Case Studies in Model-Driven Reverse Engineering

André Pascal
LS2N CNRS UMR 6004 - University of Nantes, France

Keywords: Model-Driven Reverse Engineering, Re-engineering, Legacy Systems, Model, Abstraction, Process.

Abstract: Without abstraction, third party maintenance can hardly make evolve (even documented) software applications. In this paper we address the problem of re-engineering software applications to add abstraction in order to improve their continuous evolution. In three different case studies, we make use of Model-Driven Reverse Engineering for extracting component software architecture, for aligning business and application logic in information systems and for re-engineering a holonic software manufacturing process. We report lessons learnt for future developments.

1 INTRODUCTION

The maintenance process of software applications represents more than 70% of the total of software cost and this percentage is still growing up when maintaining old legacy applications (Dehaghani and Hajrahimi, 2013). Abstraction is a key factor to maintain complex software applications because the concepts are more resilient than their implementations. Without abstraction, third party can hardly make evolve software applications due to numerous implementation details, tricky programming, hard-coded parameters, deprecated frameworks, etc. The business application concepts are merged with implementation issues to build an intricate software. The reference manuals are rarely sufficient to get easily into the source code: implementation details are given but the design decisions are not easy to trace. Moreover the specifications are often deprecated against the programs. Due to lack of information, maintenance developers may reinvent the wheel or redo errors. Let call this phenomenon the *abstraction debt*.

We are convinced that Model-Driven Engineering (MDE) is a gainful approach to develop long-term software systems. According to (Selic, 2008), the different MDE paradigms can be reduced to two main ideas: raising the level of abstraction and raising the degree of computer automation. MDE techniques have proven useful not only for developing new software applications but for re-engineering legacy systems (Cuadrado et al., 2014). This paper deals with Model-Driven Reverse Engineering as a means to produce software abstractions.

Reverse engineering is the process of comprehending software and producing a model of it at a high abstraction level, suitable for documentation, maintenance, or re-engineering (Rugaber and Stirewalt, 2004). The goal is to re-introduce abstractions in the legacy applications in order to gain advantage later when making the software evolve. We need techniques and tools for reverse engineering (RE) that take into account the domain specific features. In this paper we relate RE cases. This is not a systematic study but lessons from our experience.

This paper highlights the complex use of reverse engineering in the context of re-engineering. We experimented different approaches on different case studies that belong to different application domains: extracting software component architecture for better design, software modernisation of information systems and rethinking a manufacturing control application. The lessons learnt from these experimental works open tracks for future vision and future work.

The paper is structured as follows. Section 2 overviews Model-Driven Reverse Engineering and open issues. Three case studies are then presented. Section 3 reports experimentations on software architecture extraction. Section 4 reports an experimentation in the information systems domain with pre-defined target models and large-scale case studies. Section 5 presents an on-going experimentation in the manufacturing domain with pre-defined target models and a medium size case study. Lessons learnt and related works are discussed in Section 6. Finally, Section 7 summarises the contribution and draws perspectives.

Pascal, A.
Case Studies in Model-Driven Reverse Engineering.
DOI: 10.5220/0007312502560263
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 256-263
ISBN: 978-989-758-358-2

2 BACKGROUND

Abstraction hides implementation details. During the development of software systems, high level abstraction models refer to system analysis and design while low level ones refer to implementation and deployment. The Object Management Groupe (OMG) identified four types of models in the Model-Driven Approach (MDA): *Computation Independent Model* (CIM), *Platform Independent Model* (PIM), *Platform Specific Model* (PSM) and an *Implementation Specific Model* (ISM) (Brown, 2004). The relations between *model elements* of different layers are *refinement* or *traceability*. Sometimes inheritance is used to materialize the abstraction between comparable model elements. People also use *abstraction layers* to represent the organisation of complex architectures. Typical examples are the ISO stack of protocols and services for telecommunications or the service architecture approach (SOA). In MDA, an engineering process can be seen as a sequence of model transformation where each transformation refines an (abstract) model to a more concrete one by the means of information (*cf.* Figure 1). In this paper, the term **Information** denotes any piece of knowledge (source code, documentation artefacts, configuration, procedures, models, etc.) about a software application.

