Use of Compiler Intermediate Representation for Reverse Engineering: A Case Study for GCC Compiler and UML Activity Diagram

Rania Mzid[1,3] [a], Asma Charfi[2] and Najmeddine Etteyeb[1]

[1]*ISI, University Tunis-El Manar, 2 Rue Abourraihan Al Bayrouni, Ariana, Tunisia*
[2]*Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France*
[3]*CES Lab ENIS, University of Sfax, B.P:w.3, Sfax, Tunisia*

Abstract: Nowadays systems are no longer made from scratch, they use existing third-party components or legacy software. Providing methods/techniques to facilitate the comprehension of existing software is beneficial to increase productivity, especially when dealing with their reuse and/or modernization. Model Driven Engineering (MDE) offers a set of guidelines to manage the complexity of software systems during their development. In that context, the reverse-engineering process aims to describe a source code at higher level of abstraction using automatic transformations. This paper proposes an extensible MDE approach for behavioural reverse engineering. The proposed approach aims to make the reverse transformation independent of the source programming language. Starting from a given source code written in any programming language, the proposed approach integrates an intermediate step based on compiler's front-end to generate an intermediate representation. Then, it performs a model transformation to extract behavioural aspects from the source code and generates a graph that describes its control flow. The different steps of the approach are automated. We apply the approach to case study using GCC and GIMPLE as intermediate representation and UML activity diagram as control flow graph to show its viability.

1 INTRODUCTION

Embedded systems are commonly used in many fields such as automotive, avionics, telecommunications, medical and consumer electronics. These applications are providing multiple sophisticated features which are customized to meet the user's needs diversity. As a result, the software development process for the embedded systems may be both time and effort consuming. Difficulties caused by the development task were raised since the beginning of the software crisis. Several projects have faced catastrophic failures due to over-budgets, over-time, product non-conformity or even unachievable projects (Glass, 2006) (Jørgensen and Moløkken-Østvold, 2006). Moreover, the exponential evolution of technologies, programming languages and platforms used to develop such systems are making the development task more and more challenging. Model Driven Engineering (MDE) (Favre, 2004) (Bézivin, 2005) (Atkin-

son and Kuhne, 2003) (Seidewitz, 2003) proposes solutions to enhance the productivity during the development process of software embedded systems. MDE promotes a rise in level of abstraction, to manage the increasing complexity, by introducing the use of models at the different development stages from the specification to the implementation.

MDE defines two main processes : forward engineering and reverse engineering (Raibulet et al., 2017) (Nelson, 2005). Forward engineering aims to generate executable code from high level descriptions of the system. Reverse engineering can be defined as the process of understanding software and producing models describing it at a higher level of abstraction. However, reverse engineering can be used in different contexts. In (Martinez et al., 2013), four types of reverse engineering are discussed. This classification is based on the level of impact on the existing software: (i) Re-documentation which involves the creation or revision of system documentation, (ii) Design recovery which consists on creating a model or any formal description of the system at a higher level of

---

[a] https://orcid.org/0000-0002-3086-370X

211

abstraction, (iii) Restructuring that is a lateral transformation of the system within the same level of abstraction. Also maintains same level of functionality and semantic and (iv) Re-engineering which involves a combination of reverse engineering for comprehension, and a reapplication of forward engineering to re-examine which functionalities need to be retained, deleted or added.

The most common use of reverse engineering is design recovery which aims to handle the complexity of an existing source code throw model generation. Indeed, representing a source code at higher level of abstraction have several advantages : (i) provides the various stakeholders with common point of view to the project using human readable artefacts. This could improve the coordination between them, even if they have no technical experience, resulting in a faster progress and a more robust software solution. (ii) allows running simulations on reverse engineered models to test critical systems before deploying them (Lima et al., 2020) (Eshuis, 2006) (Banti et al., 2011) (Ouchani et al., 2014) (iii) accelerates program comprehension which in turn facilitate software reuse and legacy software modernization (Martinez et al., 2013). The automation of models generation from source code could accelerate program comprehension and thus increase productivity during the development of software systems. However, with the wide variety of programming languages, enabling automatic transformation is very challenging. In fact, for each considered programming language a set of artefacts must be considered: (i) the meta-model of the language, (ii) the transformation rules and (iii) the meta-model of the target high level representation. The definition of these artifacts requires a lot of efforts and is time consuming.

