Towards a Model Transformation based Code Renovation Tool

Norbert Somogyi[a], Gábor Kövesdán[b] and László Lengyel[c]
Budapest University of Technology and Economics, Műegyetem rkp. 3, Budapest, Hungary

Keywords: Code Modernization, Modeling, Model Transformation.

Abstract: Maintaining legacy software has always required considerable effort in software engineering. To alleviate these efforts, extensive research has been dedicated to automate the modernization of such systems. The process includes several challenges, such as the syntactic translation of the old software to a modern programming language, the mapping of the type systems of the source and target languages and the paradigm shift if the two languages use different approaches, such as transforming procedural code to the object-oriented or functional paradigm. In the case of procedural to object-oriented transformations, the state-of-the-art solutions are not capable of automatically producing satisfactory results and some researchers suggest that complete automation will never be achieved. In our paper, we report on our work in progress on using recent advances in the fields of modeling and model transformation to build a software modernization tool. Our solution capitalizes on the advantages of the Ecore-based modeling ecosystem of Eclipse and focuses on not just the syntactic translation of the system, but also on the paradigm shift of procedural to object-oriented transformations. Our approach builds a semantic model from the original source code written in C language and produces Java code by analysing and transforming this model.

1 INTRODUCTION

Software is a living artifact that is rarely left alone once developed. As software evolves and is continuously modified, the structure of the source code becomes less clear, which makes it much harder to understand its original intent, modify it or extend it with additional features. This is called code rot (Izurieta and Bieman, 2013).

It would be desirable for software engineers to periodically review and reconsider the technology stack of a particular software and migrate to a more modern and ideal option. However, it is rarely implemented for two main reasons. First, when done manually, it assumes the risk of introducing so-called regressions, bugs that are accidentally introduced by modifying software. Secondly, corporations are driven by business goals and are less likely to allocate resources on projects that do not provide value for the market but are solely for maintenance purposes. Code modernization tools could help in this challenge by at least partly automating this process, mitigating the risk of regressions and cutting down on costs. In this paper, we report on our work in progress towards a code modernization tool that modernizes C code using modeling and model transformation techniques. Our tool uses the Ecore-based modeling ecosystem of Eclipse (McAffer et al., 2010) and focuses on translating procedural software into an object-oriented structure using model transformations. The tool produces Java code, but the approach and the tool itself are extensible to support different source and target languages as well.

Based on the conclusions drawn from the cited papers and tools in Section 2, we believe that there is a lack of procedural to object-oriented code modernization tools that actually use the various object identification methods and algorithms proposed in research. Therefore, we aim to build a semi-automatic tool that performs the conversion from procedural C code to object-oriented Java code without interaction as much as possible. We also propose two novel approaches when tackling this problem: call chain analysis and using a DSL (Domain Specific Language) for the transformation of standard library calls.

Our main focus is to combine and expand upon the various object identification methods proposed by existing literature with the Ecore-based modeling ecosystem of Eclipse, which we believe to be a huge advantage in the field of code modernization. The

[a] https://orcid.org/0000-0001-6908-7907
[b] https://orcid.org/0000-0003-1527-2896
[c] https://orcid.org/0000-0002-5129-4198

Somogyi, N., Kövesdán, G. and Lengyel, L.
Towards a Model Transformation based Code Renovation Tool.
DOI: 10.5220/0010246401910198
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 191-198
ISBN: 978-989-758-487-9

reason for this is because it is a well-developed and well-known ecosystem that consists of a great deal of various model-driven tools. Many of these are greatly compatible with each other because all of them use EMF (Eclipse Modeling Framework) models. Thus this gives the opportunity to integrate these tools into a code modernization framework, which can greatly increase the capacities of the tool and reduce the costs of developing it.

The paper is structured in the following way. Section 2 reviews related work, state-of-the-art tools and highlights the challenges in the field. In Section 3, we present our approach and the model transformations used during the translation process. In Section 4, we evaluate the feasibility of our solution, exploring the advantages and shortcomings of our work. Section 5 concludes the paper, highlighting some future work.

