Challenges in Reverse Engineering of C++ to UML

Ansgar Radermacher, Marcos Didonet Del Fabro, Shebli Anvar and Frédéric Chateau
Université Paris-Saclay, CEA List, France

Keywords: UML, Reverse Engineering, C++, Tool Support.

Abstract: Model-driven engineering provides several advantages compared to a direct manual implementation of a system. In reverse-engineering applications, an existing code basis needs to be imported into the modeling language. However, there is an abstraction gap between the programming language (C++) and the modeling language, in our case UML. This gap implies that the model obtained via reverse engineering is a model that directly mirrors the object-oriented implementation structures and does not use higher-level modeling mechanisms such as component-based concepts or state-machines. In addition, some concepts of the implementation languages can not be expressed in UML, such as advanced templates. Therefore, new systems are often either developed from scratch or model-driven approaches are not applied. The latter has become more attractive recently, as IDEs offer powerful refactoring mechanisms and AI based code completion - model-driven approaches need to catch up with respect to AI support to remain competitive. We present a set of challenges, based on examples, that need to be handled when reverse engineering C++ code. We describe how we handle them by improving reverse engineering capabilities of an existing tool.

1 INTRODUCTION

Model-driven engineering (MDE) provides several advantages compared to the direct implementation of a system. These include notably the possibility to specify requirements and perform analysis on the model. MDE also offers the option to define high-level behaviors with state-machines which are semantically close to the system specification. Code generation assures that the implementation is well synchronized with the model. Refactoring with architectural impact can be better done on model level, for instance changing inheritance hierarchies. In case of UML (OMG, 2017), the Object Management Group (OMG) standardized the modeling language which fostered its adoption by the industry and the development of several modeling tools.

Reverse engineering is not only interesting for legacy systems. It also plays a role during system evolution. Fig. 1 shows the different steps in round-trip engineering: (1) an initial model representation can be obtained using reverse engineering of legacy code, (2) the generation of code, (3) the evolution of the code, for instance in the context of debugging, (4) the reverse of the model from the code, and (5) evolution of the model. Please note that the reverse from code (4) is quite different from the initial reverse of legacy code (1). First, it needs to update an existing model which implies identifying elements in the source model in order to update them instead of re-creating them. Both (2) and (4) become synchronization tasks. While this aspect makes the reverse engineering task more difficult, it is effectively easier due to more knowledge about the generated code, as we will see in section 2. Note that steps from (2) to (5) are iterative.

However, an existing codebase often hinders the use of a model-driven approach, as reverse engineering mechanisms have some limitations that we will discuss in section 2. These range from issues related to the representation of programming details in a modeling language to the more fundamental problem that higher level concepts offered by the modeling language are hard to detect and the UML model often remains too close to the implementation. We categorize six kinds of challenges that are currently not or only partly handled by related work. We explain the difficulties and then outline how we handle them in a later section. In this case, many advantages of modeling can not be used. We show some of these aspects based on code that we have found in real reverse engineering projects.

The paper is structured as follows. In section 2, we explain the challenges in reverse engineering, no-

Radermacher, A., Fabro, M., Anvar, S. and Chateau, F.
Challenges in Reverse Engineering of C++ to UML.
DOI: 10.5220/0012720200003690

In *Proceedings of the 26th International Conference on Enterprise Information Systems (ICEIS 2024) - Volume 2*, pages 272-279
ISBN: 978-989-758-692-7; ISSN: 2184-4992

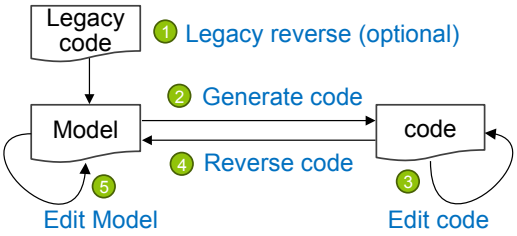

Figure 1: Different phases in round-trip engineering.

tably from non-generated code. Section 3 shows tool support for many of these challenges in our tooling. We outline related work on reverse engineering in section 4, including commercial tools and academic approaches. We present our conclusions in section 5.

2 REVERSE ENGINEERING CHALLENGES

Object-oriented languages are in principle compatible with the modeling concepts offered by UML. However, programming languages have some subtleties that cannot be expressed directly. As UML is an extensible language, a suitable profile may add support for these issues, but this is not always possible. In the following subsections, we expose the typical problems encountered when applying reverse engineering to real-life systems.

