

ANTLR4-Driven Model-Driven Reverse Engineering: Bridging Source Language Parsing and Metamodel Instantiation

Mohamed Karim Khachouch ¹, Ayoub Korchi ¹, Mohammed Bekkali ², Younes Lakhrissi ³

¹ SIGER Laboratory FST of Fez, Sidi Mohamed Ben Abdellah University, MOROCCO

² Research Center STIS, M2CS, Department of Applied Mathematics and Informatics, ENSAM, Mohammed V University, Rabat, Morocco

³ SIGER Laboratory ENSA of Fez, Sidi Mohamed Ben Abdellah University, MOROCCO

mohamedkarim.khachouch@usmba.ac.ma

Abstract. Model-Driven Reverse Engineering (MDRE) is a crucial methodology in software engineering for analyzing and evolving legacy systems through the extraction of higher-level abstractions from lower-level artifacts. This study proposes a novel approach that integrates ANTLR4-generated visitor patterns with metamodeling to streamline the MDRE workflow. The ANTLR4 parser is leveraged to generate a visitor that traverses the syntax tree of the source system's language, enabling the instantiation of target metamodel elements, including the Knowledge Discovery Metamodel (KDM). The resulting model is serialized into an XMI file, facilitating subsequent engineering tasks. The proposed method addresses the challenges of parsing complex source languages, accurately instantiating metamodel elements, and generating faithful target models encapsulated in a standardized format. A case study involving a Java Gradle project demonstrates the approach's practical application and effectiveness. The contributions of this work include a systematic methodology for efficient source language parsing, model generation, and integration with established metamodels, as well as insights into the implications for future software development practices and the advancement of MDRE techniques

Keywords: MDRE, Model-Driven Reverse Engineering, Knowledge Discovery Metamodel, Grammar, ANTLR4-Powered.

1. Introduction

Model-Driven Reverse Engineering (MDRE) is an approach in software engineering that focuses on the systematic extraction of higher-level abstractions from lower-level artifacts, such as source code or databases, (Raibulet et al., 2017). This methodology is essential for the maintenance and evolution of legacy systems, as it allows for the creation of models that reflect the system's architecture and behavior, (Agt-Rickauer, 2020). MDRE is particularly valuable for understanding complex systems, as it aids in the identification of core components and their interrelations, providing a visual and conceptual framework that assists stakeholders in navigating the structure of existing software. The process serves as a foundation for reengineering efforts, facilitating the transformation of outdated systems into modern architectures while preserving essential functionalities, (Bézivin, 2005). In the context of rapidly evolving technology, MDRE is a critical methodology for ensuring the robustness, scalability, and adaptability of software systems, (Yun, 2013). It enables organizations to mitigate risks associated with system modifications, improve software quality, and ease the transition to new platforms or technologies. Consequently, MDRE contributes to the sustainability and efficiency of software development practices, (Stahl et al., 2006).

Despite significant advancements, existing MDRE techniques often face challenges, such as accurately extracting architectural intent and behavior, navigating varied technological stacks, and ensuring the fidelity of reverse-engineered models.

This study aims to address the challenges of accurately parsing complex source languages, instantiating metamodel elements, and generating faithful target models in the context of Model-Driven Reverse Engineering (MDRE). The proposed approach leverages ANTLR4-generated visitor patterns and integrates them with metamodeling techniques to streamline the MDRE workflow, improve the efficiency of model generation, and enhance the fidelity of the resulting models. Utilizing an ANTLR4-generated visitor based on the grammar of the source system's language, the method systematically traverses the syntax tree. Each visitor method instantiates the corresponding element of the target metamodel, constructing the destination model, which is then serialized into an XMI file. This integration of the visitor pattern with metamodeling streamlines the MDRE workflow and exemplifies the method's practical application through the generation of a Concrete Syntax Tree (CST) within a Java project. The strategy simplifies the MDRE process and improves its accuracy and efficiency, potentially setting a new standard for reverse engineering practices.

The Knowledge Discovery Metamodel (KDM) can be used to capture the essential aspects of the system being analyzed, which illustrates the practical application and versatility of our approach

The article's objective is to present a comprehensive MDRE approach using the ANTLR4 parser to generate a visitor for transforming source system languages into a target metamodel. This method addresses the challenges of modeling complex software systems and contributes to software engineering by enhancing the understanding and manipulation of legacy code. The main contributions include a systematic methodology for parsing source language grammar, instantiating metamodel elements, and generating a target model encapsulated into an XMI file. The process facilitates the reverse engineering of software systems and ensures the fidelity of the resulting models to the original system. A case study involving a Java Gradle project illustrates the method's versatility and effectiveness. The approach's potential to streamline MDRE processes and its implications for future software development practices are discussed, positioning this work as a significant advancement in reverse engineering.

2. Theoretical Foundations

2.1. Preliminaries

Model-Driven Reverse Engineering (MDRE) is a systematic methodology in software engineering that is essential for the analysis, comprehension, and reconstruction of the architectural framework of existing software systems, (Raibulet et al., 2017). This approach is critical for legacy systems, which

are often characterized by incomplete documentation and a series of incremental modifications that may conceal the systems' foundational design principles, (Belfrit Victor, 2013). MDRE addresses multiple challenges associated with the inherent complexity of software systems. These include the accurate extraction of architectural intent and behavior, (Sabir et al., 2019), the navigation of varied and potentially outdated technological stacks, (García-Borgoñón et al., 2023), and the verification that reverse-engineered models accurately represent the original system and serve as a robust basis for further development or migration efforts, (Bruneliere, 2018).

MDRE practitioners are required to bridge the gap between high-level conceptual models and low-level executable code, (Farmer & Gruba, 2006). This necessitates the use of sophisticated tools and methodologies that can interpret complex code structures and translate them into comprehensible and operational models. The interpretative aspect of this process requires a significant level of expertise, as engineers must determine the appropriate level of abstraction for system components and the most effective representation of these components within the model, (Chandrasegaran et al., 2013).

With the increasing scale and complexity of software systems, the importance of MDRE in ensuring the sustainability and adaptability of these systems is amplified. It represents a field of study and application within software engineering, dedicated to the ongoing development of more advanced, efficient, and accessible reverse engineering tools and techniques, (Raibulet et al., 2017). The overarching goal of MDRE is to enable organizations to prolong the functional lifespan of their software assets, enhance system performance, and support a seamless integration with new technologies, (Kienle & Müller, 2010). This contributes to the broader goal of achieving sustainable and forward-looking software development practices.

In this context, MDRE emerges as a vital discipline that not only addresses the immediate technical needs of software maintenance but also aligns with strategic objectives for long-term software asset management. The methodology's capacity to facilitate the understanding of complex software architectures and to support the evolution of these architectures' positions, (Favre, 1 C.E.). MDRE as an indispensable tool in the software engineer's repertoire. As such, MDRE is poised to play an increasingly significant role in the field of software engineering, particularly as systems become more interconnected and as the pace of technological change accelerates.

2.2. Concrete Syntax Tree (CST)

Also known as a parse tree, is a data structure that represents the syntactic structure of a source code written in a programming language. Unlike an Abstract Syntax Tree (AST), which abstracts away certain syntax details, a CST includes every detail from the source code, such as whitespace, comments, and punctuation. This comprehensive representation makes CSTs particularly useful for applications that require access to the exact original source code structure, like code formatters, linters, and style checkers, (Rakić & Budimac, 2013). The CST mirrors the grammar of the language and is generated during the parsing phase of compiler design, where it serves as an intermediary between the raw code and the more abstract representations used later in the compilation process.

