MOMOCS

Model driven Modernisation of Complex Systems

PROCEEDINGS OF MODEL-DRIVEN MODERNIZATION OF SOFTWARE SYSTEMS 2008

ORGANIZED BY

Luciano Baresi
Politecnico di Milano (Italy)

Jean Bézivin
INRIA (France)

EVENT CO-LOCATED WITH ECMDA

MOMOCS
methodology and related tools for fast reengineering of complex systems

Table of contents

Fine-Grained Historical Analysis of Software System Evolution: A Framework and Examples of Applications

Massimiliano Di Penta

Department of Engineering, University of Sannio
Via Traiano, 1 - 82100 Benevento (Italy)
dipenta@unisannio.it

Abstract. Software maintenance and evolution is one of the most critical phases in software life-cycle, and surely the most effort-consuming one, taking up to 80% of the total development effort. About 50% of this effort is required to analyze and comprehend existing software, for which very often the only, reliable source of information is the source code. This explains the growing interest that software reverse engineering has gained in the last 20 years. Traditionally, reverse engineering techniques are based on static analysis of source code and of binaries, sometimes combined with dynamic analysis based on information related to system execution.

Nowadays, the availability of data from software repositories - i.e., Concurrent Versions Systems (CVS), SubVersioN (SVN) and bug tracking systems (e.g., Bugzilla) provides the possibility of analyzing the evolution of software systems from a different perspective, where the traditional static and dynamic analysis are combined with a third dimension, characterized by the analysis of historical data.

This talk overviews a framework aimed at extracting and combining facts from different kinds of software repositories, and overviews case studies in which the framework has been applied, specifically, for analyzing the evolution of clones, design patterns, crosscutting concerns and vulnerable instructions.

Demands for Modernization - Survey Analysis

Karsten Tolle[1], Evaldas Verselis[1], Viktor Paland[2] and Simone Hannemann [2]

[1] Johann Wolfgang Goethe-University Frankfurt am Main, Germany
{tolle, verselis}@dbis.cs.uni-frankfurt.de
[2] SIGS-DATACOM GmbH, 53842 Troisdorf, Germany
{viktor.paland, simone.hannemann}@sigs-datacom.de

Abstract. To survive in the market your systems need to be adaptable to changes. To do so you either redevelop it or you modernize it. Especially for complex systems a redevelopment is mostly not possible due to time and money constraints. By the growing functionality and complexity of existing systems during the last years, this is true for many systems. Therefore the market for modernization is growing and will become even more important in future. This paper provides an analysis of a survey we did in order to understand the existing demands within this modernization area. The survey has been performed in context of the European project MOMOCS (MOdel driven MOdernization of Complex Systems).

Keywords: Modernization, SOA, MDA, RUP, XP.

1 Introduction

The global competition of nowadays markets is increasing the pressure to be able to react on changes and new requirements much faster then years before. The companies can not risk any more to do business as usual. However, existing systems (hardware and software) are in place and are sometime tidily coupled with business processes. The goal is to reuse and adapt these legacy systems to the new needs. This would be much faster, cheaper and saves knowledge and done investments so far compared to recreating the system from scratch. For the same reason we currently see a hype for areas like Service Oriented Architecture (SOA) or Model Driven Architecture (MDA). If you have those architectures in place, modernization should be easier. However, in order to have this you already need to change modernize your system.

For the software development process we can find different methodologies from agile once like eXtrem Programming (XP) to the more rigorous and complex Rational Unified Process (RUP). Within the EU-Project MOMOCS (MOdel driven MOdernization of Complex Systems) [1] we developed a methodology called eXrem end-User dRiven Process (XIRUP) [2] that takes the advantages of existing development processes and addresses the peculiarities for modernization. On top we developed tools supporting this methodology mainly regarding the steps of generating

the models reflecting the existing system and transforming it to the modernized to-be model. A more detailed description of our aims can be found in [3].

In order to ensure that we are producing our methodology and tools for the needs of the modernization market, we conducted a survey in form of an online questionnaire. In the next section 2 of this paper we describe details about how we performed the survey as well as the questions we asked and our intentions behind. In section 3 we try to understand who participated the survey in order to classify the feedback and our conclusions out of it. Section 4 provides results by looking at the numbers received. Finally we summarise and draw our conclusions in section 5.