2.1 Mapping Between UML and Code Is not Bijective

Mappings from a source to target set have mathematical properties. An injective mapping implies that distinct elements of the source model map to distinct elements of the target model. A surjective mapping denotes that every element in the target model has an associated element in the source model. If both criteria are satisfied, the mapping is bijective.

In general, the transformation from pure UML to code is not surjective: we can find constructs in the source code that have no direct counterpart in UML. To address this issue, UML meta-model elements can be extended via stereotypes as parts of a UML profile. Consequently, the model of the mapping in question will be based on a combination of UML and a language profile. However, this mapping is also not injective, as the same source code could be mapped to different model elements. One example is the use of a directed association between two classes A and B. A typical mapping to code is that class A has a reference (pointer) to class B. If we find such a case in the code, we can either map it to an attribute stereotyped

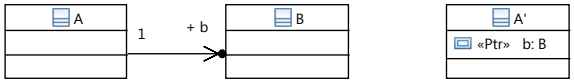

Figure 2: Map C++ pointer to an attribute or an association

as pointer or to the directed association, as shown in diagram 2: The result of reversal could be either class A or class A'.

2.2 C Pre-Processor

The C++ language inherits a very old mechanism from C, a textual pre-processor that expands macro definitions and resolves file includes. A UML profile cannot adequately handle the text replace mechanisms, as these may have arbitrary effects. For instance, several declarations may be only processed optionally (#ifdef) or parts of a name might be modified. Let us look at a relatively complex pre-processor definition and usage that has been found in existing code.

```
1 #define DECLARE_ALIAS_PARAM(suffix) \
2 using CfgParam##suffix=Cfg::Param<Cfg::
      Type::suffix>;
3 ...
4 DECLARE_ALIAS_PARAM(U8)
```

The double hash symbol enables the concatenation of an argument without added white-space. When expanded, the previous definition results in the following declaration.

```
1 using CfgParamU8=Cfg::Param<Cfg::Type::U8
      >;
```

This declaration defines an alias (via the using statement, discussed in subsection 2.3) for a type that is based on an existing template Cfg::Param and an enumeration called Type within class Cfg (the enumeration has among others a literal called U8). It is not clear how such a macro should be represented in the model, nor whether it needs to be represented at all.

The pre-processor may be used at compile time for configuration purposes, including conditional expansion with #ifdef. The reversed model corresponds therefore to a particular variant of the original code and needs to be modified manually in order to generalize it.

2.3 Language Details

The following code serves as an example for several nitty-gritty C++ details. It declares a class template that takes a class called Svc (Service) as a formal parameter. The template class also includes a variadic template method.

```
1  template<class Svc> class ActivableSvc {
2   public:
3    ActivableSvc() = default;
4
5    template<typename... Args>
6    bool activate(Svc& svc, Args&&... args);
7    bool deactivate();
8
9    bool isActive() const;
10
11   using ExecStatus = typename Svc::
       ExecStatus;
12
13   static constexpr ExecStatus
       STATUS_INACTIVE = Svc::STATUS_INACTIVE
       ;
```

The first observation is that this class is a template; we will come back to this aspect further. Line 3 instructs the C++ compiler to define a default constructor even when other constructors are defined. This possibility exists since C++11. UML does not have a means to express this, so we need to rely on UML's extension mechanism via a suitable profile. The declaration in line 5 and 6 has several C++ specific elements that can not be expressed in UML. It declares a template method with two additional aspects: the declaration uses the keyword typename and is variadic as indicated by the three dots. The keyword indicates that Args denotes a type, the variadic declaration that the method can be bound to an arbitrary list of arguments. The method itself (in line 6) has two parameters. The first parameter is passed by reference. The second is a so-called rvalue reference (a feature that has been introduced in C++11, it references a temporary object. Without going into detail, it is often used to implement move semantics). Like the template parameter, it is variadic.

Line 9 contains a method declaration that is constant, implying that it does not change the attributes of the owning class (which is an important hint in the context of code analysis and multi-threading). This information can be modeled using the isQuery property of an operation or via a stereotype.