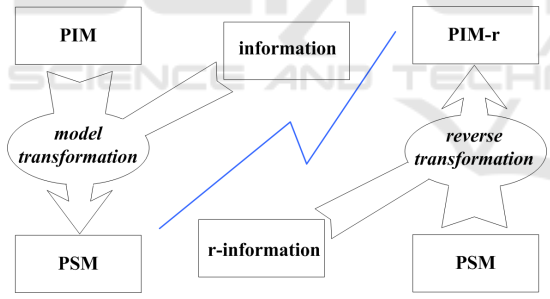

Figure 1: Reverse Model Transformation.

Model-Driven Reverse Engineering aims at producing high (abstraction) level models from software systems. Various objects can lead to use MDRE in software maintenance:

- Extract information of lost or deprecated design documentation.
- Understand an existing software solution with missing documents.
- Re-factor application to improve the quality or follow new coding standards.
- Align business processes with legacy applications.
- Upgrade technical framework releases or updating technical components.
- Extract software components to put on the shelf.
- Improve genericity by replacing hard coded information by configuration files.
- Modify the presentation layer or the persistence layer in n-tier web applications.
- Change the programming languages (*e.g.* from Cobol to Java).

Abstraction can raise at different abstraction levels, from program representation to the application architectures or business processes. Consequently, different kinds of models are expected with various notations like de facto MDE standards such as UML, OCL, MOF, EMF, SysML, AADL, BPMN or customised models defined with domain specific languages (DSL). The source information also differs and may include binary code, source code, configuration files, tests programs but also textual documentation or user scenarios. Conversely to engineering, MDRE can be made of transformations but usually transformations are not reversible as illustrated by Figure 1.

In the sections 3 to 5, we overview three different cases of reverse engineering practice for re-engineering: For each case we present the objectives, the proposed approach and the results.

3 SOFTWARE ARCHITECTURE EXTRACTION

The first case focus on reverse-engineering software architecture to reduce software architecture erosion, a common problem in legacy applications. Because they do not know or do not understand the original architectural intent, maintainers introduce changes that may violate the intended architecture and properties. The objective of this research project[1] was to establish a link between component implementations and component models in order to statically check properties such as safety and liveness. In reverse engineering, one cannot expect a “random” application to follow strict development patterns, for example with clear separation of communications, data types, components types, etc. In this work we suppose that an application was developed with componentisation in mind, but not necessarily with the required rigour. This would be the case for example, when one designs an architectural model, with proper specification of components and allowed communications, but implement the application with typical industrial approaches such as Corba, .NET, J2EE, or OSGI that focus on the runtime infrastructure, but provide little

[1] https://tinyurl.com/y7fjujgh

support for automatic verification of properties.

Figure 2 depicts the abstraction processes. Both processes are different but the Structural Abstraction (SA) stands first because the Behavioural Abstraction (BA) takes benefit from the result of SA. We defined a common component meta-model (CCMM) as a standard for component based modelling languages such as Kmelia[2] or Sofa[3]. These languages have tool support to verify models and check properties.

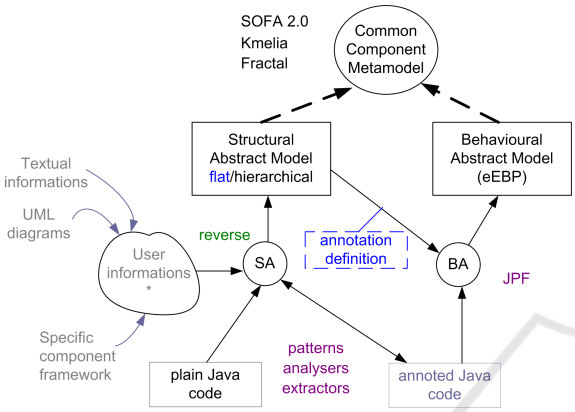

Figure 2: Component Abstraction Process.