Please note that this paper is an improved and squeezed extraction of a longer survey analysis we did in December 2007 [4].

2 Survey background

On 13th of September 2007 an email asking the people to fill an online questionnaire was sent by SIGS-DATACOM to 45.100 of their contacts in the IT area. The mail was received by 43.700 contacts while 1.400 could not be delivered (Hardbounces 3,1%). From those received, 10.060 opened the mail and 557 had a look at the survey (pressed the link to it). Finally 193 of them took the time to fill the form and participate to the survey.

Note: Compared to numbers of similar campaigns this is a significant positive result that shows a high interest of people in the given topic. This is especially true when you consider that some of the questions should be answered by filling text fields. Even here we received amazingly many and detailed answers.

The survey consisted out of 13 questions plus sub questions. The questions and our intentions behind are listened in the table below.

Table 1: Overview of the survey questions.

Nr.	Question (alternative answers in brackets behind)	Intention
0	How many modernization projects did you realize? (0, 1, 2, 3-5, more)	To understand how experienced the survey participants are in the modernization area.
1	Duration of the Project? (1-6, 6-12, 12-24, more / in months)	In order to cluster the answers regarding project durations.
2	Industry?	To see if there are differences regarding the industry domain.
3	What did you chance? (SW, HW or technical equipment, other)	Finding out if modernization currently is restricted to software or not.
4	Modernization Goal?	To understand the reasons for modernization.
5	Number of persons involved?	For clustering answers.
6	Outcome? (great success, success, partial success, failure, disaster)	In order to put the other answers into context and to find the currently strong or weak points.
7	Your role within the project?	To understand what kind of participants we had.
8	How was the critical impact of the project on your enterprise?	To see how critical modernization is seen for the enterprises.
9	Used standards, methodologies and techniques? (Pair Programming, Test Driven Development, regular Project meetings, UML, Web Services, MDA, others)	Finding out what is currently used.
9a	Which standard, methodologies and technique would you use again?	To understand the usefulness of used standards, methodologies and techniques.
9b	Which software program did you use for the modernization project?	To see what programs are currently used.
9c	Would you use this software program again?	& and if they are found useful.
9d	Which steps of modernization should better be supported by software programs?	Where are the weakest points of current tools?
10	What worked well?/What would you repeat next time? (Text field)	In order to learn from it best practice.
11	What did not work well?/What would you change next time? (Text field)	In order to learn from it no goes.
12	Additional comments? (Text field)	To retrieve additional information we might have not thought of.

3 Who participated to the survey?

Let us try to understand who participated to this survey. This will indicate how representative the given feedback is and to which situations the feedback can be applied to.

The very first question was intended to understand how experienced the participants are in the field of modernization. As you can see in Figure 1 over three-quarter had more than one modernization project and still about 60% had more than 3. Some of the participants that answered under more than 5 indicated over 50 projects. Since some also mentioned a time span, e.g. across 3 decades , we decided to change the label in Fig.1 to Other .

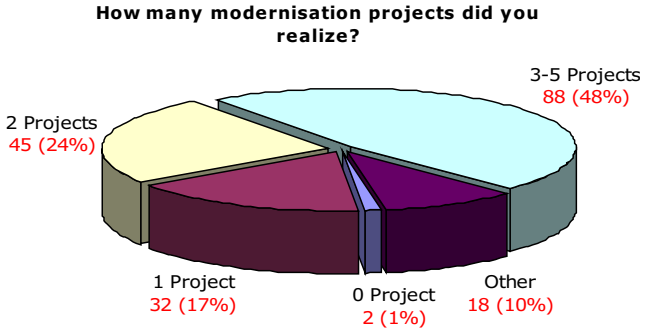


Fig. 1. The level of experience of the survey participants.

Survey question no. 7 where asking the role within the last modernization project. This indicates on which hierarchical level the participants of the survey are. It is worth to note that participants could enter different roles to describe there function more precise. Under others the most often function mentioned where consultant however, we also received as answers: supervisor, analyst, business analyst, software-quality inspector, methodology leader and DB advisor.

