Redocumentation of a Legacy Banking System

An Experience Report

Joris Van Geet
University of Antwerp, Belgium
Joris.VanGeet@ua.ac.be

Peter Ebraert
University of Antwerp, Belgium
Peter@Ebraert.be

Serge Demeyer
University of Antwerp, Belgium
Serge.Demeyer@ua.ac.be

ABSTRACT

Successful software systems need to be maintained. In order to do that, profound knowledge about their architecture and implementation details is required. This knowledge is often kept implicit (inside the heads of the experts) and sometimes made explicit in documentation. The problem is that systems often lack up-to-date documentation and that system experts are frequently unavailable (as they got another job or retired). Redocumentation addresses that problem by recovering knowledge about the system and making it explicit in documentation. Automating the redocumentation process can limit the tedious and error-prone manual effort, but it is no silver bullet. In this paper, we report on our experience with applying redocumentation techniques in industry. We provide insights on what (not) to document, what (not) to automate and how to automate it. A concrete lesson learned during this study is that the "less is more" principle also applies to redocumentation.

1. INTRODUCTION

More and more industrial processes are supported by software systems. In order to maintain and evolve those systems, a profound knowledge about their architecture and implementation details is desired [2]. Such knowledge can take an explicit form (in the documentation of the software) or an implicit form (in the heads of the experts). Unfortunately, the availability of this knowledge is often unsatisfactory. This problem becomes most apparent in legacy systems, as experts retire, documentation gets outdated or is not there at all [**?**].

Redocumentation can be used to overcome this problem. The intent of redocumentation is to recover (hence, the 're-' prefix) documentation about the subject system that existed or should have existed [7]. This recovery can be done in a *manual* or *automatic* way. Manual redocumentation requires labour from the experts (the developers, architects, users or others). It is tedious, error-prone, not intellectually stimulating and can only be applied if the experts are available [**?**]. Automatic redocumentation seems much more promising, as it does not require manual labour [**?**]. Nonetheless, some things cannot be automated.

In this study, we report about a redocumentation experience from the field which provides insights on what to automate (and what not to) and how to accomplish this. The paper is structured as follows. We start of by presenting the context in which this study was performed in Section 2. Afterwards, in Section 3, we sketch the redocumentation process we applied. We finish the paper with a summary of all lessons we learned (Section 4). We conclude (in Section 5) that the purpose of these lessons is twofold. First, practitioners can use them to improve the usability, quality and maintainability of the documentation they create. Second, researchers can use them to improve their (re)documentation tools and methods.

2. PROJECT CONTEXT

This section establishes a context in which the study was performed. It elaborates on the organization and its typical development methods, illustrating the need for documentation. Furthermore, it explains the documentation standards within the organization, some specifics about the system under study and the actual goals of the redocumentation effort.

2.1 Organization

According to Burns and Stalker [4], organizations either have a more flexible *organic* culture or a more structured *mechanistic* culture. Both have their merits and can be deliberately created and maintained to use employees in the most efficient manner according to the organization's circumstances [4, p119]. The organization in which we apply the case study adopts a mechanistic culture thereby favoring a more structured, *industrial* approach to the software development lifecycle. This results in strict rules and standard operating procedures in which all employees have clearly defined responsibilities. Communication between these *units of labour* is facilitated by comprehensive documentation [14].

The maintenance of the system on which we report, will soon be outsourced in order to cut expenses [11]. In general, it is more and more common to outsource or off-shore parts of the development process, such as the maintenance [1]. Clearly, comprehensive documentation is a prerequisite here, because the subcontractor cannot always fall back on the implicit knowledge which is present in the heads of the developers.

2.2 Documentation Standard

The organization uses an enterprise-wide documentation standard which is largely based on Component Based UML [6], tailored to the specific structure of typical banking applications. Furthermore, this standard is supported by MEGA [1], a general purpose documentation tool which is highly customizable. At its core it has only two notions: objects with certain characteristics and links between objects with certain characteristics. To support the rigorous documentation process, the MEGA tool has been customized to support this tailored version of UML.

There are documentation guidelines for all phases of the software development life-cycle. On the domain level, for example, diagrams are defined to document the business processes within that domain, indicate which parts of these processes are automated and which systems are responsible for it. On system-level, functional documentation is required in the form of use cases and conceptual data models. In turn, these have to be linked to technical documentation containing all functions, database tables, screens, batch jobs and the dependencies among them in several UML class diagrams. Furthermore, the purpose of each function is documented as a comment and the logical steps within that function are documented in UML sequence diagrams.