The input data of processes SA and BA is a collection of source code (*e.g.* plain Java code here). Additional user informationcan help to define heuristics. For example, if the implementation is based on a specific framework like Corba or Sprint, one can look for specific patterns to detect components, services or communication protocols. If there exists a deprecated abstract model, one can try to match with the current implementation. The process can also take benefit of code annotations written manually by users or automatically by some code generator.

The abstraction processes are designed as itérative transformation processes where each step consists in applying one transformation of a tool box, as depicted by Figure 3. Each iteration applies one transformation (one tool of the toolbox) to the (possibly annotated) source code and produce the same code with new annotations. The idea is to combine transformations in a customized human driven process. As an example, let detail the "Model from code" transformation of Figure 3. We developed the tool, called `Javacompext`, for structure abstraction: recovering components and communications from a plain Java application. The input is a (possibly annotated) Java source code, and the output is a set of components with several kind of relations between them (communications, links, inheritance) and a set of data types. Details are given

[2]https://costo.univ-nantes.fr/

[3]https://sofa.ow2.org/

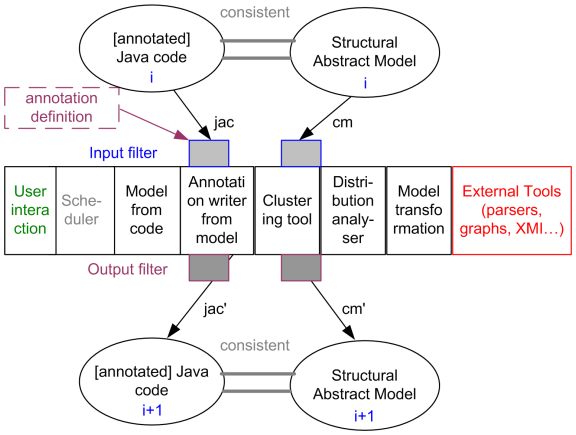

Figure 3: Iterative Abstraction Process Toolbox.

in (Anquetil et al., 2009).

We experimented `Javacompext` on various implementations of the Common Component Modelling Example (Rausch et al., 2008). CoCoME is a contest to evaluate and compare the practical appliance of existing component models and the corresponding specification techniques using a common component-based system as modelling example. Based on a UML-based description of CoCoME, a provided sample implementation, and test cases, the participating teams had to elaborate their own modeling of CoCoME, applying their own component model and description techniques. The example describes a Trading System as it can be observed in a supermarket handling sales. This includes the processes at a single Cash Desk like scanning products using a Bar Code Scanner or paying by credit card or cash as well as administrative tasks like ordering of running out products or generating reports. a deployment model, or test cases. The CoCoME specifications (abstract model) is relatively detailed with components, sequence, or communication diagrams. However it does not explicitly identify all the services, and only a few appear in sequence diagrams.

We used three instances of CoCoME, composed of a meta-model and its implementation. For example, the reference sample implementation is written in plain Java with some (undocumented) conventions. It includes 5078 LOC, 40 packages, 95 classes, 20 interfaces and 375 methods. The results were quite encouraging. Although the rules to recognize component types may seem very strict, in the chosen context (application developed with components in mind), they worked quite well. We could very quickly discover mappings between the concrete code and the abstract model which was one of our goals. `Javacompext` also highlighted big mismatches between the designed application (abstract model) and

the implemented one.

4 INFORMATION SYSTEM ALIGNMENT

The second case focus on reverse-engineering software architecture of Information System legacy applications. Maintaining legacy systems, *i.e.* the current state of the IT, besides new architectures or new business rules, remains an ongoing but costly concern (Clark et al., 2012). The problem is to reduce Business-IT misalignment between the Information Technology (IT) and Business viewpoints, which evolve separately. Our motivation is to help decision makers to capture the alignment of legacy systems with the related business models in order to underline the cross effects of IT or business evolutions. In (Pepin et al., 2016) we proposed a *method* to touch on the alignment of legacy systems by a pragmatic way. Different meta-models are defined at different abstraction levels (application, functional and business process). They are inspired by Enterprise Architecture frameworks (*e.g.* TOGAF). We defined meta-model inter-relationships in order to support alignment. The method consists in gradually (i) building models from the business side, (ii) extracting models from the IT side and (iii) relating the models consistently.