The CST is a tree-like structure where each node corresponds to a grammatical construct present in the source code. The tree's branches represent the hierarchical relationship between the constructs, reflecting the nested nature of programming language syntax. For instance, an 'if' statement in the source code would be a parent node in the CST, with child nodes representing the condition and the body of the statement. Because it retains all syntactic information, a CST can be quite large and complex, especially for sizable source codes with many nested structures.

One of the main advantages of a CST is its ability to facilitate transformations on the source code while preserving the original formatting and comments, which is essential for refactoring tools and other applications that modify code without changing its behavior. Moreover, since the CST is a direct representation of the parsed code, it can be used to reproduce the original source code without any loss of information, making it an invaluable tool for reverse engineering and code analysis tasks, (Fondement, 2007).

In practice, while the CST is conceptually important, many compilers and interpreters opt to construct an AST directly for efficiency reasons, as the AST is typically smaller and easier to work with for semantic analysis and code generation,(Larose et al., 2023). Nevertheless, the CST remains a fundamental concept in compiler theory and an essential tool in scenarios where fidelity to the original source code is paramount. Its role in the software development lifecycle underscores the importance of understanding the syntactic structure of code and the need for precise tools to manage and improve it,(Aarssen & van der Storm, 2020).

2.3. Abstract Syntax Tree Metamodel (ASTM)

The Abstract Syntax Tree (AST) Metamodel is a foundational construct in the realm of software engineering, serving as a blueprint for the structured representation of the syntax of programming languages. It delineates a hierarchical framework that abstracts the grammatical structure of source code into a tree-like form, where each node corresponds to a syntactic construct present in the language. This metamodel is pivotal for a multitude of language processing applications, including compilers and interpreters, as it provides a means to navigate and manipulate the syntactic elements of a program systematically. The ASTM encapsulates not only the static structure of the syntax but also the relationships and rules that govern the construction of valid programs, (*About the Abstract Syntax Tree Metamodel Specification Version 1.0*, n.d.). It is designed to be language-agnostic, offering a generic template that can be specialized to fit the unique grammar of any specific programming language, (Son & Kim, 2017). In doing so, it facilitates the transformation of concrete syntax into an abstract representation that can be further analyzed or transformed by subsequent processes. The utility of the ASTM extends beyond parsing, as it is integral to the processes of semantic analysis, optimization, and code generation. Its role in Model-Driven Engineering (MDE) is particularly significant, where it acts as an intermediary between the concrete textual representation of code and the higher-level semantic models that drive system design and architecture. As such, the ASTM is not just a tool for understanding the structure of code, but a bridge that connects the detailed intricacies of programming languages with the broader conceptual models that underpin modern software development practices, (*About the Abstract Syntax Tree Metamodel Specification Version 1.0*, n.d.).

2.4. Knowledge Discovery Metamodel (KDM)

The Knowledge Discovery Metamodel (KDM) is an innovative standard from the Object Management Group (OMG) that serves as a pivotal framework for software analysis and modernization. Established as a meta-model, KDM provides a comprehensive representation of software systems, their components, relationships, and operational environments. This specification is integral to the field of Architecture-Driven Modernization (ADM), as it facilitates the understanding and transformation of legacy systems into modern architectures. KDM's structure is designed to encapsulate the various aspects of software assets, enabling a deep semantic integration of Application Lifecycle Management tools. It defines a common metadata framework that supports a wide range of software modernization activities, including but not limited to code analysis, comprehension, restructuring, and transformation, (Santos et al., 2019). KDM's inception can be traced back to the early 2000s, with the OMG's ADM Task Force recognizing the need for a standardized approach to legacy system modernization. The meta-model is organized into several layers, each targeting different facets of software systems. These layers include the Infrastructure Layer, Program Elements Layer, Resource Layer, and Abstractions Layer, which together provide a holistic view of software assets. The Infrastructure Layer captures the low-level system elements, while the Program Elements Layer deals with the higher-level constructs such as algorithms and data structures. The Resource Layer represents the physical and technological resources, and the Abstractions Layer encapsulates the conceptual elements of the system, (*About the Knowledge Discovery Metamodel Specification Version 1.4*, n.d.).

One of the key features of KDM is its ability to act as an intermediary representation, allowing for the

exchange of information between different tools and platforms. This interoperability is crucial for organizations looking to leverage diverse toolsets for their modernization efforts. KDM also supports the discovery of reusable components within existing software, promoting efficiency, and reducing the redundancy often associated with large-scale modernization projects, (Santos et al., 2019).

The specification has undergone several revisions, with the latest version, KDM 1.4, being formalized in December 2016. This version includes enhancements that further refine the model's capabilities and extend its applicability. The ongoing development and maintenance of KDM underscore its significance in the realm of software modernization, providing a robust foundation for organizations to revitalize their IT infrastructure and align it with contemporary business strategies and technological advancements. In essence, KDM is not just a tool for modernization but a strategic asset that enables continuous evolution and adaptation in an ever-changing technological landscape, (*About the Knowledge Discovery Metamodel Specification Version 1.4*, n.d.).

2.5. ANTLR4

ANTLR4, or ANOther Tool for Language Recognition, serves as a parser generator that is integral to the development of compilers, interpreters, and various language processing tools. Its framework is designed to be both robust and adaptable, enabling the definition of grammars for a wide range of context-free languages. Upon defining a grammar, ANTLR4 generates a parser capable of interpreting, processing, executing, or translating structured text or binary files. This functionality renders ANTLR4 a critical component in both academic research and industrial applications, (Parr, 2007).

The utility of ANTLR4 is evident in its facilitation of the parsing process, which is often complex. It provides a systematic approach from the initial design of a language to its practical implementation. ANTLR4 aids in the generation of abstract syntax trees, which are essential for semantic analysis, thus supporting the creation of software that can process and manipulate language constructs with precision, (Tomassetti, 2017).

The versatility of ANTLR4 is underscored by its compatibility with numerous programming languages, allowing for its integration into diverse development environments, (*Antlr/antlr4*, 2010/2024). Its reliability and efficiency have led to its adoption in significant projects and by prominent organizations. For instance, Twitter utilizes ANTLR4 for query parsing, while Hive and Pig employ it within Hadoop data analysis systems, (*ANTLR Testimonials*, n.d.).

Beyond its practical applications, ANTLR4's contribution to education is noteworthy. It is commonly used in university curricula to instruct students on the fundamentals of language recognition and parsing. ANTLR4 thus bridges the gap between theoretical concepts and their application, equipping developers with the means to translate structured language concepts into functional software, (Ortin et al., 2022).

3. Literature review on existing works

3.1. MoDisco

MoDisco, an Eclipse project, stands as a model-driven platform engineered to support the reverse engineering of legacy systems through comprehensive model extraction and analysis. Built upon the Eclipse Modeling Framework (EMF), MoDisco's architecture facilitates the creation and manipulation of software models, leveraging Java and the Eclipse plugin development environment to provide a solution that can be adapted and extended for various modernization initiatives, (Barbier et al., 2010).

At the heart of MoDisco's capabilities is the generation of Knowledge Discovery Metamodel (KDM) representations from existing source code. This multifaceted process commences with the parsing of source code to construct an intricate model that mirrors its structural and semantic attributes. The platform encompasses a suite of "discoverers" that autonomously scrutinize the code, yielding KDM models suitable for subsequent analysis, transformation, or documentation, (Brunelière et al., 2014).

The ecosystem of MoDisco is characterized by its breadth, offering an array of plugins and extensions that cater to a multitude of programming languages and technologies. This versatility permits

customization to meet specific project requirements, whether the focus is on Java, C++, or other programming languages. Moreover, MoDisco's integration with other tools within the Eclipse suite ensures a cohesive user experience for those acquainted with the Eclipse environment, (Bruneliere et al., 2010).