Some parts of this information are easy to retrieve automatically, other parts are not. For example, it is usually possible to retrieve technical data dependencies from the source code, but one cannot automatically retrieve the logical steps of a program from the source code because this requires human interpretation.

2.3 System under Study

The system under study is the Custody Services Back-end (which we will refer to as CSBE) of a large Belgian Bank. In short, custody is a service consisting in holding and administering securities[2] on behalf of third parties [5]. CSBE is implemented in CA GEN, a Computer Aided Software Engineering (CASE) application development environment marketed by Computer Associates[3]. CA GEN was originally known as IEF (Information Engineering Facility)[4]. Developers can create entities, procedures, procedure steps and action blocks in the CASE tool using both visual models and textual code. Consequently, COBOL code and DB2 database schemas are generated which are compiled and deployed on an IBM mainframe running z/OS.

The system is currently undergoing a technical conversion for two reasons. First, it aims to move away from the CA GEN environment, which will no longer be supported, and replace it with EGL, a platform-independent, business-oriented language that is being introduced throughout the entire organization. Second, it aims to move away from an outdated mainframe environment which is currently only being supported to run this one system created in CA GEN.

[1]http://www.mega.com/

[2]Security: a certificate attesting credit, the ownership of stocks or bonds, or the right to ownership connected with tradable derivatives. — New Oxford American Dictionary

[3]http://www.ca.com/

[4]http://en.wikipedia.org/wiki/Information_Engineering_Facility

When the technical conversion project started, there was no documentation about CSBE available in MEGA. However, since the maintenance being done on the current version of CSBE (in parallel with the technical conversion) is being outsourced, it is necessary to document the required changes very well. Therefore, the MEGA documentation is created on an as-needed basis, i.e., if a function needs change (1) the function is documented as-is, (2) the change is documented, and (3) the documentation is sent to the subcontractors for implementation. As such, some documentation is already available in the MEGA tool.

Although the documentation standard itself is very well documented and clearly articulates which things should be documented and how, there are two ways in which CSBE cannot adhere to this standard. Firstly, the documentation standard has been created under the assumption that mainframe systems will only be used on the back-end, i.e., without direct interaction with the end-user. This is indeed a policy being pushed within the organization, however, our legacy system still uses mainframe screens to interact with end-users. These screens are not provided in the documentation standard. Secondly, a subsystem is defined as a set of data and a set of functions working on that data. Functions can be exposed to other systems but the data will always be encapsulated within the subsystem (much like with component based development). However, CSBE is not structured like this. It is one big system but due to technical scalability issues it has been divided into logical subsystems all working on the same data. This is likely to cause issues when documenting the system.

2.4 Project Goal

Knowing that it is not possible nor desirable to automate all documentation, the goal of this project was to automatically generate parts of the technical documentation (so nothing on domain or functional level) to provide a basis for a complete redocumentation effort.

More specifically, we decided to (i) reduce laborious, repetitive and boring manual effort, (ii) focus on quick wins, i.e., reduce a significant amount of effort with minimal investments, and (iii) take into account reuse of the redocumentation process for other legacy systems in the future.

2.5 Why not buy a tool?

Given this goal, one could argue to buy a commercial tool that can collect the necessary information from the source system and transform it into correct documentation. However, in [13] we have already argued that the biggest cost for applying reverse engineering on mainframe is in the various languages and dialects used in these legacy systems. Furthermore, van Deursen and Kuipers attest that they have yet to see a legacy system that does not use compiler-specific extensions or locally developed constructs [12]. Should there be a commercial tool available that, after the necessary customization, could collect the necessary information, it would not be worth the cost as the next system within the organization will be written in yet another language. Moreover, no commercial tool will be available to transform these facts into the documentation tool MEGA, which is specifically tailored to the organization's needs.

Consequently, a customized redocumentation process is the most cost-effective, especially because it can be tailored to some organizational standards making reuse within the organization more likely.

3. REDOCUMENTATION PROCESS

From a high-level perspective, two steps are necessary to redocument a system. Firstly, one needs to extract facts about the system. That can be done starting from a static representation of the system (e.g., the source code), from already available documentation (text documents, spreadsheets, ...) or from the people working with the system. Secondly, these facts need to be combined and transformed into the correct documentation format (e.g., UML diagrams or hyperdocuments).