Reverse-engineering contributes to step (ii) in providing a way to feed the models by mining the existing information sources (code, models, documents). We implemented techniques to feed the corresponding models from legacy information (source code, data and models when they exist). Figure 4 shows the involved models and transformations performed during step (ii).

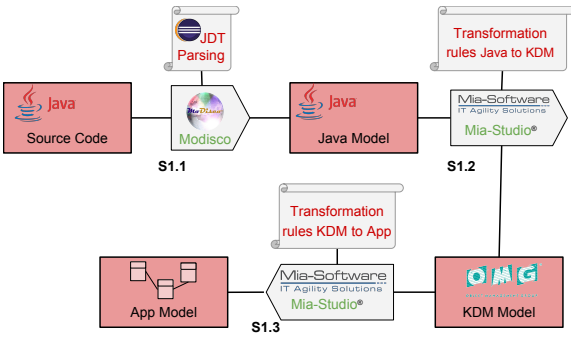

Figure 4: Transformation steps.

The goal is to retrieve an *application model* from the source code. The distance between the programming language level and the application architecture level is too wide to be processed in a single transformation step. Therefore the application model is produced by a stepwise abstraction where three transformations are necessary as depicted in Figure 4.

1. The *Source Code Reverse Engineering (S1.1)* consists in analysing the source code files of a set of programs in order to get a representation of the code in a model (PSM). This assumes a meta-model for each target programming language *e.g.* Java, C++, etc. We implemented Java Reverse Engineering using Modisco (Brunelière et al., 2014). This tool discovers the Java model with the assistance of the Java Development Tools (JDT) which parses the source code and computes an Abstract Syntax Tree (AST) according to a Java meta-model.
2. The *Intermediate Transformation (S1.2)* produces an instance of the Knowledge Discovery Meta-model (KDM)[4] intermediate model (PIM). KDM includes many layers to save different aspects of common programming languages and more generally it defines common concepts for software assets and their operational environments. We used only a part of KDM, mainly the code package which contains all the features of the current programming language. We faced the problem of scaling the model size because Modisco is not dedicated to store much information. So, we had to write our own transformation using the Mia-transformation tool.
3. The *Transformation to a High-level Abstraction (S1.3)* is to transform the KDM model into our App model which captures only the architectural information to be aligned later with the business. The abstraction distance between input and output is really larger than those of step S1.1 and S1.2. The existing tools suffer from limitations here. Finer algorithms are required such as the transformations exhibited for case 1 in Section 3. As a matter of fact, we exploited specific information, related to our case studies (see below) to build an algorithm and a set of rules to detect the different concepts from the application meta-model. This algorithm is more or less complex depending on the architecture used in the source code. Making such an abstraction exploits the inter-relationships described between the models to align. For example, we wrote a Mia-transformation to obtain an application model containing the links between components, interfaces, services, functions and data objects. The source code of the cases studies used a specific file naming based on name pre-

[4]Used by the modernization community in mining tools. http://www.omg.org/technology/kdm/

fixing. This naming policy identifies the software architecture role of different Java interfaces and classes. This unusual feature helps us to create an convenient transformation in a short time.

The experimentation was led on real case studies provided by French Mutual Insurance companies. The inputs include heterogeneous data such as java source files, enterprise architecture repositories (MEGA), databases. For example, one case study was a complete source code written in Java. The input source code was large: 33,400 classes, about 3,400,000 code lines. A side-effect of the study was to evaluate the capability to handle large-scale systems. The experiments showed that big mappings are hardly manageable by humans and tool assistance is mandatory. The reader will find more details in (Pepin et al., 2016).