The using statement in line 11 declares an alias, it imports the name ExecStatus from the Svc class into the ActivableSvc template class. Again, the keyword typename indicates that the imported element is indeed a type (to overcome the template "dependent name" limitations in C++). With respect to UML, alias types are not supported and need again the help of a stereotype. This stereotype needs to reference the imported type. It seems that an attribute in the stereotype pointing towards the imported type is a possible solution. However, in that case, the type is unknown at template declaration time, as it depends on the binding of the Svc template parameter. Another option is to stick to the qualified name, as C++ does. However, this option breaks the modeling philosophy which states that objects are referenced by identity and not by name, implying for instance that a renaming of the Svc parameter would break the reference in the alias declaration (if not automatically renamed by tooling). A pragmatical way out is to use a combination where model reference and textual binding are both possible, but only the latter should be used, if the former is not possible.

Line 13 declares a constant using the C++ keyword constexpr in combination with static. While the latter can be expressed with UML, the former requires a stereotype.

2.4 Higher Level Concepts

The goal of a reverse engineering step is to recover a UML model. Ideally, this model makes use of higher level constructs such as component-based abstractions for structural specifications or state-machines for behavioral ones. Since programming languages do not have these mechanisms, we cannot find them directly in the code. Instead, we find artifacts of an implementation of these concepts in a specific language.

During forward engineering the code generation applies an implementation *pattern* that defines how to *map* higher level concepts to the programming language. In the case of component-based modeling, the pattern denotes for instance how a port should be represented in the programming language. A state-machine requires a more complex pattern. Several variants have been examined and compared in (Pham et al., 2017). The key issue is that applying a pattern during forward engineering is a relatively simple task while the detection of the used pattern is challenging. The main reasons are outlined as follows

(1) The reverse engineering algorithm does not know which implementation pattern has been applied for a specific concept. A simple example is attributes or parameters with a non-1 multiplicity. In the code, we have elements typed with instantiated container classes such as std::list or std::vector. Should a reverse mechanism obtain exactly what it finds or should it replace the template container with the bound type and star multiplicity? This issue is already not trivial for simple cases, it is impossible to recover a state-machine (and not the implementation artifacts) from the code, since there are more than a dozen common implementation variants. The main exception is when the reverse engineering code knows exactly which pattern has been applied.

(2) The artifacts belonging to a single concept are

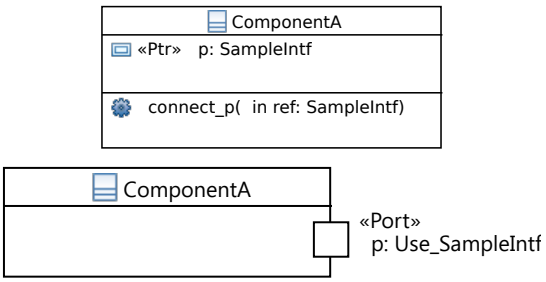

Figure 3: (Upper part) recovery of implementation model, (lower part) higher level model.

typically spread within the code and mixed with manual code or code related to other patterns. For instance, a port enabling access to operations provided by a connected component can be associated with an attribute storing a reference to the interface of the connected component *and* a method to setup this reference, as shown in the following code. A very simple option is to use a pointer to store the reference to an interface, another to instantiate a template.

```
class ComponentA {
  SampleIntf *p; // RPort<SampleIntf> p;
public:
  // initialize port reference
  void connect_p(SampleIntf * ref);
};
```

Obviously, a class diagram of ComponentA is not very helpful for understanding the architecture, while a composite structure diagram is, as shown in Fig. 3.

(3) If the original code has been developed manually (i.e., without code generators), the pattern is likely not applied consistently, even in the same project, notably if implemented by multiple developers or in order to apply optimizations.

This means that, up to now, the extraction of higher level concepts only works in specific cases, notably for reversing code that has originally been generated with annotations about the used design patterns.

2.5 Granularity

A larger system may contain a thousand or more files, each possibly containing several class declarations (exactly one in Java). Such a setup does not scale well enough to be able to store the whole model in a single file, even if database solutions such as CDO (Stepper et al., 2023) may offer a workable solution. The reverse engineering mechanism must therefore find the right level of granularity to store artifacts.

Libraries pose an additional problem. Attributes of application classes and method parameters are often typed with classes from libraries. The question is whether the reverse engineering tool should continue analysing the library code or stop. It can only do the latter, if it creates at least one incomplete type with the right qualified name (i.e., in the right hierarchy of UML packages) to avoid leaving application elements untyped. If we consider rich libraries, such as the classes and interfaces of the java.lang package or the standard template libraries of C++, the modeling tool ideally already provides the libraries and stops reverse engineering at this step. However, providing complete libraries for each version of a certain language is very costly and most tool vendors do not provide them.