To tackle this problem, we propose in this paper a reverse engineering approach based on MDE principles. This approach aims to reduce the effort and the time required to perform the reverse engineering transformation from source code to models when different programming languages are considered as inputs. The proposed approach defines two main steps : 1) in the first step, we generate an intermediate representation from the program code to be independent from the source programming language, and 2) the second step aims to produce high level models from the intermediate representation. In this paper, we focus especially on behavioural models that describe at high level the control flow in the code. The passage between the steps are automatically done. The proposed approach is generic and extensible to support different source programming languages while maintaining reasonable time to perform the reverse task.

The originality of this research is manifested by four aspects :

- The proposed approach aims to extract behavioural aspects from an existing program code and produce its control flow graph which could facilitate program comprehension,

- The approach integrates an intermediate step based on compiler intermediate representation in order to make the reverse transformation independent of the source programming language, which may accelerate the reverse engineering process especially when dealing with existing software written in different programming languages,

- We develop a tool which generates automatically control flow graphs from a given source code. The tool uses GCC front-ends and GIMPLE (Pop, 2006)(Merrill, 2003) as intermediate representation and generates Unified Modeling Language (UML) activity diagrams (Cook et al., 2017),

- We apply the contribution to an IoT case study.

This paper is organized as follows. Section 2 discusses the related work. The proposed approach is detailed in Section 3. In Section 4, we describe the tooling support we provide. In Section 5, we apply the proposed approach to a case study to show its applicability and section 6 concludes this paper and outlines some future directions.

2 RELATED WORK

The related work in this paper is twofold : (1) the work dealing with the reverse engineering problem and (2) the work combining compiler techniques and model driven engineering.

2.1 Reverse Engineering Approaches

Many works in the literature deal with the problem of reverse engineering. The paper (Korshunova et al., 2006) proposes a tool called CPP2XMI to be part of a tool chain for software analysis called SQuADT. The tool aims to reverse engineer UML class, sequence and activity diagrams from C++ source code. The work in this paper adopts the vision of OMG standard called Architecture Driven Modernization (Martinez et al., 2013) to automate the reverse transformation for a specific purpose. In (Bruneliere et al., 2014), the authors propose an open source project called MoDisco for model-driven reverse engineering. The main objective of this work is to provide automated framework for the understanding, documentation, modernization, and quality assurance of

legacy systems. The authors in (Bergmayr et al., 2016), propose an open extensible framework for reverse engineering of executable behaviours from existing software codes. It provides fUML behavioural models discovery from JAVA code. This work uses the MoDisco (Bruneliere et al., 2014) framework to perform the reverse transformation. Although the importance of these works, none of them propose solutions to manage the complexity of the reverse engineering transformation when dealing with source code written in different programming languages. In (Kienle and Müller, 2010), the authors consider this problem and propose a reverse engineering environment called Rigi. Rigi is capable to reverse engineer software systems written in different programming languages. The Rigi architecture defines two main modules : the graph editor (used to visualize the software entities and their dependencies) and the extractor. The proposed environment offers the language independent exchange format to decouple extractors from the graph editor. Rigi defines extractors for C and COBOL which are parsers built with the help of Yacc parser generator and store extracted information in the textual exchange format known as RSF (Rigi Standard Format). For the reverse engineering of software systems written in other languages, users are expected to produce RSF files, so that the extraction is not language-independent.Like our approach, this work deals with source codes written in different programming languages and propose a generic solution based on a pivot language. However, in our work, we focus also on reducing the number of model transformations when different programming languages are considered as inputs. This objective is ensured by using compiler frontends to generate a common representation and then reduce the number of transformations to perform. Indeed, in this work, we take advantages of the intermediate representation of compilers to decouple the reverse transformation from the source programming language. So, the compiler front-end of the considered language generates the intermediate code to unify the reverse engineering process and accelerate thus models production.