2 RELATED WORK

2.1 Challenges in the Field of Code Modernization

Code modernization has a number of cornerstones that must be considered when aiming to build such solutions. In their paper, Terekhov and Verhoef (Terekhov and Verhoef, 2000) have highlighted the difficulties of automated source code modernization that are beyond purely syntactic translation. Firstly, the target programming language may lack some of the source language's features that must be simulated in the target language. Another challenge is mapping the data type of variables. Different programming languages have a wide range of differences regarding data types, e.g. C has a universal pointer type (void *) that can refer to arbitrary data in the memory, whereas Java deliberately lacks such primitive type to avoid bugs that originate in manual memory management. Moreover, data types that are missing in the target language must be emulated. Furthermore, some languages, such as Cobol, have several dialects that differ syntactically and also semantically. Semantic differences are another considerable challenge in general. For example, most languages have a function that prints a text to the standard output, but the way they are called is different. For example, C's *printf* works with its own format string and expects NULL-terminated character arrays, whereas Java's *System.out.println* works with a different format description and takes instances of String objects.

Considering the difficulties of automated conversion, partially automated or interactive tools seem more viable. The tools that attempt to perform truly parallel conversion in practice may have some deficiencies. For example, the tool presented by Sneed in (Sneed, 2011) has a systematic way of creating object-oriented classes, but it can be argued whether the resulting object-oriented model feels natural and resembles what a programmer would have created from scratch.

2.2 Cluster Analysis Methods

Discovering potential candidate classes in source code is often referred to as object identification. A commonly used approach in identifying objects from procedural code is cluster analysis (Everitt et al., 2009). This method creates partitions out of the entities (data structures, functions, global variables) present in the code based on the dependencies between them. The goal is to minimize the dependencies between the created partitions and create classes from them. Although most of the solutions that use this approach are largely similar, there exist basic differences as well.

The solution presented by Zou and Kontogiannis (Ying Zou and Kontogiannis, 2001) builds an intermediate XML representation from the AST-s (Abstract Syntax Tree) and then uses an incremental algorithm to divide the entities of the algorithm to different clusters. The approach used by Czibula and Czibula (Czibula and Czibula, 2011) is similar to this but uses a different, heuristic partitioning algorithm created by the authors. The algorithm defines the distance between different entities and, based on this property, creates clusters that have very minimal distances between each other. The algorithm is tailored to provide better performance against particularly large software.

In conclusion, these approaches are similar to our object identification algorithm because they try to find functionally related parts of the application to consider them for class creation. Our method, however, focuses not only on the dependency relation between the entities, but also on using parameter and return type analysis, as proposed by (Ying Zou and Kontogiannis, 2001). This makes our approach identify more classes than regular cluster analysis. Our solution also proposes a novel method of easing the automatic translation of standard function library calls (Section 3.3).

2.3 Code Modernization and Reengineering Tools

Due to the importance of modernizing legacy software, much effort has been made to create tools capable of alleviating this process. However, these tools usually have vastly different goals and focuses.

For example, the tool developed by mtSystems (mtSystems, 2020) is capable of precise, fast and fully automatic syntactical translation from C to Java code, but the created code has two considerable disadvantages. Firstly, when translating pointers into Java, this tool uses different generated container classes to emulate pointer functionality, which makes the generated code feel unnatural. Secondly, the tool does not deal with shifting paradigm from procedural to object-oriented design. We believe that using model transformations to create an object-oriented design is key to making the generated code feel as natural as possible, thus our tool focuses on this aspect.

MoDisco (Bruneliere et al., 2014) is an extendable, model transformation based reverse engineering framework. The process of the transformation consists of 3 important steps. First, the source code is parsed and a higher level semantic model is created that implements OMG's standard of KDM (Knowledge Discovery Metamodel) (Pérez-Castillo et al., 2011). In the next step, various model transformations may be defined and ran on the model that refine and improve it step by step. MoDisco reuses ATL (ATL, 2008) to define these model transformations. ATL is a model transformation language and framework. The final step of the transformation process is code generation based on the model. It should be noted that MoDisco uses the Ecore-based modeling ecosystem of Eclipse as well. It is what has inspired us to use it in our paradigm shift-focused code renovation tool.