Notwithstanding its capabilities, MoDisco presents certain challenges. The tool's extensive feature set is accompanied by a significant learning curve, necessitating a solid grasp of model-driven engineering principles and familiarity with the Eclipse platform for effective utilization. Performance considerations also arise when processing large codebases, as the complexity of the resultant models may impede processing efficiency, (Scheidgen & Fischer, 2014).

MoDisco offers a robust framework for the modernization of legacy systems, aligning with the principles of contemporary software engineering through a model-driven methodology. The platform's proficiency in generating KDM models from source code positions it as a strategic asset for organizations aiming to decipher and reconstitute their software infrastructure. As such, MoDisco contributes to the broader discourse on sustainable software practices and the evolution of legacy systems within the field of software engineering.

3.2. Gra2Mol

Gra2Mol is a domain-specific language (DSL) tailored for the extraction of models from source code, particularly in the context of software modernization. It is designed to facilitate the transformation of text to models, or more specifically, grammar to models, which is a crucial step in reverse engineering processes. The architecture of Gra2Mol is centered around the concept of grammar-to-model transformation, where mappings between grammar elements and metamodel elements are explicitly defined. This approach allows for a structured and rule-based transformation process, where each rule specifies the correspondence between a source grammar element and a target metamodel element, (Izquierdo et al., 2008).

The technology behind Gra2Mol leverages the power of DSLs to simplify the definition of code-to-model transformations. It incorporates a powerful query language that enables the retrieval of scattered information from the source code's syntax tree. This feature is particularly useful for extracting detailed and accurate models that reflect the original source code's structure and semantics. The transformation process involves defining mappings through a special kind of assignment known as binding, which can either be locally resolved or trigger the execution of another rule, (Cánovas Izquierdo & Molina, 2009). Gra2Mol's ecosystem includes a development environment that supports various programming languages, making it a versatile tool for different modernization scenarios. It is part of a broader suite of tools that aim to bridge the gap between traditional grammarware and model-driven development (MDD) approaches. However, like any tool, Gra2Mol has its limitations. One of the main challenges is the potential complexity of creating transformations for large and intricate grammars. Additionally, the specificity of DSLs means that Gra2Mol may require significant customization when applied to languages or grammars outside its immediate scope.

In practice, Gra2Mol is used to obtain Knowledge Discovery Metamodel (KDM) representations from source code, which is a fundamental step in understanding and modernizing legacy systems. The process begins with the source code that conforms to a specific grammar, followed by the application of Gra2Mol transformations to produce a KDM model. This model then serves as a high-level abstraction of the source system, facilitating further analysis and transformation activities, (Izquierdo, n.d.).

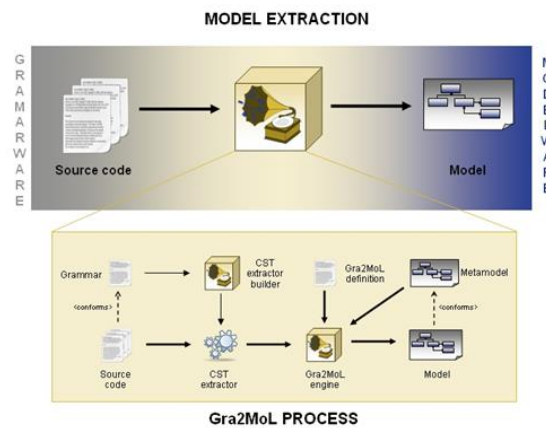


Fig. 1 : Execution process of Gra2Mol,

Gra2Mol represents a significant advancement in the field of software modernization, offering a specialized solution for the challenging task of reverse engineering. Its rule-based architecture, combined with a powerful query language and flexible transformation definitions, makes it an asset for organizations looking to modernize their legacy systems.

3.3. DBLModeller

DBLModeller is a tool developed to facilitate the extraction of Knowledge Discovery Metamodel (KDM) and Software Measurement Metamodel (SMM) models from the database layer of software systems. It is designed to assist in the model-driven reverse engineering process by providing a means to understand and analyze the structure and behavior of databases within legacy systems. The architecture of DBLModeller is built upon the Eclipse Modeling Framework (EMF), which allows for the creation and manipulation of models in a standardized way. The tool is primarily written in Java, which is known for its platform independence and robust ecosystem, and it utilizes HTML for its user interface components, (Ellison et al., 2016).

The utilization process of DBLModeller involves taking a SQL dump file, which contains the Data Definition Language (DDL) statements of the database, and a set of workload measurements provided as a CSV file. These inputs are then processed by DBLModeller to generate corresponding KDM and SMM models. This is particularly useful for understanding the database layer's impact on the overall performance and structure of the software system being analyzed, (Ellison et al., 2018).

DBLModeller's ecosystem benefits from the extensive support and community surrounding Java and Eclipse-based tools. This provides users with a range of resources for extending and integrating the tool within their existing development environments. However, the tool's reliance on specific input formats (SQL dump and CSV files) can be seen as a limitation, as it requires the data to be prepared in a certain way before processing. Additionally, being an Eclipse-based tool, it inherits the limitations of the Eclipse platform, including resource consumption and the need for familiarity with Eclipse plugins and development practices, (Milazzo et al., 2016).

In summary, DBLModeller represents a specialized solution for extracting and analyzing models from the database layer of software systems. Its integration with EMF and reliance on Java make it a powerful tool within the model-driven engineering community. However, its effectiveness is contingent upon the user's ability to provide the necessary inputs in the required formats and their proficiency with Eclipse-based development tools.

3.4. CS2AS

CS2AS, which stands for Concrete Syntax to Abstract Syntax, is a transformation tool that plays a

crucial role in bridging the gap between the concrete syntax (CS) of a textual language and its abstract syntax (AS), which is essential for model-driven engineering (MDE) tasks such as model transformation and code analysis. The architecture of CS2AS is designed to facilitate the conversion of human-readable programming languages into executable machine representations, a process that has traditionally been challenging due to the discrepancies between textual language grammars and model information. CS2AS addresses this by providing a declarative domain-specific transformation language (DSTL) that allows for the specification of complex CS-to-AS mappings, including name resolution and multi-way mappings that are often required for non-trivial languages like the Object Constraint Language (OCL), (Herrera et al., 2015).

The technology behind CS2AS leverages the Object Constraint Language (OCL) for specifying the transformation rules, which are then used to generate models conforming to the Knowledge Discovery Metamodel (KDM). This approach ensures that the transformation process is both checkable and reusable, leading to more consistent specifications and compliant tooling. The use of OCL as the foundation for CS2AS’s DSTL provides a robust and expressive language for defining the necessary transformations, making it a powerful tool for MDE practitioners, (Herrera et al., 2015).

In terms of its ecosystem, CS2AS is part of the Eclipse Modeling Framework, which provides a rich set of tools and plugins for model-driven development. This integration with Eclipse allows CS2AS to benefit from the extensive support and community surrounding the platform, enabling users to extend and customize the tool according to their specific needs, (Sanchez-Barbudo Herrera, 2017).

However, CS2AS is not without its limitations. The complexity of creating transformations for intricate grammars and the specificity required for DSLs mean that CS2AS may require significant customization when applied to languages or grammars outside its immediate scope. Additionally, the performance of CS2AS can be impacted when dealing with large and complex models, which may lead to slower processing times, (Sanchez-Barbudo Herrera, 2017).

Overall, CS2AS represents a significant advancement in the field of software modernization, offering a specialized solution for the challenging task of bridging the gap between concrete and abstract syntaxes. Its rule-based architecture, combined with the expressive power of OCL, makes it an asset for organizations looking to modernize their legacy systems and embrace model-driven engineering practices.