2.2 Compilers Techniques and MDE

Although one can argue that Compilers and MDE are two orthogonal and very different domains, many works try to combine those domains and take benefits one from each other to enhance system developing. In (Charfi et al., 2012), authors propose a UML compiler: it is just another GCC front end. Along with gcc, g++, gcj, gnat, etc. authors propose guml that allows compiling directly UML classes and State

Machines. Technically, this is achieved by developing a model transformation from UML State Machines to GIMPLE. This approach has the advantages to enhance the quality of the binary code produced by GCC in term of foot print. In fact, by bypassing the code generation (Charfi et al., 2010) (from UML to 3rd generation languages), and enhancing some GCC optimizations (such as SSA dead code elimination and RTL blocks merging) (Charfi et al.,), the code produced from guml is more compact than the code produced by g++ running the -Os option. The GIMPLE Intermediate Representation for GCC was beneficial to achieve this performance. Other recent work (Brauckmann et al., 2020) takes also advantage of using compilers' intermediate forms : the CFG as well as the AST. These IRs are enhancing the deep learning models of code. Instead of relying on sequences of words (just like natural languages processing techniques), authors show in (Brauckmann et al., 2020) that relying on compiler IRs permits to outperform state-of-the-art approaches based on token sequences. They succeed then in identifying a more optimal CPU/GPU mapping for OpenCL kernels.In this paper, we also take advantages of using the compiler IR, but not to enhance foot print (Charfi et al.,) nor for deep learning purposes such as (Brauckmann et al., 2020), but for reverse engineering.

3 PROPOSED APPROACH

We propose in this paper a generic solution for reverse engineering that could be easily extended and personalized for any programming language. the proposed approach uses the compilers front-ends to generate the intermediate code representation which is independent of the source programming language. The reverse-engineering transformation (i.e., design recovery) considers as input the intermediate code to unify the next step of the process for any language. Figure 1 presents in details the different steps in the proposed approach. The process consists in two steps : compiling step and model transformation step. The first step, compiling, considers as input the source code written in a given programming language. This step uses the appropriate compiler front-end to generate the intermediate code. The intermediate code is an intermediate representation based on a programming language which is independent of the source one. This intermediate code conforms to a meta-model called *IR meta-model*. The second step is a model transformation step (i.e., reverse engineering) which aims to extract behavioural aspects from the code to facilitate its comprehension. In order to per-

form this transformation, we have to create the target meta-model which depends on the formalism used to represent the control flow (i.e., *CFG meta-model*) and define the set of transformations rules.
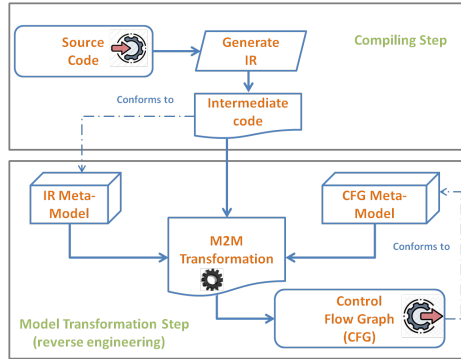

Figure 1: Proposed approach:Detailed steps.

4 TECHNICAL DETAILS AND IMPLEMENTATION

In this section, we present the developed framework. In order to automate the first step of the proposed approach (i.e., compiling step), we choose the GNU Compiler Collection GCC (Pop, 2006) (Merrill, 2003) as compiler. The intermediate representation in GCC is called GIMPLE (Pop, 2006) (Merrill, 2003), which is used in the developed tool to represent the intermediate code. The second step of the proposed approach consists in generating a graph that represents at a higher level of abstraction the control flow in a given source code. For the developed tool, UML activity diagram (Cook et al., 2017) has chosen as a target control flow.

The developed tool is intended to be part of the Eclipse Papyrus Modelling tool (Guermazi et al., 2015) which is an industrial-grade open source Model-Based Software Engineering tool. Indeed, this work should be integrated into Papyrus as a new reverse engineering functionality. To this end, multiple technical decisions have been considered. Figure 2 illustrates the implementation architecture with focuses on the tools and technologies considered to automate each step of the proposed approach.