3 TOOL SUPPORT

Papyrus (Papyrus-developers, 2024) is an open-source, Eclipse-based UML modeling tool. It has an extension called Papyrus SW designer (Radermacher et al., 2024). This extension comes with a C/C++ profile for UML and supports code generation as well as reverse engineering. During forward engineering, the tool applies the formatter of the Eclipse CDT (CDT-developers, 2024) project. In this way, the code formatting can be adapted to one's particular needs. There are other preferences that are currently not configurable, for instance whether the attributes section should be placed before (default behaviour) or after the methods section; or the preferred order of appearance of the public, protected and private sections of a class (regardless of being properties or operations).

3.1 Bijective Transformations, Recovery of Higher Level Models

In section 2.1, we discussed the problem that a transformation from model to code is not bijective, i.e., it cannot be reversed non-ambiguously. This hinders in particular the recovery of higher level concepts from the code (see section 2.4). In the the following paragraphs, we look at a simple variant of the reverse engineering task: the design recovery from a model that has been generated previously with the same tool.

For CBSE concepts and state-machine code generation, the tool supports the execution of a chain of model to model transformations, as shown in Fig. 4. The objective is to keep the code generator (model-to-text) simple and implement advanced code generation features in separate transformations that can be exchanged without touching other parts of the generator. The transformation realizes an *implementation pattern* of the concept. This idea has been initially pre-

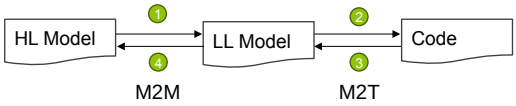

Figure 4: Generating concepts with model to model transformations (HL = high-level, LL = low-level).

sented in (Radermacher et al., 2009) for component-based structures).

For these transformations, (Pham et al., 2018) proposed a technical trick to assure that the mapping from model to code remains bijective: the code generator produces code that has been called *extended* in the sense that it seems to add semantic features to a programming language that support component-based concepts or state-machines in C++. However, these concepts are mainly based on meta-programming via template definitions (and some pre-processor macros), the resulting code can be compiled with C++. In case of CBSE, the example in section 2.4 for representing a port contained two options to map a port to code. In the case where a simple pointer is used to store the reference, the reverse mechanism cannot conclude that the reference belongs to a port. However, if a specific template class (here RPort) is used instead, the reverse is possible given that the implementation pattern is known.

In the sequel, we examine the generated code of a state machine, as shown in the following code fragment.

```
Statemachine FIFOMachine {
 InitialState
 Idle{};
  State SignalChecking {
     StateEntry entryCheck();
     StateExit exitCheck();
  };
  State Discarding {
  };
  State Queuing() {
     StateEntry entryQueue();
  };
  TransitionTable {
    ExT(Idle,SignalChecking,
        DataPushEvent,NULL,signalCheck);
     ExT(SignalChecking,dataChoice,
        NULL,NULL,NULL);
     ExT(dataChoice,Queuing,NULL,valid,
    NULL)
  },
```

Although the code looks like a textual state-machine specification, it is valid C++ code that executes a state-machine. Technically, the execution uses preprocessor macros and C++ templates. This is a major difference to textual modeling languages (TMLs) such as Umple (Badreddin et al., 2014) that provide bi-directional mapping from the TML to code

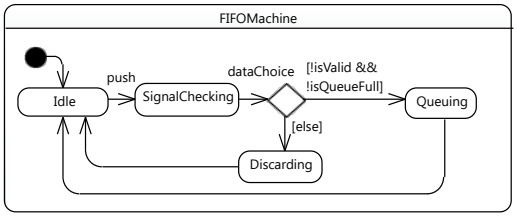

Figure 5: Result of reverse engineering a state-machine.

(today, PlantUML (PlantUML, 2023) is more widely used, but it does not come with a code mapping). In our approach, programmers can use their favorite IDE while TMLs force programmers to change their working environment. In (Maro et al., 2015), the authors integrate graphical and textual editors for UML profiles to allow developers to work in both of the representations.

Due to the declarative character, the recovery of the state-machine is relatively simple. In fig. 5, we have reversed the code of this state-machine (again, the reverse mechanism only covers the semantic model). The Machine contains several states, including a choice states with guards and entry actions. This only works, since the code has initially been generated in a specific way that is known by the reverse engineering mechanism.