As mentioned before, code modernization is a broad term and tools may focus on different aspects of the process. Our goal is to focus on the actual paradigm shift between procedural languages and object-oriented design. Upon further inspection of the MoDisco framework, we believe it is best used in two different scenarios, none of which suit our needs well. Firstly, it functions well as a refactoring tool. Secondly, the tool focuses on architecture modernization. It supports modernization with regard to various technologies, for example JEE (Java Enterprise Edition) and JUnit. Furthermore, another reason we believe the tool is not optimal for our goals is because the various object identification algorithms are way too complex to be described using ATL.

3 TRANSFORMING PROCEDURAL CODE

3.1 Transformation Methodology

Our method of transforming procedural code into an object-oriented design shares ideas with the traditional approach employed in many earlier solutions. This includes parsing the source code, building a model and generating code from it. However, in certain areas, we take new directions.

Using the ASTs (Abstract Syntax Trees), a much higher-level representation of the source code is built in the form of a semantic model, similar to the approach used by MoDisco. It preserves the semantics and thus the intent of the original code but eliminates all of the language-dependent details. OOGen is then subjected to various model transformations. M2M (Model-to-Model) transformations (Langer et al., 2010) are responsible for identifying the potential objects of the system, and M2T (Model-to-Text) (Oldevik et al., 2005) transformations are used to generate code based on the model.

Figure 1 illustrates the architectural view of our solution. As mentioned before, using the Ecore-based modeling ecosystem of Eclipse enables the tool to be compatible with most other model-driven tools that use or expect models that adhere to Ecore. This makes the transformer easy to integrate with such tools, which allows us to reuse them in our solution. This reduces development costs and may further increase the capabilities of a code modernization tool. One concrete tool is Xtext, which we used to develop the DSL that describes transformations of standard library functions (Section 3.3). In the future, more and more such tools may be reused with the transformer.

It should be noted that the concept of the transformation process itself is language-independent. Nevertheless, there are some parts of the transformation that are dependent on the source language or the target language, yet these components can be easily replaced to support a wider range of source and target languages. Parsing the software requires a parser. Similarly, code generation is dependent on the target language, since the code generator must be capable of creating code in the implied language.

3.2 Object Identification: Model Transformations

The quality of the created object-oriented design largely depends on how well potential objects are identified. Thus, it is an essential part of the trans-

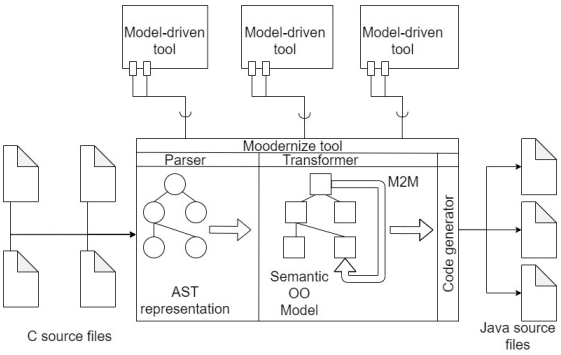

Figure 1: The overall architecture of our solution.

formation process. In our solution, object identification steps are implemented as model transformations. Based on existing literature, we propose the following combination of object identification techniques.

3.2.1 Structure Collection

A basic first step in transforming procedural code into an object-oriented design is to collect all data structures by traversing the AST-s and create a corresponding class for each of them. Each class encapsulates all the data members of the structure as private members and generates appropriate public getter and setter methods for accessing these fields. These classes form the basis of the application.

3.2.2 Parameter and Return Type Analysis

In this step, global functions are analyzed with the intent of assigning them to the classes created in the previous step. To this end, the parameters and the return type of every global function should be taken into consideration. This approach was first proposed in (Ying Zou and Kontogiannis, 2001).