Although not strictly necessary, we choose to add an extra step in between. Namely, the representation of the extracted facts in a language-independent model, i.e., independent from the source system characteristics and independent from the target documentation format. This way, part of the approach can easily be reused for redocumenting other systems within the organization using the same documentation format. Furthermore, when the documentation format changes (which is currently happening in the organization), our redocumentation approach can be easily adopted as well.

Figure 1 depicts the details of these three steps, which we will explain in the next three sections.

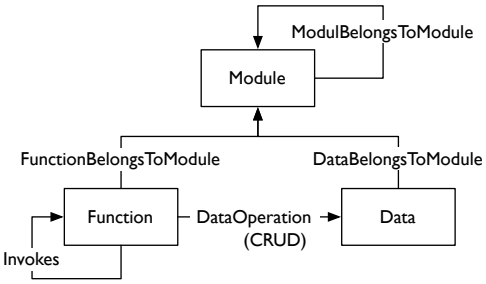

Figure 2: Basic RSF meta-model

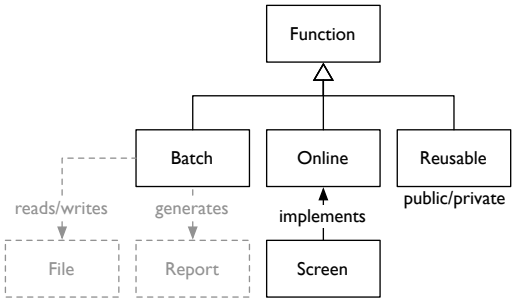

Figure 3: Different functions within RSF meta-model

3.1 Fact Extraction

On a technical level, there are two kinds of facts which can be gathered. On the one hand, *structural* facts can be extracted from the system; for example, program names, functional dependencies, data dependencies, etc. On the other hand, there is also more *semantic* information; like the purpose of a function or the logical steps within a function. The latter are notoriously harder to extract, mainly because they are not provided (consistently) as they are not necessary to compile and run the system.

We gather both structural and semantic facts from three different sources.

The CA Gen Environment. Since CA Gen is a CASE tool it does not store its information as plain-text source code. Rather, it stores its information in the CA Gen Host Encyclopedia, the master data repository containing screen definitions, interaction models, pieces of code, etc. A public interface to this complex host encyclopedia is provided in the form of about 100 database tables. Although limited detail is provided (no information about individual statements), the most important facts for information systems are available. Structural facts such as action step usage, data usage and data definitions, but also semantic facts such as descriptions of action blocks and data definitions. We extract this information via several relatively straightforward SQL queries. We download the generated SQL reports from the mainframe for later processing.

Spreadsheets. Some information is not available in the CA Gen environment. For example, CSBe has been created as one big system, without the notion of subsystems. During its lifetime, it has been split up rather arbitrarily in several CA Gen models because of technical scalability issues. Because of the technical conversion to the standard platform, a more functional separation has been semi-automatically created and documented in a spreadsheet. We use this input to categorize all functional entities in the correct subsystem, which we need to create separate documentation repositories for each subsystem.

Existing Mega Repositories. As explained before, some documentation recently became available in the Mega tool due to outsourced maintenance. The structural parts can just be replaced by what we will generate. However, some more semantic parts required significant manual effort and cannot be automated. For example, documenting the logical steps within one function is an inherently manual process as it requires human interpretation. To make sure we take this documentation into account we extract it from the existing Mega repositories.

3.2 Fact Representation

All the facts extracted in the previous step are stored in a plain-text format, be it table-like SQL reports, comma separated values or the Mega export format. This makes them relatively easy to process using nothing but regular expressions and string manipulations implemented in Perl.

As intermediate format we use RSF, the Rigi Standard Format, conceived as part of the Rigi Reverse Engineering Environment. RSF is a lightweight, text-based exchange format