3.2 C Pre-Processor

The tool has a specific stereotype «Include» that can be applied to a class. It stores textual elements that should be copied in the generated file. Thus, it can handle macros which are automatically filled by the reverse process, but the mechanism only handles macros that are within specific markers produced by code generation. Thus, macros are currently ignored during legacy reverse. The reverse mechanism operates on an AST with macro expansions, i.e. we get the *using* declaration that we have seen in section 2.2 but not the original macro use (DECLARE_ALIAS_PARAM). In this case, the expansion is not a disadvantage. Otherwise, it would not be possible to represent the alias type in a suitable way on the model level.

3.3 Language Details

We've applied our tool to the ActivableService class shown in section 2.3 and fixed deficiencies with respect to C++11 in an iterative way. The model has been automatically created, the diagram in Fig. 6 has been created manually via drag and drop of the ActivableService from the model explorer to the class diagram.

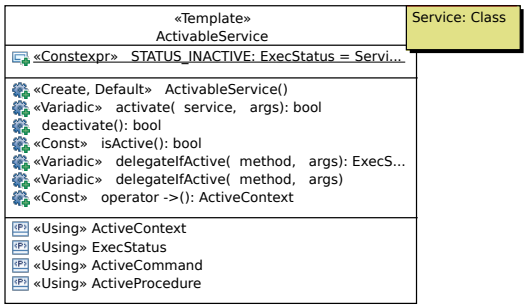

Figure 6: Reverse engineering of class ActivableService.

The application of the C/C++ profile has preserved the programming language details, notably with respect to pointers, (rvalue) references, const and variadic directives. The constructor has a «Create» stereotype from the UML2 standard profile. Alias types have been created as nested classes. Therefore, they appear in the nested classifier compartment.

3.4 Granularity

In order to handle libraries, as discussed in 2.5, the profile contains the stereotypes «External» and «ExternalLibrary» (on the package level) that are applied to empty classes and provide information how to include a specific class or the whole library (include directory setup and linker information). If an analysed class is located outside of the directory/project that is reversed, the mechanism checks first whether the type is already in one of the standard libraries. If not, it creates an empty class which is marked as external. In case of C/C++, a specialization of the stereotype provides information about include directives.

3.5 Efficient Template Binding

Papyrus SW designer provides an additional stereotype «TemplateBinding» that can be applied to typed elements, e.g. attributes or parameters. This stereotype has an attribute that stores a list of (actual) types that bind the formal parameters simply via the order, in a similar way as C++ does. For instance an attribute "vector<string> strList" becomes an attribute strList that is typed with a vector. The stereotype «TemplateBinding» is applied to the strList attribute and references the string class, enabling template bindings without an auxiliary class.

3.6 Performance Evaluation

We have applied the Papyrus C++ reverse engineering mechanism to a larger project, consisting of several source folders and a mix of C++ and C code. The lines of code are indicated in table 1. The process took about 50 seconds, based on an already indexed CDT project. The execution time is thus quite acceptable for a relatively large project.

Table 1: Size of code (number of lines) and obtained model of larger reverse engineering project.

C++ body	C body	header
42221	116647	77976

#model elements	#stereotypes
43646	5634

The reverse mechanisms above are not new, except for the mentioned C++11 enhancements and the support of multiple source folders. Our tool can recover high-level concepts from components and statemachines, but only if it knows the used implementation pattern, i.e. if the code has been generated by the forward engineering part of the tool. This is not the case for legacy code in which we neither know the implementation patterns nor whether a single pattern has been applied rigorously.

4 RELATED WORK

4.1 Commercial Tools

Many commercial tools, notably Enterprise Architect (SparxSystems, 2023a) from Sparx-Systems, Visual Paradigm (Visual Paradigm, 2023) and IBM's Rhapsody (IBM, 2023) support reverse engineering. However, little information about the details of the reverse mechanisms are available. It can import a statemachine from a specific file .sm, but does not support reverse engineering from source code. In case of Enterprise Architect, rather generic information can be found in (SparxSystems, 2023b).