A structure-type parameter implicates that the function is dependent on the implied type. Since the function uses this type for some reason or the other, it could be beneficial to assign this method to the class that corresponds with the type of the parameter. A structure-type return value may imply that the function either updates the value of a variable of the implied type or creates and returns a new instance of it. Either way, it is expedient to assign the function to the class that corresponds with the type of the return value. It should also be noted that return type analysis is a stronger sign of cohesion and thus should take priority over the results of parameter analysis.

3.2.3 Call Chain Analysis

In the third step of object identification, we propose the following method of finding candidate classes. This idea expands on the concept of cluster analysis. We analyze every global function that has not been assigned to a class in the previous step. The goal of this model transformation step is to further refine the remaining global functions by encapsulating functions that may semantically belong together.

To this end, a CG (call graph) is built that represents the inter-procedural control flow of the application. The call graph is a directed graph where nodes represent functions, and an edge from vertex f to vertex g implies that function f calls function g at least once. Our algorithm seeks call chains and creates a class for each of these chains that meets two criteria. Firstly, each and every function in the chain should be called relatively rarely by the other functions in the application that are not part of the analyzed chain. We propose this value to be no more than 5% of the number of all the other functions outside of the current call chain. This requirement (maximum allowed inbound calls) ensures that the created class is coupled loosely with the rest of the application. Secondly, the number of methods in the created class must exceed a predefined safety minimum value. We propose this to be minimum of 3 functions in a class. We believe that 2 functions together may not be a strong sign of cohesion, whereas at least 3 likely yield classes that actually encapsulate functions that logically belong together.

We define our algorithm that traverses a call graph and creates classes encapsulating methods in call chains that satisfy the requirements mentioned above in Algorithm 1. Let $maxf_{in}$ denote the maximum allowed inbound calls and F the set of all remaining functions to analyze.

3.3 A Novel Approach of Transforming Standard Library Calls

After object identification, another important aspect of modernizing between languages is the automatic handling of standard library function calls. Transforming them is an important part of modernizing legacy software. However, doing so automatically is a complex and very non-trivial problem. So much so that even well-developed tools struggle with these issues.

We propose a novel method of easing the automatic conversion between standard library calls of different languages. We have created a DSL (Domain Specific Language) (Fowler, 2010) designed specifically for describing such transformations. We have created the language using the Xtext (Bettini, 2016) framework of Eclipse. When designing the language, we have paid attention to making it as

Algorithm 1: Call chain analysis.

```
Input: CG = (V(CG), E(CG)) call graph
Output: C = {c_1, c_2, ..., c_n} set of
        created classes
Function CreateClasses():
    N = NumberOfAllFunctions()
    max_fin = CalcMaxfin(N)
    F = V(CG)
    C = {}
    for ∀f ∈ F do
        c = Analyze(F, f, {})
        if |c| ≥ 3 then
            C = C ∪ c
        else
            for ∀g ∈ c do
                F = F ∪ g
    return C
Function Analyze(F, f, c):
    F = F \f
    f_in = NumberOfInboundCalls(f)
    if f_in > max_fin then
        return {}
    c = c ∪ f
    for ∀g ∈ ChildrenOf(f) do
        if g ∈ F then
            c = Analyze(F, g, c)
    return c
```

generic as possible. This makes the approach language independent as long as the source or target language is either procedural or object-oriented in nature. The grammar of the language[1] is publicly available on GitHub. It is defined in the file "hu.bme.aut.apitransform.ApiTransform.xtext" in the project "hu.bme.aut.apitransform".

Let us now take a look at an example to illustrate the feasibility of our approach. Listing 1 depicts the handling of fopen and Listing 2 shows how to close the stream when encountering an fclose call.

These scripts can be used to ease the refactoring of basic C file handling. The first one describes that for every fopen call encountered per scope basic Java file handling instances should be created (Filereader and Bufferedreader) with the appropriate constructor parameters. The second script says that when the corresponding fclose call is found, the stream must be closed in the generated code as well by calling the close method of the FileReader instance.