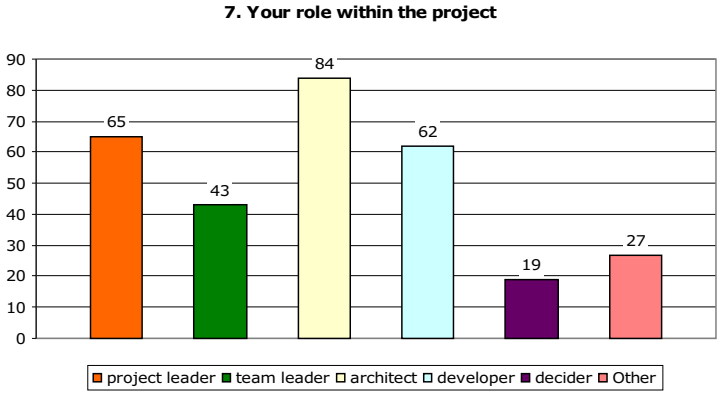


Fig. 2. Trying to understand the management level of the survey participants.

As you can see in Figure 3 below all main industry branches are covered. Under others we received answers like: aerospace, hospital IT, oil and gas, nuclear power plant, consulting, ICT and more.

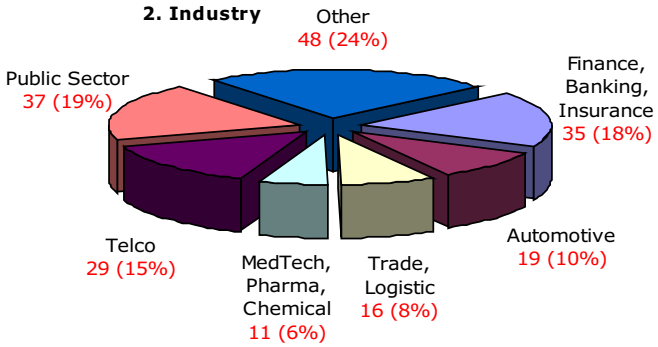


Fig. 3. Overview of the businesses domain the participants are related to.

Last but not least we asked the participants to provide their e-mail address in order to provide them the analysis of the survey. 152 of the 193 participants (~79%) where interested in the survey results and provided their e-mail addresses. Of course the single e-mail addresses are under privacy protection. However, by just looking at the country code and company related parts of the e-mails, we see:

- The distribution in regards of countries corresponds to the European focus of the SIGS-DATACOM database.
- We received feedback from a wide mixture of company sizes, from global players like: T-Mobile, Audi, Volkswagen, EDS, Bosch and Airbus (and others) down to SMEs.

4 Analysis of received answers

For us one of the most important goals was to see if the objectives of MOMOCS fit to existing market needs. Especially because due to the time constraint of two years for the project MOMOCS concentrates on tool development only for the following two areas:

- Modeling of the existing legacy system.
- Transformation of a model to receive the modernized model under considerations of given constrains.

The other steps that are needed for the complete XIRUP methodology to go from legacy system to the implementation and deployment of the modernized one are handled with existing tools or manually. The question: *Which steps of modernization should be better supported by software programs?* was therefore essential for us. As possible answers (multiple selections where allowed) we provided:

a) Understanding and modeling of former programs
b) Dividing into logical components
c) Transforming and verification up to new requirements

d) Simulation
e) Transformation from model layer to development
f) Software-distribution, installation and configuration (deployment)

As we see from the feedback of the participants, most of them search for a tool supporting the understanding (sometimes also called: relearning) of their existing system and to model it. Also the second main area MOMOCS tools are supporting is highly requested. Looking at the numbers, one can also point out that all areas where selected quite often.

