with aktiverank," in *Proc. Int. Semantic Web Conf.* Cham, Switzerland: Springer, 2006, pp. 1–15.

[41] AutoMapper. (2022). *Automapper*. Accessed: May 2, 2022. [Online]. Available: https://github.com/AutoMapper/AutoMapper

[42] Microsoft. (2022). *Efcore*. Accessed: May 2, 2022. [Online]. Available: https://github.com/dotnet/efcore

[43] Microsoft. (2022). *Aspnetcore*. Accessed: May 2, 2022. https://github.com/dotnet/aspnetcore

[44] Microsoft. (2022). *Mqttnet*. Accessed: May 2, 2022. [Online]. Available: https://github.com/dotnet/MQTTnet

[45] NLog. (2022). *Nlog*. Accessed: May 2, 2022. [Online]. Available: https://github.com/NLog/NLog

[46] E. M. Voorhees, "The TREC question answering track," *Natural Lang. Eng.*, vol. 7, no. 4, pp. 361–378, Dec. 2001.

[47] C. D. Manning, P. Raghavan, and H. Schütze, "Introduction to information retrieval," *Natural Lang. Eng.*, vol. 16, no. 1, pp. 100–103, 2010.

ALISSON SOLITTO DA SILVA received the B.S. degree in computer science and the degree in information technology management from Eurípedes de Marília University Center (UNIVEM), in 2018 and 2019, respectively, and the M.S. degree in computer science from the Postgraduate Program, São Paulo State University ''Júlio de Mesquita Filho,'' in 2022.

In 2020, he was a Professor with the Computer Science Program and specialization courses with UNIVEM. He is currently a Software Architect, designing solutions for companies in the financial sector and leading development teams to adhere to best software development practices. His research interests include software architecture, design patterns, frameworks, software maintenance, and software engineering.

ROGÉRIO EDUARDO GARCIA received the B.S. degree in computer science and the M.S. and Ph.D. degrees in computer science and computational mathematics from the Institute of Mathematical and Computing Sciences, University of São Paulo (ICMC-USP), São Carlos Campus, São Paulo, in 1998 and 2006.

From 2012 to 2013, he was visiting The University of Alabama. He is currently an Associate Professor with the Department of Mathematical and Computer Science, Faculty of Science and Technology, São Paulo State University, Presidente Prudente Campus, where he was initiated as an Assistant Professor, in 2006. He is the author of 17 chapters of books and more than 100 articles. His research interests include program understanding, change impact analyses, software visualization, maintenance, and engineering.

LEONARDO CASTRO BOTEGA received the Ph.D. degree in computer science from the Computer Department, Federal University of São Carlos (UFSCar), São Carlos Campus, São Paulo, Brazil, in 2016. From 2017 to 2018, he was a Postdoctoral Fellow with the University of São Paulo (ICMC-USP), São Carlos Campus, São Paulo. He has been a Professor of computer science and information science, since 2008. He is currently an Advisor and a Researcher of the Graduate Programs on Computer Science and Information Sciences, São Paulo State University. His research interests include critical data science, semantics, and data-driven decision-making. He is the author of more than 100 articles on these subjects.