[1] https://github.com/gaborbsd/Moodernize

Listing 1: Instantiations for fopen.

```
transformation {
 source function fopen {
  parameters: path mode
 }
 targets {
  instantiation {
   instance in {
    className: java.io.FileReader
    parameters: path
   }
   instance br {
       className: java.io.
                  BufferedReader
    parameters: in
   }
  }
 }
}
```

Listing 2: Closing the stream at fclose.

```
transformation {
    source function fclose {
     parameters: file
    }
    targets {
     function close {
      parameters:
      owner: in
     }
    }
}
```

4 EXPERIMENTAL EVALUATION

In this section, we present the feasibility evaluation of our approach and the tool on a number of open-source software publicly available on GitHub. Apart from "CTestProject", which is a hand-crafted C project used to quickly test new features introduced to the tool, all these applications are real, existing C projects. Most of these programs are really complex and consist of considerable amounts of source code. Table 1 depicts the software used during the evaluating transformations.

Most of these programs are very low level and have dependencies that were not available during compilation. Some of them may be compiled on specific operating systems only. Thus, the transformations of these programs introduce considerably more errors than they would if all dependencies were available. Regardless, the reason we have chosen to use these applications as test subjects is that open-source complex software written in C is hard to come by. This way, we can be certain that the transformation completes without any runtime exceptions even on software that consist of hundreds of thousands of source code. We also get a glimpse on how these transformations work on real, complex software.

Table 1: The transformed test applications and their total sum of LoC (Lines of Code) across all header and source files.

Availability	LoC
https://github.com/dajobe/flickcurl	33k
https://github.com/pbatard/rufus	86k
https://github.com/hashcat/hashcat	176k
https://github.com/curl/curl	179k
https://github.com/git/git	229k
https://github.com/obsproject/obs-studio	240k
https://github.com/vim/vim	419k

Listing 3: Sample methods of the class dict.

```
1  public void redis_add(int times)
2  public void redis_fetch(int times)
3  public void redis_del(int times)
4  public dict dictCreate
5  (dictType type,  Object privDataPtr)
6  public int dictResize()
7  public int dictGenericDelete
8  (Object key, int nofree)
```

https://github.com/NorbertSomogyi/MoodernizeExamples

The components of the tool[2], the OOGen[3] metamodel and the transformed projects[4] that the tool produced are all publicly available on GitHub.

The core part of our solution was the use of various model transformation steps to create an object identification algorithm that creates various classes from the procedural source code. In the next few subsections, we evaluate the results of our transformations.

4.1 Parameter and Return Type Analysis

The goal of this step is to assign responsibility to classes based on the signature of the global functions. In many cases, this step successfully added methods to created classes. Take for instance the class ”dict”, created in the CBufferedTree, project which implements a dictionary in the form of a linked list in the original C code. This class gained several methods, some samples of which are depicted in listing 3.

As we can see, these methods truly do belong in this class. They are responsible for adding, fetching, deleting entries, creating a certain type of dictionary or resizing it. Similarly, many other classes (e.g. dictentry, container, node) were assigned several methods that would have been placed in them even when creating the design manually by scratch. Based on these results, we conclude that this step of the algorithm truly does create useful object-oriented design.

Listing 4: Sample class created by call chain analysis.

```
1  public static void _exit_
2  localization(Object reinit) {
3      ...
4      free_dialog_list();
5      mtab_destroy(reinit);
6      ...
7  }
8  private static void free_dialog_
9  list() {...}
10 private static void mtab_destroy
11 (Object reinit) {...}
```

4.2 Call Chain Analysis

This transformation step is responsible for further refining those global functions that were not assigned to a class in the previous analysis. It creates classes encapsulating methods that logically belong together. Across all the transformed programs, several call chain classes were created. These classes are named in the following format: $< FirstFunctionName > To < LastFunctionName >$. Listing 4 shows an example taken from the transformed Rufus project, where a total of 16 call chain classes were successfully created.

We can see that one of the methods is public, the others are private. This means that the private methods are never called by any other part of the application. Moreover, the public method actually calls the other 2 private methods at some point. This implies these classes truly may belong together semantically. In other classes, similar results can be observed. To sum up, what we accomplish with this transformation step is that instead of putting every ”leftover” global function in the global ModernizedCProgram class, many of these functions are at least encapsulated with other functions that logically belong together.