5 MANUFACTURING CONTROL SYSTEMS RE-ENGINEERING

The third case focus on reverse-engineering a manufacturing application in order to re-engineer it according to MDE. The context of this case study is manufacturing control in Industry 4.0 where software systems become of prime importance. The starting point is a double finding, from literature review and current practice. Related works mention that service engineering in the context of Cyber-Physical Production Systems is still a craft activity, usually at the implementation level (Rodrigues et al., 2015; Morariu et al., 2013). In practice, we started from the application of Gamboa Quintanilla *et al.* that implements a Service-oriented Holonic Manufacturing Systems (SoHMS) (Quintanilla et al., 2016). A HMS is an agent-based systems that acts as a digital twin of the real manufacturing workshop ; it enables to control or simulate the workshop. Gamboa implemented a manufacturing workshop, called SOFAL. Its current implementation is depicted in Figure 5. It consists of two Java applications that exchange informations through sockets. Both applications are tightly coupled to the SOFAL workshop and reconfiguring it implies to compile the software. The programs includes 160 classes, 1240 methods and 14802 lines of code. The documentation is a PhD report including UML diagrams. Products and manufacturing orders are described with a human-machine interface (HMI). SoHMS, the running control system, can be bound to a simulator (Arena tool) or to the real workshop.

The objective is to switch from classical development to MDE. Instead of coding, the idea is to introduce automation and generate the code from models in order to be more reactive when reconfiguring the workshop. Moreover, handling models enables to reason with abstract concepts and to verify expected properties without taking into account useless implementation details. The motivations and new organisation are explained in (Tebib et al., 2019).

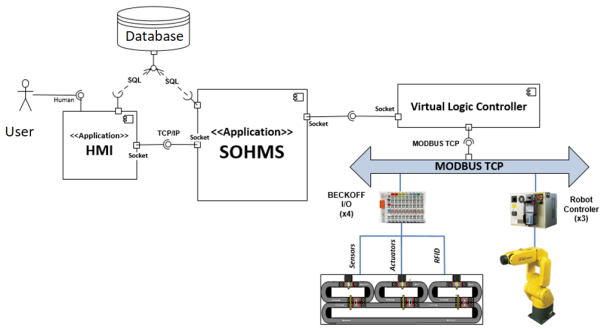

Figure 5: SOFAL manufacturing software.

The role of reverse-engineering is to mine the source code to capture model elements that can be compared with the available UML diagrams. It helps to discover and understand the application. Behind discovering we also want to separate the concepts that are specific to SOFAL from those which are common to different workshop. Factoring concepts will help to get higher abstraction and promote reuse. This abstract part of the application constitute the SOHMS framework, a Java library we will shared by the future manufacturing projects. The specific layer includes those features which depend on one specific workshop (products, ressources, orders, flows). We also separate the workshop elements (resources and flows) from the manufacturing ones (orders and products). The primer are rather static (even for reconfigurable manufacturing systems) while the latter are merely dynamic, time dependent and subject to quality of service constraints.

The reverse-engineering process extracts abstractions to be sorted in the above categories (common/specific, workshop/manufacturing). We choose UML as modelling language due to its widespread use in the development community but also because various MDE tools accept UML models (or EMF models in the Eclipse world). We tried different tools for reverse-engineering UML models from Java source code such as Modelio, Papyrus, Modisco, Eclipse UML, Enterprise Architect or ObjectAid. Few of them enable visual facilities for exploring the resulting diagrams, ObjectAid was helpful for this.

Finally, even if this is not a definitive opinion, the existing tools mainly propose a static representation of the code (system structure and system behaviour). We already made similar findings in the ex-

perimentations of Section 3 and Section 4. The automatic extraction of communication scenarios (UML sequence diagrams) or object life cycles (UML statecharts) from plain Java are, as far as we explored, almost limited in the tested tools. ObjectAid[5] provides visual representations to be included manually in class diagrams. Sequences diagrams were available in the licensed version. Other tools usually provide internal representation or XMI format files.