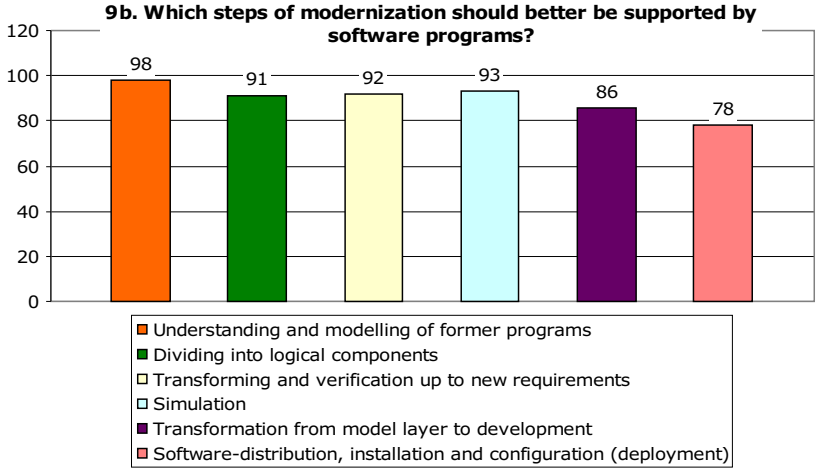


Fig. 4. All modernization steps are requested to be better supported by software tools.

When we look what are main areas for modernization we find that three-quarter are on Software. Predefined answers where Software, Hardware, Technical Equipment and others. Under others we received many entries indicating a mix of software, hardware and/or technical equipment modernizations. Another often mentioned and important issue in this respect is the modernization of processes.

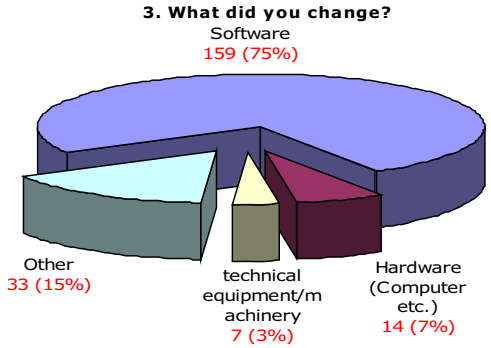


Fig. 5. Areas of modernization.

In our question 4 of the survey we asked for the modernization goal. Customization of new requirements is in fact very generic; it therefore was selected

by nearly each of the participants showing that the answers where not made just random.

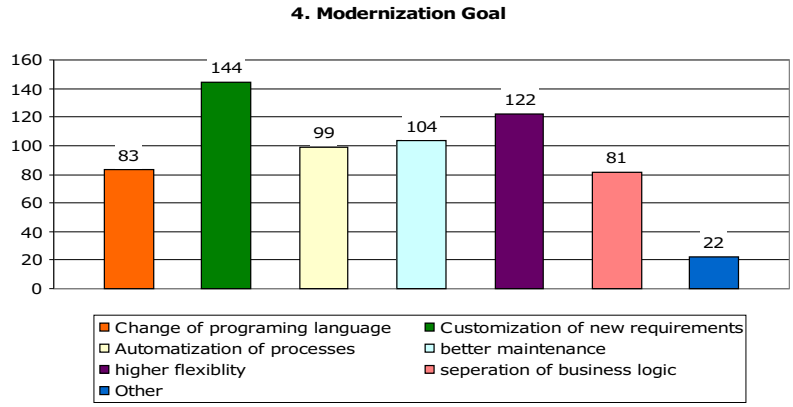


Fig. 6. Objective for the modernization.

Important is also to understand how important the modernization projects and their goals are seen by the involved persons. As you can see in Figure 7 three-quarter indicated the modernization project as essential or very important for the company.

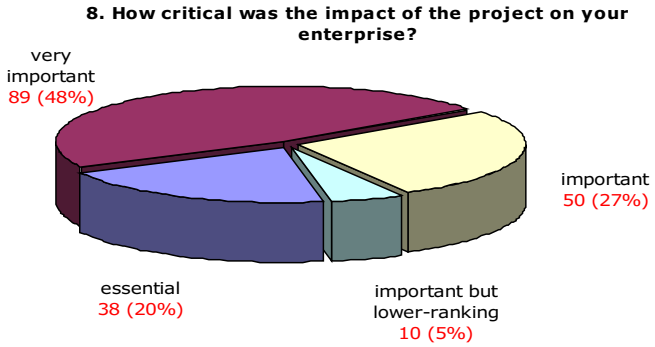


Fig. 7. Relevance of the modernization project for the enterprise.