4.3 Standard Library Calls

To further elaborate on the examples presented in Section 3.3, let us now take a look at how exactly the the scripts depicted in listing 1 and listing 2 are used to actually translate C file handling snippets. These scripts describe how to handle the C fopen and fclose calls. Coupled with two basic scripts that replace the getC call with readLine (creating a String instance before the calls) and putchar with System.out.print, these can be used to actually translate a simple file

[2]https://github.com/gaborbsd/Moodernize

[3]https://github.com/gaborbsd/OOGen

[4]https://github.com/NorbertSomogyi/MoodernizeExamples

Listing 5: Java file handling snippet.

```
java.io.FileReader in =
    new FileReader("asd.txt");
java.io.BufferedReader br =
    new BufferedReader(in);
String line;
while ((line = br.readLine())
        != null) {
 System.out.print(line);
}
in.close();
```

handling code to its Java equivalent. Using the aforementioned scripts, the Java code shown in listing 5 is generated as a sample Java file handling snippet.

Based on these results, we believe this approach is useful in easing the translation of not-too-complicated code samples. However, to faithfully translate more complex code, further improving the language might be necessary. Particularly, detecting code patterns instead of matching singular function calls could potentially improve the expressive power of our DSL.

4.4 Critical Analysis

In this section, we summarize the strengths and weaknesses of our approach and tool and compare them to the other tools mentioned in this paper.

One of the main strengths of our approach is the use of Eclipse's Ecore-based metamodeling ecosystem. This is similar to what the Modisco tool uses and employing it in a dedicated code modernization tool is a novel approach of tackling this problem. Naturally, our OOGen metamodel is not as detailed as Modisco's metamodel that adheres to OMG's KDM specification. For example, KDM has a separate layer dedicated to the architecture of the modernized program. This is not touched upon in our approach, because for now we have chosen to focus on the model transformations that identify classes in the system. The reason we have not reused Modisco's KDM compatible metamodel is that we had OOGen available from a different project and it was easy and fast to use it. In the future, reusing the MoDisco metamodel will be beneficial to our solution. To sum it up, although Modisco's metamodel is currently more advanced than ours, it is still a considerable advantage against any other tool or approach formerly presented in this paper.

The next notable advantage of our approach is the use of model transformations. These are responsible for identifying potential objects as candidates for class creation. This is an advantage when compared to e.g. mtSystems, for this otherwise state-of-the-art tool does not use any object-identification algorithms proposed in former literature. When compared to Modisco, we believe that these algorithms are too complex to be described using ATL, and have deemed it necessary to implement them "manually". On the other hand, some earlier solutions use more steps in their object identification algorithm, opposed to the 3 steps that we currently use. In the future, adding more steps will be useful to create even better object-oriented design.

We have also proposed the novel approach of call chain analysis. It is similar to the previous cluster analysis approaches seen in Section 2.2. It analyzes the dependency between entities of the system and creates classes that minimize dependencies between the created classes. However, call chain analysis only uses the functions of the system and not the global variables or data structures. Thus, it is used only as a refining step in the end instead of being the main algorithm that creates the object-oriented design. The advantage of this is that we can combine dependency analysis with other forms of object identification, as seen in this paper. The disadvantage is that it does not consider data and classes should usually be used to encapsulate behaviour with the data it operates on. Thus, this step may further be improved in the future by trying to combine the classes created by call chain analysis with global variables that the functions may heavily use.

Finally, our solution also uses an approach of easing the translation of standard library calls. Not only is this a novel approach of tackling this problem, but neither existing literature nor state-of-the-art tools have made an effort to at least partially automate this problem. Needless to say, we believe this is a considerable advantage in favor of our solution.

On the other hand, there exist a number of C language constructs that our reference implementation tool does not handle properly as of now.The mtSystems tool and other similar cutting-edge tools excel at syntactic translation, leaving little to none compilation errors in translated projects.