In practice, the automatic extraction provided too much useless details to prevent us to understand how the two applications were designed. The gap between code models and design model is really big. Consequently we adopted a two-way approach, where reverse-engineering activities interleave and modelling activities in an iterative process (Figure 6).

1. Reverse engineering (bottom-up) discovers the implemented structures (for both structural and behavioural aspects) with code model extraction, code reading.
2. Model engineering (top-down) writes UML models (class diagrams, sequence diagrams, activity diagrams...) from research reports, interviews, assumptions and of course the results of the previous reverse engineering iterations.

The idea is to iterate and converge until a sufficient level of agreement is reached.

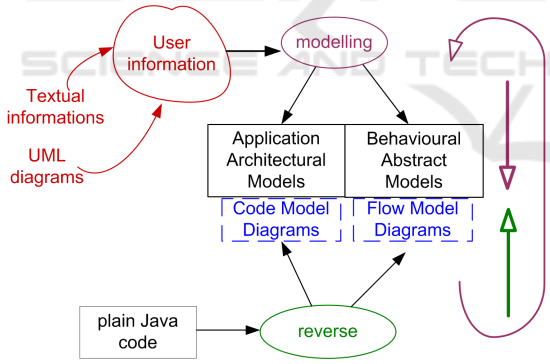

Figure 6: Two-way reverse engineering.

The result of the two-way reverse engineering is a collection of UML diagrams that helps to build the SoHMS framework which contains the core execution of the holonic system. It serves as documentation of the SoHMS framework which is currently designed as a Java library. We are working on a modelling support (a Domain Specific Language and a visual editor implemented with Sirius[6] and on model transformations to Java and to FlexSim[7], a simulation system. This part is still under construction in the process but details are available in (Tebib et al., 2019).

[5]http://www.objectaid.com/
[6]http://www.eclipse.org/sirius/
[7]https://www.flexsim.com/

6 DISCUSSIONS

We present lessons learnt from the above case studies and related works.

When model-driven reverse engineering legacy systems, it is a primary requirement to have techniques to build new models. We observed that the existing tools mainly propose a static representation of the code (system structure and system behaviour), similarly as described in the experimentations of Section 3 and Section 4. The behavioural abstraction is more complex to establish because investigating the computation flows is a kind of evaluation. MDRE tools provide convenient abstract views of the code but more intelligent algorithms are required to raise in abstraction. In the three above presented cases, we experimented various strategies.

1. In the case presented in Section 3, heuristics enable to classify candidate classes into components or data types.
2. In case studies of Section 4, we use engineering methods and rules including naming conventions and package organization.
3. In the case presented in Section 5, we compared the (manual) top-down models from the bottom-up abstractions at each iteration step.

During these experiences, we note the following findings about MDRE.

1. The process is guided by the objectives (what you look for) and the results will depend on them. It can be used to prove properties or estimate the quality (*e.g.* case 1), to evaluate the impact of evolution (*e.g.* case 2) or to re-engineer a new application (*e.g.* case 3). Other scenarios are given in (Raibulet et al., 2017), including comprehension, documentation, software quality assessment.
2. Automatic high-level reverse-engineering for general purpose object languages such as Java or C++ stay a myth. Efficient generic tools exist, such as Modisco, which provide an abstract view of the program but can hardly raise in abstraction to architecture levels. Having information or expertise knowledge is mandatory to compute abstractions, *e.g.* heuristics or design choices, naming or structuring conventions, traceability links, best practices, patterns, etc. In Section 3, we added annotations in the source code to assist the tool in detecting the abstractions. In the best case one may have an incomplete model designed during the en-

gineering process. Also the process must be interactive or user guided because the (reverse-)design decisions depend on experts. Having third party MDRE is an additional drag. The result would be better with specific-purpose languages.