Technically speaking the proposed approach consists of three main phases : the first phase generates the GIMPLE code using the GCC front-end. Since GIMPLE is an intermediate language, the intermediate code conforms the GIMPLE Grammar. The second phase is the modeling phase which consists in creating Abstract Syntax Tree (AST) Model. The AST model is a graph representation of the intermediate
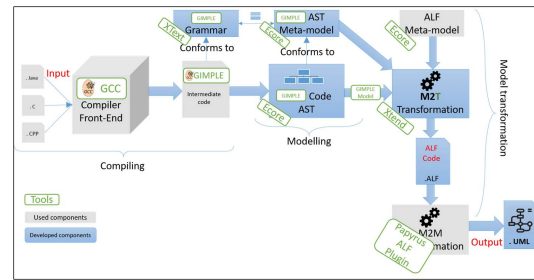

Figure 2: Implementation Architecture.

code semantically equivalent to it and which must conform to the AST meta-model. The third step, is the model transformation step. In this step, we chose to perform a model to text transformation (M2T) from the intermediate AST model to ALF code (OMG, 2011) instead of doing a model-to-model transformation. This choice allows to reuse the already available ALF tool (Seidewitz and Tatibouet, 2015) in Papyrus which performs a transformation from ALF code to UML activity diagram.

4.1 Intermediate Code Generation

The current version of the developed tool uses three of GCC 5.4 front-ends which are gcc, g++ and gcj for the programming languages C, C++ and JAVA, respectively. These front-ends were configured to generate the GIMPLE code. Figure 3 shows an example of a conditional statement "if" using GIMPLE raw syntax. This GIMPLE code was generated by compiling a C++ main function using g++. This code shows how GIMPLE flattens the high abstract statements into lower statements closer to machine instructions. The "if" statement is decomposed into a condition (gimple_cond), three labels (gimple_label) that indicates which instructions to execute in case of a true or false condition and a go to instruction (gimple_goto) to navigate to the next instruction.

At the end of this phase, we obtain a GIMPLE file written in its raw syntax. The file contains intermediate code that is semantically equivalent to source code. This code can be processed in the following steps independently of the original source language.

4.2 AST Model Generation

Since the solution is intended to be part of Eclipse Papyrus, we need to integrate the results from the previous phase to the Eclipse development platform. To ensure this integration we use Eclipse Xtext tool (Eysholdt and Behrens, 2010) which allows loading any GIMPLE file in Eclipse and creating an equiv-

```
main ()
gimple_bind <
  int D.2458;

  gimple_bind <
    int i;
    int k;

    gimple_assign <integer_cst, i, 0, NULL, NULL>
    gimple_cond <eq_expr, i, 5, <D.2455>, <D.2456>>
    gimple_label <<D.2455>>
    gimple_assign <integer_cst, k, 1, NULL, NULL>
    gimple_goto <<D.2457>>
    gimple_label <<D.2456>>
    gimple_assign <integer_cst, k, 0, NULL, NULL>
    gimple_label <<D.2457>>
    gimple_assign <integer_cst, D.2458, 0, NULL, NULL>
    gimple_return <D.2458 NULL>
  >
  gimple_assign <integer_cst, D.2458, 0, NULL, NULL>
  gimple_return <D.2458 NULL>
>
```

Figure 3: Example of a conditional statement "if" using GIMPLE raw syntax.

alent model using its language grammar. We create thus a grammar, using Xtext, for the GIMPLE files generated by GCC. An excerpt of this grammar is given in Figure 4.

```
grammar org.xtext.example.gimpleDSL.Gimple with org.eclipse.xtext.common.Terminals

generate gimple "http://www.xtext.org/example/gimpleDSL/Gimple"

GimpleFile:
  (functions decls+=Function_Decl)*;

Function_Decl:
  {Function_Decl}(modifier+=Modifier)*   (function rtn type=Var_Type? name=Var_Name)?

  main block=Block_stmt
;

/*********
* BLOCK stmt *
*********/
Block_stmt:
  name='gimple_bind' "<" (bind expr vars=Bind_Expr_Var)?   (bind expr body=Bind_Expr_Body)? ">"
;

/*********
* List of possible STMT *
*********/
Simple_Stmt:
  Block_stmt | If_stmt | Switch_stmt | Goto_stmt | Return_stmt | Label_stmt | Call_stmt | Assign_stmt
;
```

Figure 4: Excerpt of the Xtext grammar for GIMPLE.