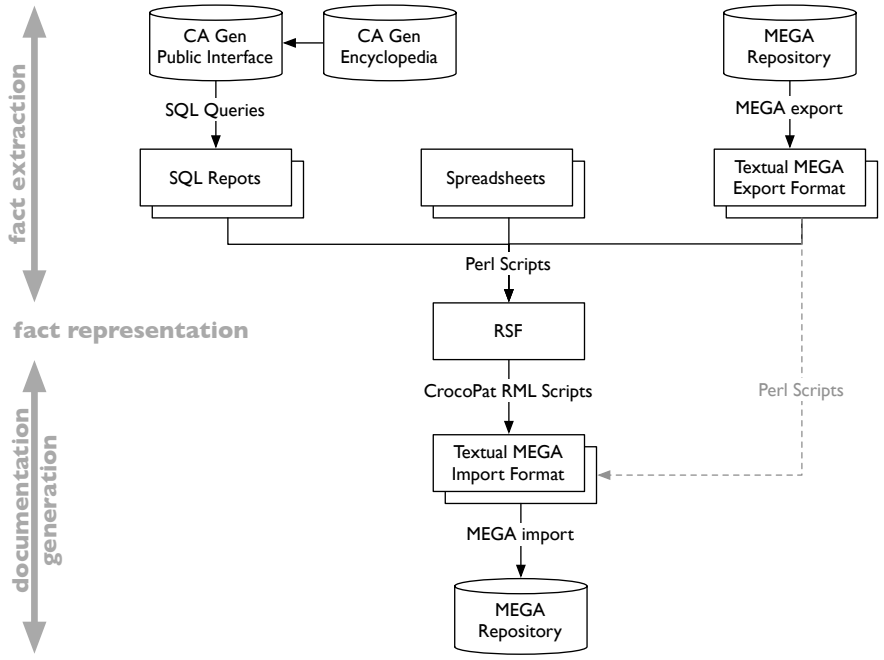

Figure 1: Overview of the redocumentation process

that uses sequences of tuples to encode graphs. As such, a tuple can (1) define a node and its type, (2) define an edge between two nodes, or (3) bind a value to an attribute. Using this format, one can specify anything that can be represented as a typed, directed, attributed graph. Software systems are commonly expressed as such graphs [9].

We choose RSF for its simplicity and for the potential reuse of tools and expertise from the FETCH [5] tool chain [8].

Figure 2 depicts the basis of our RSF meta-model. It reflects the common structure of information systems in the financial company for which we applied the case study, which boils down to data processing. Functions are basically steps within an automated process which, in our case, are implemented in COBOL programs generated from CA GEN. A data entity is simply a table in a database. Functions can manipulate data using CRUD operations (creating, reading, updating or deleting rows) on a table and invoke other functions to delegate some of its work. Both functions and data entities belong to a module, which in our case represents a part of a subsystem.

Figure 3 shows the different kinds of functions that are available. A function can be (1) a batch job used in bulk data processing, (2) an online step represented by a mainframe screen for interactive processing by a user, or (3) a reusable piece of functionality used by all three kinds of functions. A reusable function can either be public or private indicating whether or not it is an exposed functionality of the subsystem. A batch job typically reads and writes files in bulk and generates a report. These batch characteristics are denoted in gray because they are not yet incorporated into our redocumentation process.

[5]FETCH stands for Fact Extraction ToolChain, info and tools via http://lore.ua.ac.be/fetchWiki/

3.3 Documentation Generation

After collecting all facts of CSBE in one RSF file, we process it to generate the desired documentation format. For this, we use CROCOPAT, a graph query engine by Dirk Beyer [3]. It supports the querying of large software models in RSF using PROLOG-like Relation Manipulation Language (RML) scripts. This way we can, for example, easily extract all the different interactions of a specific function, or an entire call-graph starting from an online screen.

Of course, for documentation purposes, this information has to be loaded into a documentation tool. For this, we use the textual MEGA input format. In essence, this is no more than a sequence of `.Create` and `.Connect` statements, to add certain objects to the MEGA repository and connect them in a meaningful way. What kind of objects can be created and how they can be connected depends on the meta-model used, in our case it was a meta-model based on UML. However, how this meta-model translates to specific MEGA statements is not documented at all. As such, we had to reverse engineer the MEGA format itself. The import and export formats of MEGA are very similar. Hence, we modeled some toy examples in MEGA and exported them to see how it would translate to a combination of objects and connections.

Listing 1 gives a simple example of the MEGA import format where two modules are created on lines 1–4, which translate to UML packages. Note how we can attach characteristics of objects like the name in line 2, for instance. Further-

more, two functions are created in lines 6–14, which translate into UML classes. The first function is marked as a public reusable function by means of a pre-existing UML stereotype and attached to the first package using the `OwnedClass` relation. The second function is marked as a batch job and attached to the second package. Lastly, on line 16, the reusable function is defined to be *required* for the batch job, essentially documenting an invoke relation.