3. One step reverse-engineering is impossible to raise in abstraction. We are convinced that the intelligence is in the transformation process not in the individual transformations. Only small steps can be easily implemented by developers with simple transformation rule sets. For example, coding UML models (including classes, state-transitions and activity diagrams) directly into Java is harder and less reusable than transforming (i) to a UML profile without aggregation, and bidirectional association, (ii) to a UML profile without multiple inheritance, etc.
4. There are no universal process. Each of our case study was different in terms of source information and MDRE objectives. Consequently the reverse engineering must be able to accept customizations (or local knowledge) as input parameter. As mentioned in (Raibulet et al., 2017), domain specific approaches provide better results *e.g.* for web services, Cobol programs or relational databases, etc.
5. Similarly to machine learning, a reverse engineering technique, designed for a given goal (lesson 1) in a given context (lesson 4), will be improved by applying it in new case studies. In the case of Section 3, the tool provides different results for the different implementations we tested. An open track is to improve the heuristics of the RE transformations by learning.
6. Discovering a model is much harder than comparing a model with an implementation. As experimented in the case of Section 5 and in some examples of Section 4, it is more easy to find something expected than something unknown. Having a reference provides not only potential abstractions but also the way abstractions were designed. In a MDE approach, the knowledge of the engineering transformation process impacts the way to drive the reverse-engineering transformation process. In particular, if the transformation process is made of *small steps*, it is simpler to write the reverse-transformations because the semantic distance is small between an implementation (a set of elements of the source code) and its abstraction. However reverse-engineering is never a symmetric transformation of engineering.
7. MDE helps in MDRE. If the engineering process is made of model transformations, it should be easier to get back to models. For example one can set traceability links in the code (*e.g.* by the means of code annotation). Model *round-trip engineering* goes further and involves synchronizing models and keeping them consistent. As a means for round-trip, Bork *et al.* propose an approach for reverse engineering of code generated by a template based code generator (Bork et al., 2008).

In coherence with MDA and lesson 3, we consider a reverse-engineering process to be a composition of reverse model transformations as depicted by Figure 1. The cost to pay is to manage intermediate meta-models. To avoid the multiplicity of modelling languages, one can define operators to reduce or to augment the features of one base language, similarly to UML profiles. Another problem is to manage meta-model evolution ; solutions with graph transformation are provided in (Mantz et al., 2015).

MDRE has been an active research field since a more than a decade. We already point out comparisons with related approaches on component based reverse-engineering (case 1) in (Anquetil et al., 2009) and legacy reverse engineering for application models (case 2) in (Pepin et al., 2016).

Reis and Da Silva propose a MDRE approach to produce high-level specifications of legacy information systems in a human-controlled way taht are reinjected in forward engineering (Reis and da Silva, 2017). A family of UML-based modelling language, called XIS*, enable to express the specifications.This approach could be complementary to ours since it focus on databases while we work on programs.

Considering MDRE in general, we refer to the recent synthesis of Raibulet *et al.* (Raibulet et al., 2017). Fifteen approaches are referenced and eleven partial approaches are discussed. The referenced approaches are classified into general purpose (or generic) approaches and domain specific ones (*e.g.* for cobol, web-services, web applications or relational databases). These approaches use a wide range of meta-models (*e.g.* OMG standards but mùany DSLs) but merely few tools (the most cited are MoDisco and ATL). Various case study are used but no benchmark emerged.

The most advanced approaches for MDRE is the one of Brunelièere *et al.* (Brunelière et al., 2014), based on Modisco, an open source MDRE framework. We are convinced Modisco is very helpful for model discovery ; we used it in case 2. However complementary approaches are necessary to raise in abstraction, as shown in case 2.

7 CONCLUSION

In the context of software maintenance, the well-known technical debt is almost always accompanied by an abstraction debt. Even in the good case were the specifications include software architecture and design models they are usually not up-to-date and not consistent with the current implementation and runtime configurations. Model-Driven Reverse Engineering of a legacy system provides means to get abstract models from the source code (or from the binary code). However a standard extraction from code, usually do not reach the goal because the tool support is more efficient on the static parts than the behavioural part. Experiments show us that MDRE is a complex activity that requires expert assistance, customization to fit the MDRE goals and progressive abstraction raising. MDRE is not symmetric to MDE ; the traceability links that bind (abstract) design concepts to (concrete) implementation concepts are many-to-many. Things can change if MDRE is anticipated during engineering by using small step transformations and by putting explicit traceability information in the source code *i.e.* preparing round-trip.