Then, as shown in Figure 5, a meta-model for the GIMPLE abstract syntax tree (i.e., AST Meta-Model) in Ecore format is generated. Xtext, produces in addition a list of JAVA classes for the AST elements which provides run-time support for the generated instance model from GIMPLE code (i.e., AST model).

```
platform:/resource/org.xtext.example.gimpleDSL/Gimple.ecore
  gimple
    GimpleFile<T>
    Function_Decl
    Simple_Stmt
    Block_stmt -> Simple_Stmt
    If_stmt -> Simple_Stmt
    Goto_stmt -> Simple_Stmt
    Return_stmt -> Simple_Stmt
    Label_stmt -> Simple_Stmt
    Call_stmt -> Simple_Stmt
    Assign_stmt -> Simple_Stmt
    Bind_Expr_Var
    Bind_Expr_Body
    Var_Decl
    Var_Type
    Var_Name
```

Figure 5: Excerpt of the generated GIMPLE Ecore Meta-model.

Figure 5 gives an excerpt of the generated meta-model. It is worth noting that the grammar and the

AST meta-model are created only once. However, we generate an instance model (i.e., AST model) for each GIMPLE file in an Eclipse run-time. This AST model serves as input for the third phase (i.e., the model transformation).

4.3 UML Activity Diagram Generation

In order to generate ALF code from any obtained GIMPLE model instance (i.e., AST model), we perform a model to text transformation using an Eclipse code generator called Xtend (Bettini, 2016). The transformation implements a set of predefined mapping rules between the GIMPLE Meta-model and the ALF meta-model elements. The developed transformation program using Xtend analyses the GIMPLE model (i.e., AST model), executes the transformation rules corresponding to the visited model node, and generates the equivalent ALF code. Once the ALF code is generated, we use the ALF tool (Seidewitz and Tatibouet, 2015) already available in Papyrus which provides an Xtext based ALF code editor and a compiler that executes a Model-to-Model transformation to generate UML activity diagrams from ALF code.

5 CASE STUDY

This section presents the practical application of the proposed tool with a simplified case study from the Papyrus for IoT (Internet of things) Project (Dhouib et al., 2016). Papyrus for IoT uses an IoT model-driven methodology to guide the IoT system designer during the development and supervision of IoT systems. The considered example in this paper is based on a smart IoT-based home automation system (Dhouib et al., 2016). More specifically, we are interested in the temperature management module inside a smart home. Like any IoT system, this module is mainly based on three essential parts, namely, a processing unit, a sensor and an actuator. In this example, the processing unit that takes decisions to manage the temperature of a room, a temperature sensor to provide data from the environment and two actuators, a cooler and a heater, to perform temperature changes in the room.

5.1 Inputs

We have considered as inputs three source codes written in three different programming languages which are C, C++ and JAVA. Figure 6 the C++ source code.The program describes the behaviour of the temperature management unit. The main code function is

executed regularly in the processing unit to retrieve the current temperature from the sensor, compare it to the required temperature set by the user or automatically initialized by the system, then decides to call either the cooler or the heater.

5.2 Gimple Intermediate Representation

Once the input code is retrieved, it is automatically passed through the corresponding compiler front-end to generate the common intermediate code GIMPLE and to print it in a dump file using the raw GIMPLE syntax.

```
12  #include "TmpManager.h"
13
14     TmpSensor tmpsensor ;
15     Heater heater ;
16     Cooler cooler ;
17
18  int main (int current_tmp){
19
20        int tmp_diff = compute_diff(current_tmp);
21
22        if (tmp_diff > 0){
23            cooler.turnOn(tmp_diff);
24        }
25        else{
26            heater.turnOn(tmp_diff);
27        }
28  }
29
```

Figure 6: Simplified C ++ code for Temperature Management Unit.