5 CONCLUSION

In this paper, we have presented our work in progress code renovation tool that transforms procedural C code to Java code in a semi-automatic way. Our approach relies on creating a high-level semantic model out of the AST representation of the source code and using a series of model transformations to create a more object-oriented design. Our object identification algorithm reuses some ideas proposed in existing literature and proposes the new method of call chain analysis. The quality of the identified classes is sub-

ject to the parametrization of the algorithm. In general, the classes are highly cohesive, and they encapsulate data that logically belong to each other.

One of the main benefits of our solution is using the Eclipse EMF modeling ecosystem. This allows us to easily integrate our solution with existing MDE tools, which highly improve the capabilities of the code modernizing tool. We have also created a DSL that is used to effectively describe transformations of standard library function calls. This is a novel approach of dealing with this problem.

We have also experimentally evaluated the feasibility of our solution on various real, open-source software. We have found that our object identification methodology produces useful object-oriented design. We have also compared our solution to existing state-of-the-art tools and solutions presented in existing literature. We have determined that the novelty of our solution is the Ecore-based model-driven approach, the proposed call chain analysis method, and the DSL used for translating standard library calls.

There exist many promising aspects that could be explored upon in future work. Formal proof of the correctness of the transformation should be given in the near future. To this end, we plan to use formal verification methods to ascertain that the transformed program behaves equivalently to the original software. Also, integrating our solution with more existing model-driven tools remains our main goal. For example, instead of manually building a call graph, Eclipse PTP (Watson and DeBardeleben, 2006) may be used to easily and effectively generate it.

ACKNOWLEDGEMENTS

Project no. FIEK_16-1-2016-0007 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Centre for Higher Education and Industrial Cooperation - Research infrastructure development (FIEK_16) funding scheme.

REFERENCES

ATL (2008). ATL presentation. https://www.eclipsecon.org/2008/sub/attachments/Modeltomodel_Transformation_with_ATL.pdf. Accessed: 2020-10-05.

Bettini, L. (2016). *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition.

Bruneliere, H., Cabot, J., Dupé, G., and Madiot, F. (2014). MoDisco: a Model Driven Reverse Engineering Framework. *Information and Software Technology*, 56(8):1012–1032.

Czibula, I. G. and Czibula, G. (2011). Unsupervised transformation of procedural programs to object-oriented design. In *Acta Universitatis Apulensis*.

Everitt, B. S., Landau, S., and Leese, M. (2009). *Cluster Analysis*. Wiley Publishing, 4th edition.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Izurieta, C. and Bieman, J. (2013). A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal*, 21.

Langer, P., Wimmer, M., and Kappel, G. (2010). Model-to-model transformations by demonstration. In Tratt, L. and Gogolla, M., editors, *Theory and Practice of Model Transformations*, pages 153–167, Berlin, Heidelberg. Springer Berlin Heidelberg.

McAffer, J., Lemieux, J.-M., and Aniszczyk, C. (2010). *Eclipse Rich Client Platform*. Addison-Wesley Professional, 2nd edition.

mtSystems (2020). mtSystems documentation. https://www.mtsystems.com/. Accessed: 2020-10-05.

Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., and Berre, A.-J. (2005). Toward standardised model to text transformations. In Hartman, A. and Kreische, D., editors, *Model Driven Architecture – Foundations and Applications*, pages 239–253, Berlin, Heidelberg. Springer Berlin Heidelberg.

Pérez-Castillo, R., Guzmán, I., and Piattini, M. (2011). Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33:519–532.

Sneed, H. M. (2011). Migrating pl/i code to java. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 287–296.

Terekhov, A. and Verhoef, C. (2000). The realities of language conversions. *IEEE Software*, 17(6):111–124.

Watson, G. and DeBardeleben, N. (2006). Developing scientific applications using eclipse. *Computing in Science & Engineering*, 8:50 – 61.

Ying Zou and Kontogiannis, K. (2001). A framework for migrating procedural code to object-oriented platforms. In *Proceedings Eighth Asia-Pacific Software Engineering Conference*, pages 390–399.