Rhapsody has a quite good support for reverse engineering. On the one hand, it supports the capability to define the mapping of types to external code which avoids the import of elements from standard libraries into code. On the other, it supports preserving the structure / location of the code when re-generating. While this allows to start modeling quickly, it has also a drawback: the location of generated code follows the location within the legacy code and is therefore possibly inconsistent with standard rules. For instance, a rule prescribing that sub-folders should correspond to the package structure might not be respected by the legacy code (we have seen that folder names do not correspond to namespace definitions used by C++ files within these folders).

Table 2: Selected reverse engineering support by commercial and open source tools.

	C Pre-processor	High-level models	granularity
Enterprise architect	pre-define macros	no	select folders
Visual paradigm	pre-define macros	no (import from .sm file)	select folders
Rhapsody	centralized	no	select folders
Papyrus SW designer	associated with class definition	components and SMs, if code previously generated with same tool	CDT project

With respect to macros, Rhapsody collects and stores macros in a specific header file which is part of the model project configuration. Within this header file (that can be modified by the developer), macros are grouped according to the appearance in the original file. The handling is similar for the other two tools which offer the possibility to pre-define macros in a project central way.

All tools use a UML profile to store language specific properties. Unfortunately, the OMG has not standardized a UML profile for C++ implying that each tool vendor has its own variant of the C++ profile.

With respect to granularity, the three commercial tools support the selection of one or more folders. In case of Papyrus, SW designer, the granularity is a CDT project which can contain one or more source folders. Code that is outside of these folders is analyzed in a shallow way: if not defined already in an existing model library, empty primitive types are created. The comparison is summarized in table 2.

4.2 Academic Approaches

There is not much recent activity in the field. Most publications are already a bit older. We think that this is mainly caused by the following factors: (1) reverse engineering only recovers implementation models (as opposed to high level models). This reduces the interest in these models as architectural decisions are not visible or requirements not explicit. (2) code based IDEs have become more and more powerful. IDE refactoring operations take care of identifying and performing changes in all concerned source files. Today, they can execute more complex refactoring operations with architectural impact, such as restructuring the inheritance hierarchy. While it is still preferable to do these operations on the model level, it is technically not necessary any more.

Sutton and Maletic (Sutton and Maletic, 2007) proposed a set of rules for reverse engineering the elements within a class diagram. The paper is quite detailed with respect to the class diagram elements and includes for instance the multiplicity aspects and the question whether to map a C++ class to a UML datatype or UML class (we omitted this example). Since the semantics behind UML data-types is based on data equality, the paper proposes to base that decision on the presence a public default constructor, a copy constructor, and an assignment operator. The rule is quite questionable, as classes might be wrongly classified (the authors admitted this possibility in the paper). While the paper is very complete with respect to the class diagram, it does not cover component-based concepts nor state-machines. The paper also contains information about the reverse engineering tool called *Pilfer*, written in Python. The work around this tool has been discontinued. Sutton also observed that these tools are used less frequently during software maintenance and evolution compared to forward engineering. Although this statement has been done more than 15 years ago, it still probably holds due to the challenges we described.

Tonella (Tonella and Potrich, 2002) analysed not only the class declarations, but also the *instances* of classes that are created statically or dynamically. The former information can be captured by UML instance specifications (object diagrams), but as of today, this analysis is not supported by most tools.

Hafeez-Osman and Chaudron (Hafeez Osman, 2012) analysed the reverse engineering capabilities of eight different UML tools with respect to package, class and sequence diagrams. However, the paper remains at a very high level of abstraction: the reader only gets to know if a tool supports the reverse-engineering mechanisms for a given diagram and if yes, if this support is “good”. The paper is thus not very helpful to get a deeper insight into the topic.

A relatively recent work from Rosca (Rosca and Domingues, 2020) compared round-trip engineering approaches for the UML Class Diagram for three different UML tools (in the context of a hospital management case study. As we have seen in Fig. 1, reverse-engineering is part of round-trip. The paper examines three scenarios, in the first generated source code without changes is reversed again. In the second, reverse is executed after code changes, and in the third both existing model and code are modified before reversal. While the scenarios are promising, the paper

falls short to examine the scenarios in more detail. As the focus is more on methodology, we only get a table that compares the numbers of reversed elements (theoretical vs. obtained) for the three tools without a discussion about programming language specific issues and the correctness of the results.

5 CONCLUSIONS

In this paper, we have shown several challenges to reverse C++ code to UML. These include language details as well as the recovery of high-level models. The latter is currently in general not supported by tools. In our approach, it only works for a rather small subset of projects: the design recovery from code that has been previously generated with our approach, as the implementation patterns are known and carefully chosen during the design of the code generator with respect to a bijective mapping.