Figure 7 shows the generated GIMPLE code from the C++ source code given as input. Almost the same GIMPLE is generated from the C and the Java source code. The main structures of a GIMPLE file are functions and code blocks. Each method from source code is transformed into a GIMPLE function with a name, return type, parameters and a code block named "gimple_bind". Each "gimple_bind" forms a local context containing a set of local variable declarations followed by a set of instructions executed either sequentially or in parallel. The instructions could be any GIMPLE statement such as another embedded gimple_bind, gimple_call, gimple_assign, gimple_cond, etc. In order to construct the GIMPLE model instance (i.e., AST model) from the textual GIMPLE code, we use the developed GIMPLE editor with the previously created GIMPLE Meta-model (Figure 5). Figure 8 shows an example of the AST model generated from the C++ code for the considered case study.
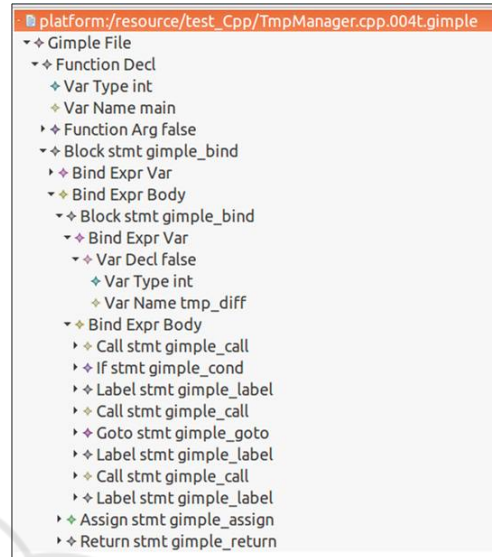

Figure 8: Generated intermediate GIMPLE Ecore model from C++ source file of the considered case study.

5.3 ALF Code

From the AST model, the tool runs the Xtend transformation in order to generate ALF code. Figure 9 shows the intermediate ALF codes generated from the C++ program. It is woth to note here that the ALF textual representation generation is only a technical constraint that we have considered to reuse an already existing compiler developed by the Papyrus Team that takes as input the ALF representation and procudes automatically the UML Activity model. One can argue that developing a compiler from GIMPLE to UML is an alternative way to generate the UML Activity representation. we have choose the first alternative to reuse as much as possible the existing ALF compiler.

5.4 UML Activity Diagram

The last step is the generation of the UML activity diagram from the generated ALF code using the already implemented ALF tool. The model explorer view and the graphical of the UML activity diagram view produced from the C++ source file are given respectively in Figure 10. The activity model describes at high level the control flow in the source file. It is worth mentioning that the intermediate results (i.e., Gimple code, Gimple instance model (AST model) and ALF

```
1
2  int main(int) (int current_tmp)
3  gimple_bind <
4      int D.2522;
5
6      gimple_bind <
7          int tmp_diff;
8
9          gimple_call <compute_diff, tmp_diff, current_tmp>
10         gimple_cond <gt_expr, tmp_diff, 0, <D.2519>, <D.2520>>
11         gimple_label <<D.2519>>
12         gimple_call <turnOn, NULL, &cooler, tmp_diff>
13         gimple_goto <<D.2521>>
14         gimple_label <<D.2520>>
15         gimple_call <turnOn, NULL, &heater, tmp_diff>
16         gimple_label <<D.2521>>
17      >
18      gimple_assign <integer_cst, D.2522, 0, NULL, NULL>
19      gimple_return <D.2522 NULL>
20  >
21
```

Figure 7: Generated GIMPLE code from C ++ code (Figure 6).

```
1  namespace TmpManager;
2
3  activity main ( in current_tmp: Integer ) : Integer
4  {
5      let  tmp_diff   :    Integer = new (0);
6
7      tmp_diff = compute_diff ( current_tmp )  ;
8      if (tmp_diff  >  0 ){
9          cooler.turnOn ( tmp_diff )  ;
10     }
11     else {
12         heater.turnOn ( tmp_diff )  ;
13     }
14 }
15
```