3.5. Critics to those tools

Tool	Architecture	Technologies Used	Utilization Process	Ecosystem	Limitations
MoDisco	Eclipse-based, extensible	Java, EMF	Parses source code to create KDM models	Eclipse Modeling Framework	Steep learning curve, performance with large codebases
Gra2Mol	Rule-based transformation	DSL, powerful query language	Transforms grammar elements to KDM elements	Flexible, supports various languages	Complexity with large grammars, customization required
DBLModeller	Eclipse-based, focuses on database layer	Java, HTML, EMF	Processes SQL dump and CSV to generate KDM and SMM	Java and Eclipse community	Specific input formats required, Eclipse platform limitations

CS2AS	Bridges concrete and abstract syntax	OCL for transformation rules	Converts human-readable code to executable models	Eclipse Modeling Framework	Customization for non-trivial languages, performance issues
-------	--------------------------------------	------------------------------	---	----------------------------	---

These tools represent a spectrum of approaches to software modernization, each with its strengths and weaknesses. MoDisco and DBLModeller share a common foundation in the Eclipse Modeling Framework, which provides a robust environment for model-driven engineering but also imposes a learning curve that may deter new users. Both tools are Java-based, benefiting from Java’s extensive ecosystem, yet they may struggle with performance when processing large and complex codebases.

Gra2Mol and CS2AS, on the other hand, emphasize the transformation aspect, with Gra2Mol focusing on grammar-to-model transformations and CS2AS on concrete syntax to abstract syntax mappings. While Gra2Mol’s DSL and query language offer powerful capabilities for model extraction, the tool’s complexity can be daunting, and it may require significant effort to adapt to different grammars. CS2AS’s use of OCL for specifying transformation rules provides expressiveness but also necessitates a deep understanding of the language for effective use.

Across the board, these tools face challenges in balancing flexibility with usability. While they offer powerful mechanisms for extracting KDM models from source code, their effectiveness is often contingent upon the user’s expertise and the specific requirements of the modernization project. The need for customization and the potential performance issues highlights the trade-offs involved in choosing the right tool for a given context. As the field of software modernization evolves, these tools must continue to adapt and address their limitations to meet the growing demands of complex legacy system transformations.

4. Proposed Approach

4.1. Details of the approach

In the proposed approach, ANTLR4 plays a critical role in the Model-Driven Reverse Engineering (MDRE) process by generating a visitor that traverses the parsed structure of the source language. This visitor is a crucial component that acts upon the grammar of the source language, which ANTLR4 uses to produce a parse tree. The visitor pattern implemented by ANTLR4 allows for operations to be performed on each node of this tree, effectively transforming the syntactic structure into semantic representations that correspond to the elements of the target metamodel. The transformation process involves mapping the constructs of the source language grammar to the appropriate abstractions in the metamodel, ensuring that the semantics of the original system are preserved and accurately reflected in the model. To instantiate the metamodel elements, the visitor methods are designed to create instances of the metamodel classes, which are then populated with the data extracted from the nodes of the parse tree. These instances collectively form the target model, which encapsulates the architectural and behavioral aspects of the source system in a structured and manipulable format. The target model is then serialized into an XMI file, which provides a standardized way to store and exchange the model data, facilitating further analysis, visualization, or transformation. This methodology not only streamlines the MDRE process but also enhances the fidelity of the resulting models, ensuring that they serve as an effective foundation for understanding and evolving the software system.

4.2. Implementation and Practical Case

This project’s implementation commenced with the establishment of a Java-based Gradle environment, structured to support Ecore and XSD metamodeling frameworks, and to process grammar specifications delineated in .g4 files.

The Gradle build automation system was configured to initiate ANTLR4, which generated the necessary files from the input grammars. This step was critical as it laid the foundation for subsequent model generation. Following this, the Eclipse Modeling Framework (EMF) Standalone libraries were employed to generate Java classes from the metamodeling specifications. This process translated the abstract metamodeling elements into concrete Java classes, which formed the backbone of the model manipulation and validation functionalities.

```

public GenModel convertXsdToEcore(String ecorePath, String genDirectory,
String genModelFileName, String javaGenDirectory, String basePackage) {
    // Map the platform resource URI to the Java generation directory

resourceSet.getURIConverter().getURIMap().put(URI.createURI("platform:/reso
urce/"), URI.createFileURI(new File(javaGenDirectory) + File.separator));
    // Load the Ecore package
    resourceSet.getPackageRegistry().put(GenModelPackage.eNS_URI,
GenModelPackage.eINSTANCE);
    // Load the Ecore file
    URI ecoreURI = URI.createURI(ecorePath, true);
    Resource resource = resourceSet.getResource(ecoreURI, true);
    EPackage ePackage = (EPackage) resource.getContents().get(0);
    // Create the GenModel
    GenModel genModel = GenModelFactory.eINSTANCE.createGenModel();
    genModel.getForeignModel().add(ecorePath);
    genModel.initialize(Collections.singleton(ePackage));
    genModel.setModelDirectory(genDirectory); // Use the genDirectory
variable
    String modelName = FileUtils.betweenLastDots(ePackage.getNsPrefix());
    genModel.setModelName(modelName);
    genModel.setComplianceLevel(GenJDKLevel.JDK210_LITERAL);
    genModel.setUpdateClasspath(false);
    genModel.setGenerateSchema(true);
    genModel.setCanGenerate(true);
    // Rename GenPackages for correct class naming
    List<GenPackage> genPackages =
genModel.getGenPackages().stream().toList();
    for (GenPackage genPackage : genPackages) {
        genPackage.setBasePackage(basePackage);

genPackage.setPrefix(FileUtils.betweenLastDots(genPackage.getPrefix()));
    }
    GenPackage genPackage = genModel.getGenPackages().get(0);
    genPackage.setBasePackage(basePackage);
    // Save the GenModel
    String genModelFilePath = genDirectory + "/" + genModelFileName +
".genmodel";
    URI genModelURI = URI.createURI(genModelFilePath, true);
    Resource genModelResource = resourceSet.createResource(genModelURI);
    if (genModelResource == null) {
        System.err.println("\tERROR: Unable to create a resource for the
URI: " + genModelFilePath);
        return null;
    }
    genModelResource.getContents().add(genModel);
    try {
        genModelResource.save(Collections.EMPTY_MAP);
    } catch (IOException e) {e.printStackTrace();}
    return genModel;
}

```

```

public void GenModelToJava(String genModelFile, String genPath) {
    PrintStream oldErr = System.err;
    try {
        System.setErr(new PrintStream(new ByteArrayOutputStream()));
    } catch (Throwable e) {
        oldErr = null;
    }

    try {
        File file = new File(genModelFile);
        Resource res =
this.resourceSet.getResource(URI.createFileURI(file.getAbsolutePath()),
true);

        GenModel genModel = ((GenModel) res.getContents().get(0));
        genModel.setModelDirectory("/");
        genModel.setCanGenerate(true);
        Generator generator;
        generator = new Generator() {
            @Override
            public JControlModel getJControlModel() {
                return new JControlModel();
            }
        };

generator.getAdapterFactoryDescriptorRegistry().addDescriptor(GenModelPacka
ge.eNS_URI, OnlyCodeGenModelGeneratorAdapterFactory.Descriptor);
generator.setInput(genModel);
generator.generate(genModel,
GenBaseGeneratorAdapter.MODEL_PROJECT_TYPE, this.monitor);
    } finally {
        if (oldErr != null) {
            System.setErr(oldErr);
        }
    }
}

```

The generation of the Concrete Syntax Tree (CST) was achieved through the implementation of a custom visitor pattern. This pattern extended ANTLR4's parse tree capabilities, enabling the traversal and processing of the parse tree to produce a CST representation. The CST was then serialized into an XMI file format, which facilitated the persistence and further manipulation of the model data. The chosen CST representation is inspired by Gra2Mol tool.