As mentioned before, we generate this output directly from our RSF file using CrocoPat scripts. However, most of the already existing Mega documentation from the parallel maintenance projects is directly integrated with the output format, without being stored in RSF. The main reason for this is that the documentation mostly involves rich-text comments which are (1) not supported by RSF and (2) stored in an obscure hexadecimal format proprietary to the Mega tool. By feeding it directly back into the input format, we circumvent all translation issues. Concretely, we use CrocoPat scripts to generate a template Mega input file for the comments and then use some Perl scripts to merge this template with the extracted comments from Mega.

Furthermore, some technical limitations to the Mega import format prevent us from automating the generation of all diagrams defined in the organizations documentation standard. For example, you can disable specific operations from a UML class in a specific view, essentially making them invisible in that view. This approach is used to indicate whether or not a function performs a specific CRUD operation on a data entity. Because the visibility of operations are cosmetic properties of a diagram, they are not accessible via the import format. Hence, they can only be disabled manually. We did provide annotations with the correct information in each of the UML diagrams so maintainers are reminded to fix it and can do so without the need for additional information.

4. LESSONS LEARNED

In this section, we enumerate and explain the lessons we learned when applying the redocumentation process of Section 3 on the software system on which we elaborated in Section 2. They are all based on observations we made in a real-life case. The purpose is twofold. First, practitioners can use them to improve the usability, quality and maintainability of the documentation they create. Second, researchers can use them to improve their (re)documentation tools and methods. We present the lessons in an arbitrary order.

4.1 Only document relevant parts

While collaborating with the banking company, it became clear that they had only very limited resources to spend on the creation and maintenance of documentation. This was quite surprising, given that the banking company has a more mechanistic culture and adopts industrialized approaches in which documentation is essential to the software development process.

Furthermore, we were hard-pressed for time to automatically generate a basis which could be manually completed (there was a strict deadline on which the manual documentation should start). Therefore, we had to prioritize the kind of views and documentation we would tackle first. Our initial thought was to take the documentation standard of the organization and start with those things that are easiest to automate and the most mundane, repetitive and error-prone to do manually. However, this assumes that all views are equally important and should be there in the end anyway. In practice this is not the case, especially because the documentation standards are not necessarily constructed by people from the field. Rather, they reflect a *nice to have* view on documentation. As long as the necessary resources are not available to actually produce the documentation, those parts with the most practical value should come first.

The lesson we learned from these observations are twofold. Not only the effort of generating documentation should be considered, but also the effort for maintaining the generated documentation has to be taken into account, when performing a (re)documentation effort. For those reasons, it seems that one should only document the parts that will be relevant and that will be maintained and kept up-to-date. A prioritization of what to document should be a part of the (re)documentation process.

4.2 Less is more

We observed that when performing automated redocumentation, it is very easy to get carried away and generate documentation just because you have access to the facts. More documentation, however, is not always better. Feedback from the maintenance team learned us that there are two motivations for avoiding an over-production of documentation. First, a huge amount of documentation might lead to an information overload for the users of the documentation. This overload makes the documentation itself becomes less usable. Second, the trustworthiness of the documentation decreases as huge documentation repositories have a reputation of not being maintained and consequently are suspected of containing outdated information. This suspicion results in a further lack of commitment to keep the documentation up-to-date: a typical example of a self-fulfilling prophecy.

The lesson we learned from this observation is that we should not only ask whether we *can* generate the documentation, but also whether we *should*. The main criteria should be whether or not the generated documentation will be used and whether or not it will be maintained. If the answer to any of those questions is no, the redocumentation effort will have been in vain resulting in documentation that will soon be outdated again. We conclude that one should only create documentation that will be used and maintained. It is best to have little documentation that is easy to understand and that can be fully trusted.

4.3 Make documentation usable

The system under study contained a rather artificial structure in which one subsystem contained all the data that is used by the other subsystems. As the documentation policy states that the description of the data is only included in the subsystem defining the data and not in the subsystem using the data, all the documentation that explains the data is grouped into one repository. This is rather annoying from a documentation point of view, where maintainers claim they would benefit more from having the documentation available where they use or plan to use the data. Indeed, this

```
1  .Create ."Package" "p20" -
2      ."UML PackageName (English)" "C-Reusable Services-Stockbeheer"
3  .Create ."Package" "p81" -
4      ."UML PackageName (English)" "C-Batch-Stockbeheer"
5
6  .Create ."Class" "f3025" -
7      ."UML ClassName (English)" "E3EF0432-E0F04_SCHRIJF_INTERNE_STOCKS_EXT"
8  .Connect ."Class" "f3025" ."Stereotype" "Public Function"
9  .Connect ."Package" "p20" ."OwnedClass" "f3025"
10
11 .Create ."Class" "f3209" -
12     ."UML ClassName (English)" "E3M6180-E0F04_BESTAND_CONSISTENT_STOCKS"
13 .Connect ."Class" "f3209" ."Stereotype" "Regular Batch"
14 .Connect ."Package" "p81" ."OwnedClass" "f3209"
15
16 .Connect ."Class" "f3209" ."Required Interface" "f3025"
```

Listing 1: Example Mega input file

hinders the maintenance process and makes the maintainers sometimes guess about the data documentation instead of looking it up.