Another very interesting question was about the standards, methodologies and techniques (SMT) used question 9. Of course we needed to restrict the number of possible choices here. In the next question 9a we asked if they would use those SMTs again. A high number of the participants indicated in question 9a SMTs that they did not selected in 9. As we see in the numbers all SMTs have a higher mark in 9a than in question 9.

Our interpretation is that the participants indicated also their plans for next projects. This indicates that in future standards, methodologies and techniques will play a more important role.

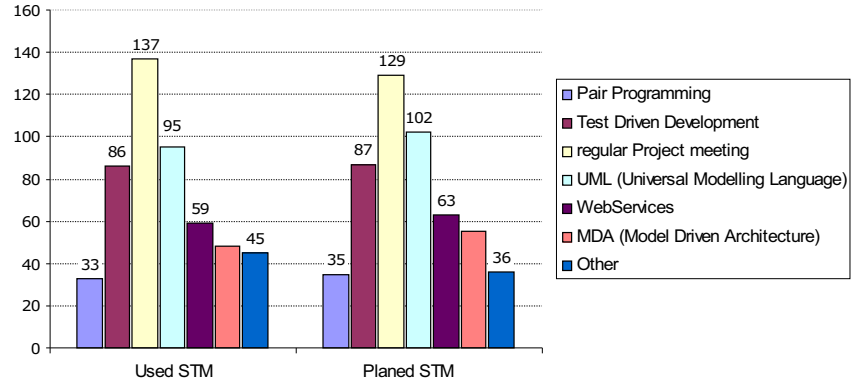


Fig. 8. Used Standards, Methodologies and Techniques (SMT). Left side: Used during the last project; Right side: Planed to be used for the next project.

The results from question 9a do not show significant differences when you cluster the feedback regarding the duration of the project. Comparing the feedback regarding the numbers of involved persons in the modernization project some changes can be seen. Below we clustered the answers to question 9 regarding the number of persons involved into the project (3 Persons and over 50 Persons both groups had 12% of all participants, therefore, the numbers can directly be compared).

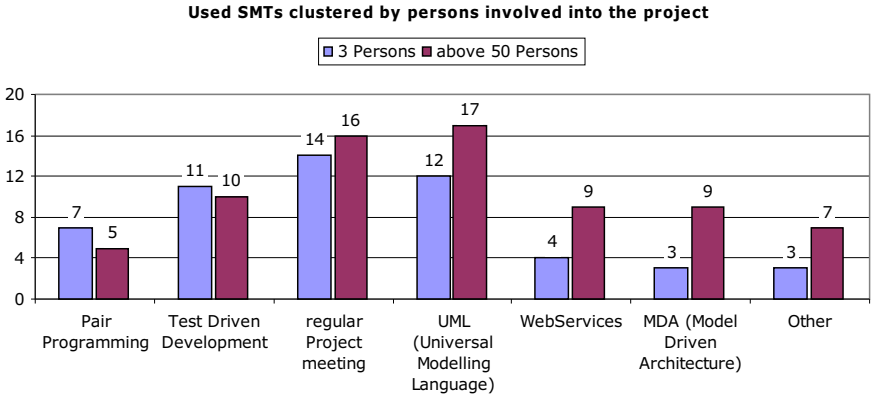


Fig. 9. Used SMTs in relationship to numbers of persons involved into the project, broken down for 3 Persons and above 50 Persons selected.

In question 9b we asked what tools had been used. We did not provide predefined answers for this question in order not to influence the users.

As a result we received very broad list of about 93 tools. Considering that only 152 participants filled this question and additional 20 of these entered generic answers like: C, C++ or Java, we can state that there is currently no tool that is dominating the modernization market.

Remarkable was the result that 90% of the participants would use the software again they used during the last project. We looked at the 10% of participants that do

not want to use the program again, however, we could not find any tool mentioned with a significant frequently.