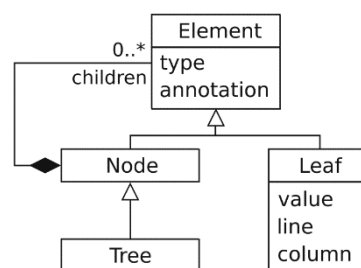


Fig. 2 : CST Metamodel

Model validation was performed using a dedicated method developed with EMF Standalone libraries. This method ensured that the generated model conformed to the metamodeling specifications by checking the structure and integrity of the model against the defined metamodel

```

public boolean validateModel(String modelString, String metaModelPath) {
    File file = new File(metaModelPath);
    Resource res =

```

```

this.resourceSet.getResource (URI.createFileURI (file.getAbsolutePath()),
true);
    var metaModel = (EPackage) res.getContents().get(0);
    resourceSet.getPackageRegistry().put (metaModel.getNsURI(),
metaModel);

    Resource resource1 =
resourceSet.createResource (URI.createURI ("src/main/resources/temp.xmi"));
    try {
        resource1.load(new
java.io.ByteArrayInputStream(modelString.getBytes()), null);
       EObject modelRoot = resource1.getContents().get(0);

        // Créer un Diagnostician pour valider le modèle
        Diagnostician diagnostician = new Diagnostician();
        Diagnostic diagnostic = diagnostician.validate(modelRoot);
        boolean isValid = diagnostic.getSeverity() != Diagnostic.ERROR;
        if (!isValid) {
            for (Diagnostic childDiagnostic : diagnostic.getChildren())
            {
                System.out.println(childDiagnostic.getMessage());
            }
        }
        return isValid;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    } finally {
        resource1.unload();
    }
}

```

This is the source code we choose as an input in our test:

```

import java.util.Scanner;
public class HelloWorld {
    public static void main(String[] args) {
        // Creates a reader instance which takes
        // input from standard input - keyboard
        Scanner reader = new Scanner(System.in);
        System.out.print("Enter a number: ");
        //nextInt() reads the next integer from the keyboard
        int number = reader.nextInt();
        //println() prints the following line to the output screen
        System.out.println("You entered: " + number);
    }
};

```

This is an excerpt of the generated CST by our method:

```

<?xml version="1.0" encoding="UTF-8"?>
<CST:Node xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:CST="urn:local:Code2KDM:CST:v1"
kind="CST">
    <children xsi:type="CST:Node" kind="compilationUnit">
        <children xsi:type="CST:Node" kind="importDeclaration">
            <children xsi:type="CST:Node" kind="singleTypeImportDeclaration">
                <children xsi:type="CST:Leaf" kind="IMPORT" value="import" line="1"/>
                <children xsi:type="CST:Node" kind="typeName">
                    <children xsi:type="CST:Node" kind="packageOrTypeName">
                        <children xsi:type="CST:Node" kind="packageOrTypeName">

```

```

        <children xsi:type="CST:Leaf" kind="Identifier" value="java" pos="7" line="1"/>
    </children>
    <children xsi:type="CST:Leaf" kind="DOT" value="." pos="11" line="1"/>
    <children xsi:type="CST:Leaf" kind="Identifier" value="util" pos="12" line="1"/>
</children>
    <children xsi:type="CST:Leaf" kind="DOT" value="." pos="16" line="1"/>
    <children xsi:type="CST:Leaf" kind="Identifier" value="Scanner" pos="17" line="1"/>
</children>
    <children xsi:type="CST:Leaf" kind="SEMI" value=";" pos="24" line="1"/>
</children>
</children>
<children xsi:type="CST:Node" kind="typeDeclaration">
    <children xsi:type="CST:Node" kind="classDeclaration">
        <children xsi:type="CST:Node" kind="normalClassDeclaration">
            <children xsi:type="CST:Node" kind="classModifier">
                <children xsi:type="CST:Leaf" kind="PUBLIC" value="public" line="3"/>
            </children>
            <children xsi:type="CST:Leaf" kind="CLASS" value="class" pos="7" line="3"/>
            <children xsi:type="CST:Leaf" kind="Identifier" value="HelloWorld" pos="13"
line="3"/>
            <children xsi:type="CST:Node" kind="classBody">
                <children xsi:type="CST:Leaf" kind="LBRACE" value="{ " pos="24" line="3"/>
                <children xsi:type="CST:Node" kind="classBodyDeclaration">
                    <children xsi:type="CST:Node" kind="classMemberDeclaration">
                        <children xsi:type="CST:Node" kind="methodDeclaration">
                            <children xsi:type="CST:Node" kind="methodModifier">
                                <children xsi:type="CST:Leaf" kind="PUBLIC" value="public" pos="4"
line="5"/>
                            </children>
                            <children xsi:type="CST:Node" kind="methodModifier">
                                <children xsi:type="CST:Leaf" kind="STATIC" value="static" pos="11"
line="5"/>
                            </children>
                            <children xsi:type="CST:Node" kind="methodHeader">
                                <children xsi:type="CST:Node" kind="result">
                                    <children xsi:type="CST:Leaf" kind="VOID" value="void" pos="18"
line="5"/>
                                </children>
                                <children xsi:type="CST:Node" kind="methodDeclarator">
                                    <children xsi:type="CST:Leaf" kind="Identifier" value="main" pos="23"
line="5"/>
                                    <children xsi:type="CST:Leaf" kind="LPAREN" value="(" pos="27"
line="5"/>
                                    <children xsi:type="CST:Node" kind="formalParameterList">
                                        <children xsi:type="CST:Node" kind="lastFormalParameter">
                                            <children xsi:type="CST:Node" kind="formalParameter">
                                                <children xsi:type="CST:Node" kind="unannType">
                                                    <children xsi:type="CST:Node" kind="unannReferenceType">
                                                        <children xsi:type="CST:Node" kind="unannArrayType">
                                                            <children xsi:type="CST:Node"
kind="unannClassOrInterfaceType">
                                                                <children xsi:type="CST:Node"
kind="unannClassType_lfno_unannClassOrInterfaceType">
                                                                    <children xsi:type="CST:Leaf" kind="Identifier"
value="String" pos="28" line="5"/>
                                                                </children>
                                                            </children>
                                                        </children>
                                                    </children>
                                                </children>
                                            </children>
                                        </children>
                                    </children>
                                </children>
                            </children>
                        </children>
                    </children>
                </children>
            </children>
        </children>
    </children>
</children>

```

The Abstract Syntax Tree Metamodel (ASTM) was generated using a similar approach. A custom visitor, implementing the ANTLR4 visitor interface, was created to traverse the parse tree. For each node visited, an instance of the corresponding class in the target metamodel was instantiated, effectively building the ASTM. The generated ASTM was then validated using the same EMF Standalone library-based method, ensuring its structural fidelity to the metamodel.

The Knowledge Discovery Metamodel (KDM) was processed in an analogous manner, applying the same generation and validation techniques to maintain a uniform implementation approach across different metamodels

5. Evaluation and Results

5.1. Evaluation criteria for the proposed approach.

To conduct a rigorous and meaningful evaluation of Model-Driven Reverse Engineering (MDRE) tools, it is imperative to establish a set of criteria that are both comprehensive and pertinent to assessing their utility and performance. The criteria selected for this evaluation are designed to cover critical aspects of MDRE tools, ensuring a thorough analysis of their capabilities and limitations.