The lesson we learn here is that the structure of the documentation has to support the way it will be used. The available information has to be easily accessible there where it is required.

4.4 Do not duplicate documentation

During redocumentation, we observed that the maintainers were not always satisfied with how the documentation was structured throughout the system. In Subsection 4.3 we already elaborated on the situation in which the data documentation is only included in the subsystem where it is defined, and not where it is used. To help the maintainers, we suggested to include the description also in the subsystems using the data. However, this suggestion was refused because this meant that every time something changed about the data, the documentation would potentially have to be updated in nine different locations. Something even the maintainers admitted would not happen.

We learn that it is not a good approach to duplicate documentation, as this would increase the maintenance costs of the documentation. Moreover, it might bring along inconsistencies within the information specified in the multiple copies. Instead, we would suggest to use references to the documentation. References can be inserted throughout the entire software system. They point to the relevant parts of documentation and support developers to obtain the relevant documentation information, wherever it is required.

4.5 Use a generic meta-model

The most straightforward way to perform a redocumentation effort is to implement a point-to-point transformation from the sources to the documentation. However, we observed that a lot of the transformation work is redundant both from the organizational as from the system perspective. Therefore, we opted to add an indirection in the redocumentation process, representing all the extracted facts in a generic way using RSF.

Consider Figure 4, in which the transformation is depicted from CSBe to the Mega UML format. The left hand side shows a point-to-point transformation, the right hand side includes an indirection via RSF. Both representations have a dotted arrow for extracting facts and a full arrow for generating the documentation from these facts. At first sight, this indirection increases the total amount of work, because the amount of arrows remains stable but a box is added.

However, when we study the system in detail we see a different picture altogether. Consider Figure 5 in which we depict the same transformation from CSBe to the Mega repository, but taking into account the different fact extraction sources and the different views to be documented. The first view could just require facts from one source to be transformed, but other views might need facts from several or all sources combined. This combination will have to be performed ad-hoc when no intermediate model is used. The example in Figure 5 shows a point-to-point transformation with nine arrows, whereas using RSF as intermediate model, the amount of arrows is reduced to six. Clearly, the necessary point-to-point transformations will explode when fact extraction sources and, more likely, documentation views increase.

Furthermore, the indirection allows us to reuse all the work on transforming to Mega, independent from the implementation language of the source system. This is very relevant in our context because Mega is used throughout the entire organization and CA Gen is only used for the Custody Services Back-end. On top of that, most banking systems have a very typical structure (an example of which we depicted earlier in Section 3.2), making it easy to define and reuse a generic meta-model. In Figure 6 we show how the indirection allows reuse of the transformations to Mega. If another banking system (such as portfolio or order management) has to be documented, we can reuse the documentation generation. For three systems this results in four arrows instead of six. Similarly, if another output format is chosen by the organization (e.g., UML implemented in another tool), we can reuse all the fact extraction work.

Concretely, we learned that it is good to use an indirection that splits up between the fact extraction and the documentation generation phases. The indirection brings along two concrete advantages. First, the extra effort it takes to implement such indirection pays off fast as the indirect architecture increase the reuse opportunities both within one