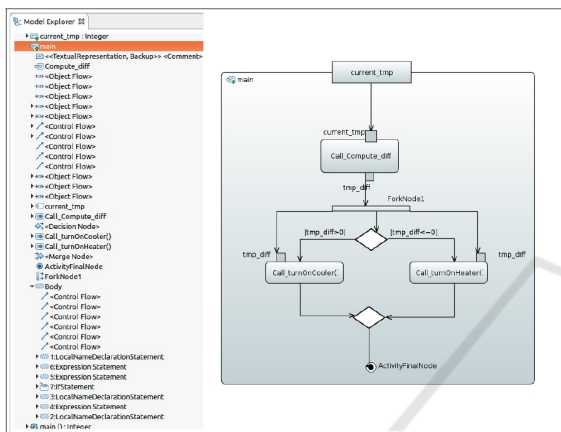Figure 9: Generated ALF code from C++ file.

Figure 10: Generated UML activity diagram from C++ source file for the considered case study.

code) are transparent for the user of the developed tool.

Unlike the textual representation of the code (C,C++,Java) shown in Figure 6 as example, the Figure 10 shows a graphical representation that is easier to understand to evaluate and to exchange mainly with non developer stakeholders. However, the main benefit from having this graphical representation is the graphical simulation of the code. In fact, Papyrus provides UML models execution by means of a module called Moka which aims at providing a generic environment for model execution (Guermazi et al., 2015). Moka natively includes an execution engine for active classes behaviors, represented by UML activities. This Papyrus extension allows the user to simulate its UML Activity diagram by visualizing the code execution at the graphical level. this visualization is very helpful to simulate the execution of the code and to validate it before its deployment that may be an expensive task mainly for embedded systems.

6 CONCLUSION

This paper presents a new approach for reverse engineering of behavioural models from source codes. Firstly, an intermediate representation is generated from an input source code. Here, we use compiler front-ends to produce the intermediate code. Then, a model transformation is performed to generate the graph that describes the control flow in the source code. The contributions in this paper can be summarized in three points : (1) We have proposed the use of compiler in the reverse engineering process in order to make this transformation independent from the source programming language. (2) We have developed a tool that generates UML activity diagrams from source files using GCC and GIMPLE as intermediate representation. (3) We have applied the proposed approach to an IoT case study. The use of the developed tool may facilitate and accelerate the comprehension, the modernization and the reuse of existing code, which may increase the productivity during the development of software embedded systems.

As a possible extension of this work, we plan to enhance the GIMPLE model and the transformation to cover more elements such as parallelism, exceptions,etc, and provide simulations options for the generated activity diagrams. The simulation of an activity diagram have been already tested in a previous work that concern the design and the execution of a robot (the Sybot collaborative robot) (Kchir et al., 2016).Another possible extension is to consider Artificial Intelligence (AI) techniques. Indeed, we aim to automatically generate a database for activity diagrams from online open source projects (GitHub /Gitlab, etc.). This database would serve for automatic transformations using machine learning techniques.

REFERENCES

Atkinson, C. and Kuhne, T. (2003). Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41.

Banti, F., Pugliese, R., and Tiezzi, F. (2011). An accessible verification environment for uml models of services. *Journal of Symbolic Computation*, 46(2):119–149.

Bergmayr, A., Bruneliere, H., Cabot, J., García, J., Mayerhofer, T., and Wimmer, M. (2016). frex: fuml-based reverse engineering of executable behavior for software dynamic analysis. In *2016 IEEE/ACM 8th International Workshop on Modeling in Software Engineering (MiSE)*, pages 20–26. IEEE.

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Bézivin, J. (2005). On the unification power of models. *Software & Systems Modeling*, 4(2):171–188.

Brauckmann, A., Goens, A., Ertel, S., and Castrillon, J. (2020). Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 201–211.

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.