Firstly, Metamodel Definition and Reuse is an essential criterion because it addresses the foundational structures upon which reverse engineering is built. This criterion examines whether an approach employs existing metamodels, such as the Knowledge Discovery Metamodel (KDM), or if it necessitates the creation of new metamodels. The reuse of established metamodels is advantageous as it facilitates interoperability with other tools and frameworks, potentially reducing the need for extensive customization. This aspect is crucial for determining how well an MDRE approach can be integrated into existing development and maintenance environments.

Tool Implementation and Reuse evaluates the extent to which existing software tools are utilized or whether new tools are developed as part of the MDRE approach. Leveraging established tools can enhance the robustness and reliability of the reverse engineering process, as these tools have typically undergone extensive testing and validation. Conversely, the introduction of new tools may require additional learning and adaptation, which could impact the overall efficiency and adoption of the approach. This criterion is significant for understanding the practical implications of implementing the MDRE approach in different contexts.

The Automation Level criterion assesses the degree of automation in the transformation processes defined by the MDRE approaches. This includes distinguishing between fully automated, semi-automated, and manual processes. High levels of automation are particularly important for improving the efficiency and scalability of reverse engineering tasks, as they minimize the need for human intervention and reduce the likelihood of errors. Evaluating the automation level provides insights into the potential labor savings and consistency of the generated models.

Scope of the Approach differentiates between general-purpose approaches and those tailored for specific applications. General-purpose approaches are designed to be adaptable to a wide range of domains and systems, making them versatile and broadly applicable. However, specialized approaches might offer more targeted solutions that are optimized for specific contexts but may lack flexibility. This criterion is crucial for assessing the breadth of applicability of an MDRE tool and its potential utility across different projects and domains.

Case Studies and Practical Applications examine the extent to which the MDRE approaches have been validated through empirical studies and real-world applications. Case studies provide concrete evidence of the effectiveness of an approach, demonstrating its practical viability and highlighting its strengths and weaknesses in actual use cases. This criterion ensures that the methodologies are not only theoretically sound but also practically applicable and beneficial in real-world scenarios.

Lastly, the Type of Analysis criterion considers whether the approaches employ static, dynamic, or hybrid analysis techniques. Static analysis involves examining the source code without executing it, which is less resource-intensive and suitable for certain types of analysis. Dynamic analysis, on the other hand, involves executing the system to capture runtime behaviors, providing a more comprehensive understanding of the system's operational characteristics. Hybrid analysis combines both static and dynamic techniques to offer a more holistic view, though it requires careful management of computational resources. Evaluating the type of analysis used helps in understanding the comprehensiveness and robustness of the MDRE approaches.

These criteria were chosen to provide a nuanced and detailed comparison of MDRE tools, addressing both their theoretical foundations and practical implementations. By evaluating these aspects, the goal is to guide practitioners in selecting the most appropriate tools for their specific needs, thereby advancing the field of reverse engineering, and promoting more efficient and effective software maintenance and evolution practices.

Our approach to Model-Driven Reverse Engineering (MDRE) utilizing ANTLR4 is rigorously

evaluated based on several essential criteria to assess its effectiveness, practicality, and applicability comprehensively.

Firstly, Metamodel Definition and Reuse examines our method's utilization of custom metamodels alongside existing ones such as the Knowledge Discovery Metamodel (KDM). This approach allows for tailored solutions to specific issues while ensuring compatibility with established frameworks, promoting interoperability. By reusing established metamodels, our approach reduces the effort required for creating new structures from scratch, facilitating easier integration with other tools and enhancing overall flexibility.

Tool Implementation and Reuse is critical for understanding the reliance on existing software tools versus the development of new ones. Our method leverages ANTLR4, a widely recognized parser generator, which aids in parsing and model generation. The adoption of ANTLR4 capitalizes on its robustness and reliability, mitigating the need for extensive development of new tools and thereby streamlining the implementation process. This reliance on an established tool enhances the approach's reliability and integration capabilities, contributing to its efficiency.

The Automation Level criterion assesses the degree of automation in our approach. The parsing and initial model generation processes are fully automated, significantly reducing the manual effort required in the early stages of reverse engineering. However, the validation and refinement of the generated models necessitate manual intervention. This semi-automated approach ensures that while the initial phases are efficient and swift, the subsequent manual validation maintains the accuracy and quality of the final models. This balance is crucial for achieving both efficiency and precision in reverse engineering tasks.

The Scope of the Approach criterion evaluates the general applicability of our method. Our approach is designed to be general-purpose, adaptable to various programming languages and systems. This broad scope makes it versatile, allowing it to be applied across different domains, ranging from enterprise applications to embedded systems. The general-purpose nature of the approach ensures that it can address a wide array of reverse engineering challenges, enhancing its utility in diverse contexts.

Case Studies and Practical Applications provide empirical evidence of the approach's practical viability. Our method has been tested on Java Gradle projects, demonstrating its effectiveness in real-world scenarios. These case studies illustrate how the approach handles complex software engineering challenges, validating its practical application. The empirical validation through case studies is crucial for establishing the approach's credibility and utility in both academic research and industrial practice. Lastly, the Type of Analysis criterion examines the analytical techniques employed. Our approach primarily uses static analysis, utilizing ANTLR4 to parse source code and generate corresponding models. However, the architecture of our method is adaptable, allowing for the incorporation of dynamic analysis techniques when necessary. This flexibility ensures a comprehensive understanding of the system, capturing both static structural information and dynamic runtime behaviors if required. By accommodating both static and dynamic analysis, the approach provides a holistic view of the system, enhancing its robustness and comprehensiveness.

In summary, our approach, when evaluated against these rigorous criteria, demonstrates a well-structured solution for MDRE. It balances automation with manual precision, offers broad applicability across various domains, and is empirically validated through practical case studies. This detailed evaluation underscores the approach's strengths and highlights areas for potential enhancement, contributing to the advancement of reverse engineering practices.

5.2. Comparison with existing methods.

To comprehensively evaluate our approach against established Model-Driven Reverse Engineering (MDRE) tools such as MoDisco, Gra2Mol, CS2AS, and DBLModeller, we examine key criteria: Metamodel Definition and Reuse, Tool Implementation and Reuse, Automation Level, Scope of the

Approach, Case Studies and Practical Applications, and Type of Analysis.

Metamodel Definition and Reuse focuses on the use of predefined versus custom metamodels. Our approach, employing ANTLR4, utilizes both custom metamodels and existing ones like the Knowledge Discovery Metamodel (KDM), balancing specificity with reusability. MoDisco, heavily aligned with OMG standards, also defines its metamodels but is more stringent in adhering to established norms. Gra2Mol, while allowing custom metamodels, often requires complex, specific transformation rules. CS2AS defines unique metamodels tailored to target languages, necessitating frequent adaptations. DBLModeller focuses specifically on database metamodels, limiting its versatility.

Tool Implementation and Reuse examines reliance on existing tools. Our approach leverages ANTLR4, ensuring robust parsing and model generation while reducing the need for new tool development. MoDisco integrates deeply with Eclipse, benefiting from the rich ecosystem but presenting a steep learning curve. Gra2Mol requires intricate rule configurations, complicating its setup. CS2AS uses specific model-to-model transformation tools, often necessitating manual adjustments. DBLModeller, confined to database modeling, uses specialized tools tailored to its niche, limiting broader applicability. Automation Level addresses the degree of process automation. Our approach is semi-automated, automating initial parsing and model generation while requiring manual validation. MoDisco offers semi-automation, with substantial manual intervention in larger projects. Gra2Mol, despite powerful transformation capabilities, demands manual configuration for each grammar. CS2AS combines semi-automation with frequent manual adjustments. DBLModeller achieves higher automation within its database-focused scope, necessitating minimal intervention.