Figure 4: Point-to-point versus indirect transformation for one system

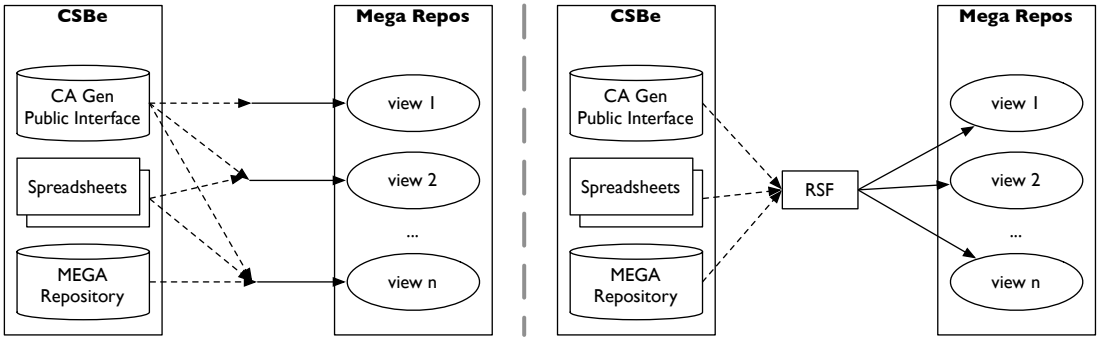

Figure 5: Point-to-point versus indirect transformation within one system

system and among several systems. Second, it allows for using a generic meta-model (such as RSF) in which the extracted facts can be expressed. As this meta-model is supported by existing analysis tools (like CrocoPat), one can take advantages of those tools for different purposes: analysis, system comprehension or redocumentation itself.

4.6 Query engines are not reporting tools

In our redocumentation effort, we made use of the CrocoPat tool. We experienced that it allows one to write very concise queries using first-order predicate calculus which execute blazingly fast. Transforming these results into the Mega output format, however, was a tedious task and required a lot of effort. Furthermore, CrocoPat lacks any kind of modularization for the RML scripts. For example, it is impossible to call a script from another script. As a result you either get one very big script doing everything, or you get a lot of duplication in your queries. In practice, the latter often results both in duplicated scripting code and duplicated calculation time.

We learned that a good query engine is a big plus during redocumentation, but that query engines alone are not enough. To comfortably generate the correct output format a good reporting tool is necessary to complement a query engine.

4.7 Be opportunistic when extracting facts

We choose to keep the fact extraction very lightweight. We extract most of the CA Gen facts using SQL queries, limiting the information to what is available in the public interface of the host encyclopedia. All other necessary facts we extracted using lexical techniques [10], for example, when processing already existing documentation. Performing a full syntactic analyses would not have been cost-effective, especially because this is a one-time operation, and no other systems use the CA Gen environment.

In general, we promote opportunistic fact extraction, i.e., looking for tools in the development and build process that already do much of the hard work, and merge all these bits and pieces of information into a coherent model.

4.8 Not everything can/should be automated

We started out with the premise that as much as possible should be automated. Of course, some factors prevent full automation. Firstly, some type of information might be necessary in the documentation but not available as facts to be extracted. For example, semantic information on the purpose of some functions or how they represent real-world objects is not necessarily available in the code or any other documentation. This kind of information requires human interpretation and is, thus, by default not automatable. Secondly, some technical limitations could prevent automation that, in principle, should not pose a problem. For example, in Section 3.3 we explained how the Mega import format does not cover all features of the Mega tool, necessitating manual corrections.

Besides the fact that some things are impossible to automate, it is not necessarily desirable to automate all things that are possible. For example, the diagrams we import in Mega lack a layout. They are drawn on the screen using a built-in layout algorithm which is not very human readable. It is possible to define the position of elements on a diagram, which we could use to create our own layout algorithm that would, to some extent, be more human readable. However, the investment of creating such an algorithm is bigger than manually layouting the diagrams on an as-needed basis. Especially because manual intervention will be necessary anyway.

We learn that full automatic redocumentation is not possible. Technical barriers and lacking semantic information necessitate a human in the loop. Furthermore, full automation is not desirable as the cost of automation can exceed the cost of doing it manually: only automate what is cost-effective.

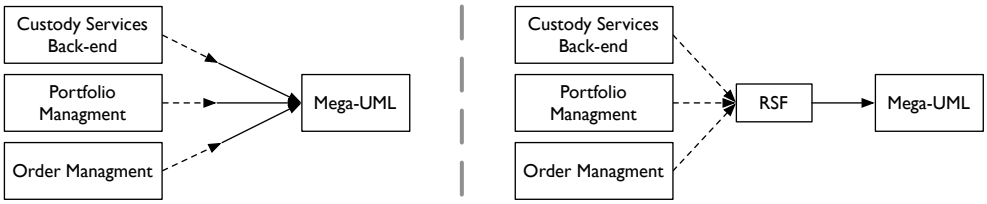

Figure 6: Point-to-point versus indirect transformation for three systems

4.9 Get regular user feedback

Because we base ourselves on the documentation standard which is used throughout the entire organization, it is not necessary to involve the users too much. In this case, users are those who create, use and maintain the documentation. Nonetheless, we opted to include regular user feedback because we made the following observations: (1) not all parts of the documentation standard were documented equally well, (2) mapping the legacy system constructs to the documentation standard was not always straightforward, especially because the legacy system does not adhere to all organizational standards, (3) not all parts of the standard are equally maintainable in practice, so we had to prioritize based on usefulness. (4) not all constructs, as defined in MEGA, were possible to import into MEGA, therefore some todos had to be flagged for manual fixing.