For the design recovery of legacy code, information of applied implementation patterns must be provided. We hope that deep learning can provide a means in the future to recover this information, as it should cope with the fuzzy application of implementation patterns by developers. In order to train these mechanisms, we plan to use the code base from the SW heritage project (Software-Heritage, 2023) and the fact that a certain subset of the projects contain both design documents and the associated code allowing us to obtain a training dataset.

ACKNOWLEDGEMENTS

This work has been funded by CEA through the platform Deeplab and by ANR PIA: ANR-20-IDEES-0002.

REFERENCES

Badreddin, O., Lethbridge, T. C., Forward, A., Elasaar, M., and Aljamaan, H. (2014). Enhanced Code Generation from UML Composite State Machines. *Modelsward 2014*, pages 1–11.

CDT-developers (2024). Eclipse c/c++ development tools (cdt). https://projects.eclipse.org/projects/tools.cdt. Oct. 2023.

Hafeez Osman, M. R. C. (2012). Correctness and Completeness of CASE Tools in Reverse Engineering Source Code into UML Model. *GSTF Journal on Computing*, 2(1):193–201.

IBM (2023). IBM Rhapsody. https://www.ibm.com/products/uml-tools. [Online; accessed Oct-2023].

Maro, S., Steghöfer, J.-P., Anjorin, A., Tichy, M., and Gelin, L. (2015). On Integrating Graphical and Textual Editors for a UML Profile Based Domain Specific Language: An Industrial Experience. In *Proceedings of the 2015 ACM SIGPLAN SLE*, pages 1–12. ACM.

OMG (2017). *Unified Modeling Language (OMG UML), Version 2.5.1*. OMG Document formal/2017-12-05.

Papyrus-developers (2024). Eclipse Papyrus. https://eclipse.dev/papyrus/download.html. Oct. 2023.

Pham, V. C., Radermacher, A., Gérard, S., and Li, S. (2017). Complete code generation from UML state machine. In *Proceedings of the 5th MODELSWARD, Porto, Portugal, February*.

Pham, V. C., Radermacher, A., Gérard, S., and Li, S. (2018). A New Approach for Reflection of Code Modifications to Model in Synchronization of Architecture Design Model and Code. In *Proceedings of the 6th MODELSWARD, Funchal, Portugal*.

PlantUML (2023). PlantUML website. https://plantuml.com/. [Online; accessed 10-2023].

Radermacher, A., Cuccuru, A., Gerard, S., and Terrier, F. (2009). Generating Execution Infrastructures for Component-oriented Specifications With a Model Driven Toolchain – A case study for MARTE's GCM and real-time annotation. In *Eighth GPCE'09*, pages 127–136. ACM press.

Radermacher, A. et al. (2024). Papyrus Software Designer. https://wiki.eclipse.org/Papyrus_Software_Designer. Oct. 2023.

Rosca, D. and Domingues, L. (2020). A systematic comparison of roundtrip software engineering approaches applied to UML class diagram. In Cruz-Cunha, M. M. et al., editors, *2020 International Conference on ENTERprise Information Systems*, volume 181, pages 861 – 868.

Software-Heritage (2023). Software Heritage. https://www.softwareheritage.org. [Online; accessed 10-2023].

SparxSystems (2023a). Enterprise Architect. http://www.sparxsystems.com/products/ea/. [Online; accessed Oct-2023].

SparxSystems (2023b). Enterprise Architect, Source code import. https://sparxsystems.com/enterprise_architect_user_guide/14.0/model_domains/notes_on_source_code_import.html. [Online; accessed Oct-2023].

Stepper, E. et al. (2023). Eclipse CDO Model Repository. https://projects.eclipse.org/projects/modeling.emf.cdo. [Online; accessed Oct-2023].

Sutton, A. and Maletic, J. I. (2007). Recovering UML class models from C++: A detailed explanation. *Information and Software Technology*, 49:212–229.

Tonella, A. and Potrich, A. (2002). Static and Dynamic C++ Code Analysis for the Recovery of the Object Diagram. In *Proceedings of the International Conference on Software Maintenance (ICSM'02), Toronto, Canada*. IEEE.

Visual Paradigm (2023). Visual Paradigm Homepage Website. https://www.visual-paradigm.com/. [Online; accessed 10-2023].