Scope of the Approach evaluates applicability across domains. Our general-purpose approach adapts to various programming languages and systems, offering broad applicability. MoDisco, though versatile, is mainly centered on Java and Eclipse ecosystems. Gra2Mol, specialized in grammar transformations, lacks flexibility. CS2AS, while effective in specific language contexts, requires significant adjustments for new projects. DBLModeller’s specialization in databases limits its general application.

Case Studies and Practical Applications highlight empirical validation. Our approach has been tested on Java Gradle projects, demonstrating its practical viability. MoDisco, with various case studies, predominantly focuses on Java projects. Gra2Mol, validated in specific grammar transformation scenarios, presents fewer case studies. CS2AS shows effectiveness in specialized cases but requires frequent adaptations. DBLModeller, limited to database applications, lacks broader case studies.

Type of Analysis considers static versus dynamic techniques. Our approach primarily employs static analysis with ANTLR4 but is adaptable to include dynamic analysis. MoDisco focuses on static analysis with potential dynamic extensions. Gra2Mol relies on static analysis based on transformation rules. CS2AS combines static and dynamic analysis for comprehensive system understanding. DBLModeller, confined to static database analysis, shows limited dynamic capability.

Criteria of comparison	Our Approach	MoDisco	Gra2Mol	CS2AS	DBLModeller
Metamodel Definition and Reuse	Custom and existing metamodels (e.g., KDM)	Defines its own, OMG standards-focused	Custom metamodels, specific transformation rules	Custom metamodels specific to target languages	Specific to database metamodels
Tool Implementation and Reuse	Uses ANTLR4, reduces need for new tool development	Deep integration with Eclipse, steep learning curve	Complex rule configurations	Specific model-to-model transformation tools	Specialized tools for database modeling
Automation	Semi-automated,	Semi-	Powerful but	Semi-	Highly

Level	manual validation required	automated, significant manual processes for large projects	requires manual configuration	automated, frequent manual adjustments required	automated for databases, minimal interventions
Scope of the Approach	General-purpose, adaptable to various languages and systems	Versatile, mainly Java and Eclipse ecosystems	Specialized in grammar transformations, less flexible	Specific to target languages, significant project adjustments	Limited to databases
Case Studies and Practical Applications	Tested on Java Gradle projects, practical viability	Various case studies, mainly Java projects	Fewer case studies, focused on grammar transformations	Effective in specialized cases, frequent adaptations required	Limited to database applications
Type of Analysis	Primarily static, adaptable to dynamic analysis	Static analysis with potential dynamic extensions	Static analysis based on transformation rules	Combines static and dynamic analysis	Static database analysis, limited dynamic capability

5.3. Discussion on the advantages and limitations of the approach.

In the domain of Model-Driven Reverse Engineering (MDRE), the proposed methodology distinguishes itself by its capacity to integrate both bespoke and extant metamodels, such as the Knowledge Discovery Metamodel (KDM). This integration facilitates a versatile application spectrum, in contrast to MoDisco, which, despite its robustness and alignment with Object Management Group (OMG) standards, may exhibit limitations in adaptability due to its reliance on predefined metamodels.

The employment of ANTLR4 for tool implementation streamlines the development process, curtailing the necessity for constructing new tools ab initio. This efficiency is advantageous when juxtaposed with the intricate integration and the steep learning trajectory associated with Eclipse-centric tools like MoDisco, or the elaborate rule configurations necessitated by Gra2Mol.

The semi-automated nature of the proposed approach, necessitating manual validation, ensures a judicious equilibrium between automation and precision. While DBLModeller proffers a high degree of automation for database-centric applications, it may not possess the requisite flexibility for more expansive applications. The manual validation component of the proposed approach guarantees accuracy and dependability, albeit at the expense of additional temporal investment.

The scope of the proposed approach is general-purpose, rendering it adaptable to a diverse array of languages and systems. This adaptability is a salient advantage over more specialized tools such as CS2AS and Gra2Mol, which are tailored to specific target languages and grammar transformations, respectively. The adaptability of the proposed approach has been substantiated through its practical viability in case studies involving Java Gradle projects.

Nonetheless, the approach is not without areas that warrant enhancement. The semi-automated aspect could benefit from the integration of machine learning algorithms to further diminish the necessity for manual intervention. Moreover, an expansion of the scope of case studies to encompass a broader array of projects beyond Java Gradle could provide a more comprehensive elucidation of the approach's applicability and constraints.

In conclusion, the proposed approach to MDRE presents a promising trajectory for researchers and practitioners in pursuit of a versatile and adaptable solution. Future enhancements could concentrate on

augmenting automation and extending the range of practical applications to consolidate its stature as a preeminent tool in the field. This discourse is intended to contribute to the scholarly dialogue within the Association for Computing Machinery (ACM) community, propelling forward the evolution of MDRE methodologies.

6. Conclusion and Future Work

The proposed approach to Model-Driven Reverse Engineering (MDRE) leveraging ANTLR4 represents a significant advancement in streamlining the reverse engineering process and enhancing the fidelity of generated models. By integrating ANTLR4-generated visitor patterns with metamodeling, the methodology addresses the challenges of parsing complex source languages, accurately instantiating metamodel elements, and generating target models faithful to the original system. The integration with established metamodels, such as the Knowledge Discovery Metamodel (KDM), further enhances the approach's versatility and interoperability.

While the current approach demonstrates promising results, as evidenced by the case study involving a Java Gradle project, future research should focus on expanding the range of practical applications to diverse programming languages and systems. This expansion would provide a more comprehensive evaluation of the approach's limitations and areas for improvement. Additionally, incorporating machine learning techniques could potentially enhance the automation level and reduce the need for manual validation, further improving the efficiency of the MDRE process.

Moreover, investigating the integration of dynamic analysis techniques alongside the current static analysis approach could yield a more holistic understanding of complex systems, capturing both structural and runtime behaviors. Such a comprehensive analysis would be particularly valuable for large-scale legacy systems with intricate interactions and dependencies.

In conclusion, while the current approach offers a robust and adaptable solution for MDRE, ongoing research and development are crucial for addressing its limitations and expanding its applicability. By focusing on automation, diverse applications, dynamic analysis, and community collaboration, future work can enhance the methodology's effectiveness, ultimately advancing the field of software reverse engineering. (Korchi, Khachouch, Lakhrissi, et al., 2024), (Korchi, Khachouch, & Lakhrissi, 2024).

References

Aarssen, R. T. A., & van der Storm, T. (2020). High-fidelity metaprogramming with separator syntax trees. *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 27–37. <https://doi.org/10.1145/3372884.3373162>

About the Abstract Syntax Tree Metamodel Specification Version 1.0. (n.d.). Retrieved May 25, 2024, from <https://www.omg.org/spec/ASTM/1.0/About-ASTM>

About the Knowledge Discovery Metamodel Specification Version 1.4. (n.d.). Retrieved May 25, 2024, from <https://www.omg.org/spec/KDM/1.4/About-KDM>

Agt-Rickauer, H. (2020). Supporting domain modeling with automated knowledge acquisition and modeling recommendations. <https://depositonce.tu-berlin.de/handle/11303/10700>

ANTLR Testimonials. (n.d.). Retrieved May 25, 2024, from <https://www.antlr.org/testimonials.html>

Antlr/antlr4. (2024). [Java]. Antlr Project. <https://github.com/antlr/antlr4> (Original work published 2010)

Barbier, G., Bruneliere, H., Jouault, F., Lennon, Y., & Madiot, F. (2010). Chapter 14—MoDisco, a Model-Driven Platform to Support Real Legacy Modernization Use Cases. In W. M. Ulrich & P. H. Newcomb (Eds.), *Information Systems Transformation* (pp. 365–400). Morgan Kaufmann. <https://doi.org/10.1016/B978-0-12-374913-0.00014-7>