We learn that, although documentation standards are available in these kinds of organizations, it is still best to work iterative and include regular user feedback. This way, one can quickly focus on the most relevant parts, resolve system-specific oddities in consensus and omit irrelevant or harmful parts early in the process.

5. CONCLUSIONS

The availability and quality of software documentation is often unsatisfactory, especially for legacy systems. To overcome this problem one can recover documentation about the subject system using automatic and manual redocumentation. We report about a redocumentation experience from the field which provides insights on what to automate (and what not to) and how to accomplish this.

We applied redocumentation on the Custody Services Back-end of a large Belgian bank, which uses an enterprise-wide documentation standard based on UML implemented in MEGA (a general purpose documentation tool which is highly customizable). As the maintenance of this system is in the process of being outsourced, the organization needs to make sure that the documentation is up-to-date. Currently, there is very little documentation available about the system, and most of it is contained within the CA GEN environment which will become obsolete. Therefore, we were asked to provide documentation within the MEGA tool. More specifically, we aim to (i) reduce tedious manual effort, (ii) focus on quick wins, and (iii) take into account possible reuse of the technique on other systems.

Our redocumentation process consists of two phases. First, we extract facts about the system. For that, we use existing artifacts such as the source code, already available documentation or from the people working with the system. Second, we transform the extracted facts into the correct documentation format. We adopt an iterative approach, in which we frequently present our findings to the experts. In turn, the experts provide feedback that we use to improve our redocumentation efforts.

Finally, we report on our experience with applying redocumentation techniques in this industrial setting. Most notably, we learned that (1) not everything can be automated, nor should it be, (2) "less is more" also applies to (re)documentation, and (3) using a generic meta-model is not only advantageous for reuse in later projects, but it also pays off for an isolated redocumentation effort. Both academics and practitioners can take advantage of the best practices represented by the lessons. While the latter can use them to improve the usability, quality and maintainability of the documentation they generate, the former can use them to improve their (re)documentation tools and methods.

6. ACKNOWLEDGMENTS

This work has been carried out in the context of the 'Migration to Service Oriented Architectures' project sponsored by AXA Belgium NV and KBC Group NV. Additional sponsoring by the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy, project *MoVES* and the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) as part of the project "Optimization of MPSoC Middleware for Event-driven Applications" (OptiMMA).

7. REFERENCES

[1] R. E. Ahmed. Software maintenance outsourcing: Issues and strategies. *Computers & Electrical Engineering*, 32(6):449 – 453, 2006.

[2] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM.

[3] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31:137–149, 2005.

[4] T. Burns and G. M. Stalker. *The Management of Innovation*. Oxford University Press, 1994.

[5] D. Chan, F. Fontan, S. Rosati, and D. Russo. The securities custody industry. Technical Report 68, Occasional Paper Series from European Central Bank, August 2007.

[6] J. Cheesman and J. Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[7] E. Chikofsky and I. Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, Jan 1990.

[8] B. Du Bois, B. Van Rompaey, K. Meijfroidt, and E. Suijs. Supporting reengineering scenarios with fetch: an experience report. *Electronic Communications of the EASST*, 8, 2007.

[9] H. M. Kienle and H. A. Müller. The rigi reverse engineering environment. In *1st International Workshop on Academic Software Development Tools and Techniques*, 2008.

[10] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.

[11] J. Tas and S. Sunder. Financial services business process outsourcing. *Commun. ACM*, 47(5):50–52, 2004.

[12] A. van Deursen and T. Kuipers. Building documentation generators. In *Software Maintenance, IEEE International Conference on*, page 40, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[13] J. Van Geet and S. Demeyer. Reverse engineering on the mainframe - lessons learned from "in vivo" research. *IEEE Software Special Issue on Software Evolution*, July 2010.

[14] V. Vinekar and C. L. Huntley. Agility versus maturity: Is there really a trade-off? *Computer*, 43:87–89, 2010.