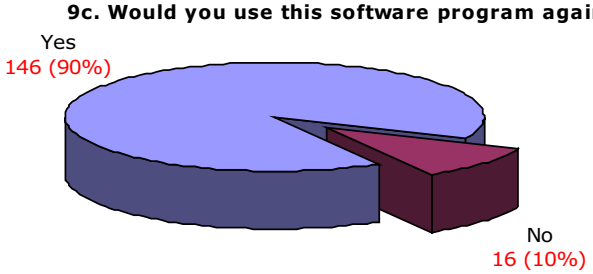


Fig. 10. Most programs should be used again.

5 Interpretations, Conclusions and Summary

As illustrated in section 3, we received feedback from a wide range of domains and companies, manly from highly experienced persons. Together with the high number of participants we can state that the shown results give a representative view of the current situation.

Here our conclusions and summary from the numbers presented in section 4:

1. MOMOCS tool development touches those areas with the highest request for better tools. However, it is important to stress that all areas are requested.
2. Most participants related modernization with software modernization . However, we need to point out that the question in this respect was not well designed. We did not give the possibility to combine the different areas. As we have within MOMOCS also the case of Siemens where the main focus is on technical equipment . We therefore believe that independent of the survey results also for those kinds of modernizations a high demand for tool and methodology support exists.
3. Modernization is critical for many companies. It is therefore worth to invest time and effort to it.
4. Standards, Technologies and Methodologies are getting more and more important. This is especially true for bigger projects with more persons involved.
5. There is no dominating tool in the market. For sure some consolidation will happen and new tools will appear. However, this will take some time when we consider that 90% of participants plan to reuse the tools they currently use. On one hand side this ensures some stability once the tool is in the market, on the other side it is difficult to enter at first place.

In section 4 we presented only some of the numbers, clustering and diagrams we did, highlighting the most important issues. Due to the limited size of this paper, we could not present all details. A more exhaustive overview of the survey feedback can be found in [4]. Here a list of conclusions we made there:

- Long running projects (over two years) mainly belong to the essential and very important projects.
- Short running projects (1-6 Months) have a higher chance to succeed than long running projects (more than two years).
- The more persons are involved into a project, the higher is the risk that the project fails.
- Long running projects tend to have more persons involved.

Conclusions taken from text field analysis:

- Test Driven Development (TDD) is very well accepted.
- Agile methodologies are more and more used, however, developers and management need to accept and support it.
- Understanding and handling requirements is difficult, especially because requirements are not always stable over time. It is also a communication issue between client and contractor. -- might be that the TDD helps in this respect.
- There is no golden path that fits every project!

Acknowledgments. The work presented in this paper is partly funded by the European Commission through the MOMOCS STREP IST-FP6-034466. The authors are solely responsible for the paper's content. Many thanks to the MOMOCS team for supporting this work by feedback, comments and various discussions.

References

1. MOMOCS Project Homepage: www.momocs.org
2. MOMOCS Consortium. XIRUP Methodology Specification. Technical Report, MOMOCS Project, June 2007
3. Alessandra Bagnato et al. MOMOCS: MDE for modernization of complex systems. Neptune 2008 (http://neptune.irit.fr/), Paris April 2008
4. MOMOCS Consortium. Modernization Project Evaluation available online under: http://www.viewzone.org/momocs/evaluation.pdf, December 2007

Toward Agile Open-Source Framework Outsourcing

Ilie Şavga and Uwe Aßmann

Institut für Software- und Multimediatechologie, Technische Universität Dresden, Germany,
{ilie.savga|use.assmann}@tu-dresden.de

Abstract. Outsourcing framework functionality to an existing commercial or open-source component may considerably reduce costs required otherwise for framework maintenance. However, outsourcing itself is a resource-intensive process with high failure risk. To reduce the latter in the context of outsourcing Java frameworks, we propose REFRAME—an agile strategy of outsourcing to open-source components. The strategy enables an incremental outsourcing and reuses existing open-source tools at its different stages.

1 Introduction

Although reusing an own framework helps a company to quickly develop quality applications, maintaining such a framework is time- and resource-intensive. Facing the appearance of new commercial and open-source frameworks with richer functionality, the company must invest more in maintenance to keep its own framework competitive. The costs required may quickly become an overkill, especially for small- and medium-size companies with limited resources. Alternatively, such company may consider framework outsourcing: delegating the whole or, at least, a part of the current framework functionality to another commercial or open-source component. [1]

For instance, the proprietary Java framework of one of our industrial partners has been successfully used in the last several years to develop a number of Web applications. However, the costs of its maintenance became recently unacceptable and the company took a strategic decision to outsource the framework.