Charfi, A., Boulet, P., and Mraidha, C. (2012). An optimized compilation of uml state machines. In *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 172–179, Los Alamitos, CA, USA. IEEE Computer Society.

Charfi, A., Mraidha, C., Gerard, S., Terrier, F., and Boulet, P. (2010). Toward optimized code generation through model-based optimization. *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 1313–1316.

Charfi, A., Mraidha, C., Gerard, S., Terrier, F., Boulet, P., Inria, H. I., Charfi, A., Mraidha, C., Gérard, S., Yvette, G. S., and Boulet, P. Does code generation promote or prevent optimizations. In *in "Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on", Espagne Parador of*.

Cook, S., Bock, C., Rivett, P., Rutt, T., Seidewitz, E., Selic, B., and Tolbert, D. (2017). Unified modeling language (UML) version 2.5.1. Standard, Object Management Group (OMG).

Dhouib, S., Cuccuru, A., Le Fèvre, F., Li, S., Maggi, B., Paez, I., Rademarcher, A., Rapin, N., Tatibouet, J., Tessier, P., et al. (2016). Papyrus for iot—a modeling solution for iot. *Proceedings l'Internet des Objets (IDO: Nouveaux Défis de l'Internet des Objets: Interaction Homme-Machine et Facteurs Humains. Paris, France*.

Eshuis, R. (2006). Symbolic model checking of uml activity diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):1–38.

Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309.

Favre, J.-M. (2004). Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, WiSME*, pages 262–271. Citeseer.

Glass, R. L. (2006). The standish report: does it really describe a software crisis? *Communications of the ACM*, 49(8):15–16.

Guermazi, S., Tatibouet, J., Cuccuru, A., Dhouib, S., Gérard, S., and Seidewitz, E. (2015). Executable modeling with fuml and alf in papyrus: Tooling and experiments. *strategies*, 11:12.

Jørgensen, M. and Moløkken-Østvold, K. (2006). How large are software cost overruns? a review of the 1994 chaos report. *Information and Software Technology*, 48(4):297–301.

Kchir, S., Dhouib, S., Tatibouet, J., Gradoussoff, B., and Simoes, M. D. S. (2016). Robotml for industrial robots: Design and simulation of manipulation scenarios. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE.

Kienle, H. M. and Müller, H. A. (2010). Rigi—an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75(4):247–263.

Korshunova, E., Petkovic, M., Van Den Brand, M., and Mousavi, M. R. (2006). Cpp2xmi: Reverse engineering of uml class, sequence, and activity diagrams from c++ source code. In *2006 13th Working Conference on Reverse Engineering*, pages 297–298. IEEE.

Lima, L., Tavares, A., and Nogueira, S. C. (2020). A framework for verifying deadlock and nondeterminism in uml activity diagrams based on csp. *Science of Computer Programming*, page 102497.

Martinez, L., Favre, L., and Pereira, C. (2013). Architecture-driven modernization for software reverse engineering technologies. In *Progressions and Innovations in Model-Driven Software Engineering*, pages 288–307. IGI Global.

Merrill, J. (2003). Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179. Citeseer.

Nelson, M. L. (2005). A survey of reverse engineering and program comprehension. *arXiv preprint cs/0503068*.

OMG, P. (2011). Action language for foundational uml (alf).

Ouchani, S., Mohamed, O. A., and Debbabi, M. (2014). A formal verification framework for sysml activity diagrams. *Expert Systems with Applications*, 41(6):2713–2728.

Pop, S. (2006). *La représentation SSA: sémantique, analyses et implémentation dans GCC*. PhD thesis, Paris, ENMP.

Raibulet, C., Fontana, F. A., and Zanoni, M. (2017). Model-driven reverse engineering approaches: A systematic literature review. *IEEE Access*, 5:14516–14542.

Seidewitz, E. (2003). What models mean. *IEEE software*, 20(5):26–32.

Seidewitz, E. and Tatibouet, J. (2015). Tool paper: Combining alf and uml in modeling tools-an example with papyrus-. *OCL@ MoDELS*, 1512:105–119.