- Belfrit Victor. (2013). Revisiting legacy systems and legacy modernization from the industrial perspective [Master Thesis]. <https://studenttheses.uu.nl/handle/20.500.12932/14919>
- Bézivin, J. (2005). On the unification power of models. *Software & Systems Modeling*, 4(2), 171–188. <https://doi.org/10.1007/s10270-005-0079-0>
- Bruneliere, H. (2018). Generic Model-based Approaches for Software Reverse Engineering and Comprehension [PhD Thesis]. <http://www.theses.fr/2018NANT4040/document>
- Brunelière, H., Cabot, J., Dupé, G., & Madiot, F. (2014). MoDisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8), 1012–1032. <https://doi.org/10.1016/j.infsof.2014.04.007>
- Bruneliere, H., Cabot, J., Jouault, F., & Madiot, F. (2010). MoDisco: A generic and extensible framework for model driven reverse engineering. *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 173–174. <https://doi.org/10.1145/1858996.1859032>
- Cánovas Izquierdo, J. L., & Molina, J. G. (2009). A Domain Specific Language for Extracting Models in Software Modernization. In R. F. Paige, A. Hartman, & A. Rensink (Eds.), *Model Driven Architecture—Foundations and Applications* (pp. 82–97). Springer Berlin Heidelberg.
- Chandrasegaran, S. K., Ramani, K., Sriram, R. D., Horváth, I., Bernard, A., Harik, R. F., & Gao, W. (2013). The evolution, challenges, and future of knowledge representation in product design systems. *Computer-Aided Design*, 45(2), 204–228. <https://doi.org/10.1016/j.cad.2012.08.006>
- Ellison, M., Calinescu, R., & Paige, R. F. (2016). Towards Platform Independent Database Modelling in Enterprise Systems. In P. Milazzo, D. Varró, & M. Wimmer (Eds.), *Software Technologies: Applications and Foundations* (pp. 42–50). Springer International Publishing.
- Ellison, M., Calinescu, R., & Paige, R. F. (2018). Evaluating cloud database migration options using workload models. *Journal of Cloud Computing*, 7(1), 6. <https://doi.org/10.1186/s13677-018-0108-5>
- Farmer, R., & Gruba, P. (2006). Towards model-driven end-user development in CALL. *Computer Assisted Language Learning*. <https://doi.org/10.1080/09588220600821529>
- Favre, L. (1 C.E.). *Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution*. In <https://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-61520-649-0>. IGI Global. <https://www.igi-global.com/gateway/book/37247>
- Fondement, F. (Ed.). (2007). *Concrete syntax definition for modeling languages*. EPFL. <https://doi.org/10.5075/epfl-thesis-3927>
- García-Borgoñón, L., Barcelona, M. A., Egea, A. J., Reyes, G., Sainz-de-la-maza, A., & González-Uzabal, A. (2023). Lessons Learned in Model-Based Reverse Engineering of Large Legacy Systems. In M. Indulska, I. Reinhartz-Berger, C. Cetina, & O. Pastor (Eds.), *Advanced Information Systems Engineering* (pp. 330–344). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-34560-9_20
- Herrera, A. S.-B., Willink, E. D., & Paige, R. F. (2015). An OCL-based Bridge from Concrete to Abstract Syntax. *Proceedings of the 15th International Workshop on OCL and Textual Modeling Co-Located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015)*, 19–34.
- Izquierdo, J. L. C. (n.d.). Gra2MoL put into practice.
- Izquierdo, J. L. C., Cuadrado, J. S., & Molina, J. G. (2008). Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization. 1–8.

- Kienle, H. M., & Müller, H. A. (2010). Chapter 5 - The Tools Perspective on Software Reverse Engineering: Requirements, Construction, and Evaluation. In *Advances in Computers* (Vol. 79, pp. 189–290). Elsevier. [https://doi.org/10.1016/S0065-2458\(10\)79005-7](https://doi.org/10.1016/S0065-2458(10)79005-7)
- Korchi, A., Khachouch, M. K., & Lakhrissi, Y. (2024). A Model-Driven Architecture Solution for Multi-Platform Mobile App Development. *Journal of System and Management Sciences*, 14(5). <https://doi.org/10.33168/JSMS.2024.0501>
- Korchi, A., Khachouch, M. K., Lakhrissi, Y., Marzouki, N. E., Moumen, A., & Mohajir, M. E. (2024). Classification of existing mobile cross-platform approaches and proposal of decision support criteria. *International Journal of Information and Communication Technology*, 24(1), 86–111.
- Larose, O., Kaleba, S., Burchell, H., & Marr, S. (2023). AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2), 233:318-233:346. <https://doi.org/10.1145/3622808>
- Milazzo, P., Varró, D., & Wimmer, M. (Eds.). (2016). *Software Technologies: Applications and Foundations: STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna Austria, July 4-8, 2016, Revised Selected Papers* (Vol. 9946). Springer International Publishing. <https://doi.org/10.1007/978-3-319-50230-4>
- Ortin, F., Quiroga, J., Rodriguez-Prieto, O., & Garcia, M. (2022). An empirical evaluation of Lex/Yacc and ANTLR parser generation tools. *PLoS ONE*, 17(3), e0264326. <https://doi.org/10.1371/journal.pone.0264326>
- Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-specific Languages*. Pragmatic Bookshelf.
- Raibulet, C., Fontana, F. A., & Zanoni, M. (2017). Model-driven reverse engineering approaches: A systematic literature review. *Ieee Access*, 5, 14516–14542.
- Rakić, G., & Budimac, Z. (2013, October 2). Introducing Enriched Concrete Syntax Trees. *arXiv.Org*. <https://arxiv.org/abs/1310.0802v1>
- Sabir, U., Azam, F., Haq, S. U., Anwar, M. W., Butt, W. H., & Amjad, A. (2019). A Model Driven Reverse Engineering Framework for Generating High Level UML Models From Java Source Code. *IEEE Access*, 7, 158931–158950. <https://doi.org/10.1109/ACCESS.2019.2950884>
- Sanchez-Barbudo Herrera, A. (2017). *Auto-tooling to Bridge the Concrete and Abstract Syntax of Complex Textual Modeling Languages* [Engd, University of York]. <https://etheses.whiterose.ac.uk/17416/>
- Santos, B. M., Landi, A. D. S., Santibáñez, D. S., Durelli, R. S., & De Camargo, V. V. (2019). Evaluating the extension mechanisms of the knowledge discovery metamodel for aspect-oriented modernizations. *Journal of Systems and Software*, 149, 285–304. <https://doi.org/10.1016/j.jss.2018.12.011>
- Scheidgen, M., & Fischer, J. (2014). Model-Based Mining of Source Code Repositories. In D. Amyot, P. Fonseca i Casas, & G. Mussbacher (Eds.), *System Analysis and Modeling: Models and Reusability* (pp. 239–254). Springer International Publishing.
- Son, H. S., & Kim, R. Y. C. (2017). xCodeParser based on Abstract Syntax Tree Metamodel (ASTM) for SW Visualization. *International Information Institute (Tokyo). Information*, 20(2A), 963–968.
- Stahl, T., Völter, M., Bettin, J., Haase, A., & Helsen, S. (2006). *Model-driven software development—Technology, engineering, management*. Pitman. <http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470025700.html>

Tomassetti, G. (2017, March 8). The ANTLR Mega Tutorial. Strumenta. <https://tomassetti.me/antlr-mega-tutorial/>

Yun, C. (2013). Reverse Engineering Java Code with Epsilon.