The next step will provide more assistance to MDRE user. For example we target the implementation of heuristics to propose a list of possible model abstraction mappings to the modeller, she can then choose the desired ones. These heuristics will depend on the nature and the semantics of the mappings. For example, when mapping two releases of the same model, it is usually easier to detect equality mapping. In specific cases, one can detect patterns or naming conventions.

REFERENCES

Anquetil, N., Royer, J., Andre, P., Ardourel, G., Hnetynka, P., Poch, T., Petrascu, D., and Petrascu, V. (2009). Javacompext: Extracting architectural elements from java source code. In *2009 16th Working Conference on Reverse Engineering*, pages 317–318.

Bork, M., Geiger, L., Schneider, C., and Zündorf, A. (2008). Towards roundtrip engineering - a template-based reverse engineering approach. In Schieferdecker, I. and Hartman, A., editors, *Model Driven Architecture – Foundations and Applications*, pages 33–47, Berlin, Heidelberg. Springer Berlin Heidelberg.

Brown, W. A. (2004). Model driven architecture: Principles and practice. *Software and Systems Modeling*, 3(4):314–327.

Brunelière, H., Cabot, J., Dupé, G., and Madiot, F. (2014). Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032.

Clark, T., Barn, B. S., and Oussena, S. (2012). A method for enterprise architecture alignment. In *Proceedings of PRET*, volume 120, pages 48–76. Springer.

Cuadrado, J. S., Izquierdo, J. L. C., and Molina, J. G. (2014). Applying model-driven engineering in small software enterprises. *Sci. Comput. Program.*

Dehaghani, S. M. H. and Hajrahimi, N. (2013). Which factors affect software projects maintenance cost more? *Acta Informatica Medica*, 21(1):63–66.

Mantz, F., Taentzer, G., Lamo, Y., and Wolter, U. (2015). Co-evolving meta-models and their instance models: A formal approach based on graph transformation. *Science of Computer Programming*, 104:2 – 43. Special Issue on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013).

Morariu, C., Morariu, O., and Borangiu, T. (2013). Customer order management in service oriented holonic manufacturing. *Computers in Industry*, 64(8):1061 – 1072.

Pepin, J., André, P., Attiogbé, C., and Breton, E. (2016). An improved model facet method to support EA alignment. *CSIMQ*, 9:1–27.

Quintanilla, F. G., Cardin, O., L'anton, A., and Castagna, P. (2016). A modeling framework for manufacturing services in service-oriented holonic manufacturing systems. *Engineering Applications of Artificial Intelligence*, 55:26–36.

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

Rausch, A., Reussner, R., Mirandola, R., and Plasil, F., editors (2008). *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, Heidelberg.

Reis, A. and da Silva, A. R. (2017). Xis-reverse: A model-driven reverse engineering approach for legacy information systems. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development,*, pages 196–207. INSTICC, SciTePress.

Rodrigues, N., Leitão, P., and Oliveira, E. C. (2015). Self-interested service-oriented agents based on trust and qos for dynamic reconfiguration. In Borangiu, T., Thomas, A., and Trentesaux, D., editors, *Service Orientation in Holonic and Multi-agent Manufacturing*, volume 594 of *Studies in Computational Intelligence*, pages 209–218. Springer.

Rugaber, S. and Stirewalt, K. (2004). Model-driven reverse engineering. *IEEE Software*, 21(4):45–53.

Selic, B. (2008). Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*, 15(3):379–391.

Tebib, M. E. A., André, P., and Cardin, O. (2019). A model driven approach for automated generation of service-oriented holonic manufacturing systems. In *Service Orientation in Holonic and Multi-Agent Manufacturing - Proceedings of SOHOMA 2018, Bergamo, Italy, June 11-12, 2018*, volume 803 of *Studies in Computational Intelligence*, pages –. Springer. To appear.