There are several prerequisites to successfully outsource their framework. First, existing applications must be kept running during outsourcing. A *big-bang* approach, when applications are stopped and need to wait for the final framework version, is hence unacceptable.

Second, the business value of the existing framework should not be lost and all important business decisions must be preserved in transition to the outsourced version. However, the whole development team that initially developed the framework left the company and little knowledge about the code infrastructure and implementation decisions is available. As a consequence, most of the business model's concepts are hidden in the implementation.

Third, and most important, outsourcing must be performed with limited human resources available. Besides others, this prohibits the company from hiring a consulting company or buying one or more commercial components to perform outsourcing.

[1] Throughout the paper we use the more general term software component and more specific term software framework interchangeably, making explicit distinction, if needed.

Addressing these conditions we propose REFRAME—an agile strategy for open-source framework outsourcing. The key idea is to systematically recover valuable business assets from the existing implementation and then use them to outsource the framework to one or more open-source frameworks. At the same time, existing applications are preserved running despite of the changes to the framework. Moreover, all outsourcing activities are performed in an incremental way.

More specifically, the strategy consists of four main phases:

I: Business model discovery. As the first step in outsourcing, the framework is thoroughly analyzed to demarcate business concepts in the existing implementation using tools for code analysis (e.g., metric tools), manually investigating the code and consulting framework documentation, when available. To make the model discovered tool-processable for the next phases, its concepts are marked in the code with a machine-processable annotation-based language. Annotations are used in the next phases to visualize and transform the code. In fact, the annotation language is a domain-specific meta-model, annotations as a whole represent the (inlined into implementation) model and pieces of annotated code are the model's implementation. The model discovery is done in an iterative manner: as the conceptual knowledge about the framework grows, developers may re-think and update earlier modeling decisions. The annotation language itself may also be enriched, once new concepts are discovered and existing ones restructured.

II: Model-driven framework preparation. Phase I discovers not only the business model, but also numerous problems in the actual implementation. The implementation of concepts may be scattered over the framework. In a single implementation class there may be several, often not closely related concepts tangled. Often duplicated code is discovered. Using annotations of Phase I, Phase II effectively adjusts existing implementation to properly implement the model. In some cases the implementation of some concepts is improved by, for instance, putting together related functionality, renaming classes and methods to follow naming conventions and restructuring the framework's class hierarchy. In other cases, previously "hidden" design decisions are made explicit by introducing new classes and methods, method overloading, and so on. Yet in other cases the functionality considered obsolete or duplicated is removed. This phase results in the framework prepared for actual outsourcing.

III: Model-driven framework replacement. Old code is systematically replaced by new code that uses an open-source framework. When replacing, the business model must be preserved, that is, replacing code implements (at least) the same concepts as the old implementation. In practice, because the framework used for outsourcing offers usually rich functionality, the existing business model can also be enriched and extended in the outsourced version, if needed (which was hard or impossible in the old implementation).

IV: Change-based application preservation. Changes applied to the Application Programming Interface (API) of the framework may break existing applications; the latter cannot either compile or run with the modified framework. To keep applications running during outsourcing, we use change information to create adapters [2, p.139] that are then placed between the framework and existing applications [3].

Adapters protect existing applications by translating between them and the changed framework. The change history is obtained by carefully recording all changes applied in Phases II and III.

Although conceptually distinct, in practice the activities of the four phases considerably overlap. By improving the implementation, code restructuring of Phase II not only depends on, but also influences intermediate results of Phase I leading to a easier concept discovery. Similar, adapters in Phase IV can already be created, when just a part of the framework is restructured or replaced, so that adapters are also built in an iterative, step-wise manner. Even if Phase I is not completely finished, but some concepts discovered are considered stable (no subject to later changes), other phases can be applied for those concepts. This reduces the risk of the potentially bottleneck Phase I for the whole project. All in all, these particularities foster agile outsourcing in an iterative and incremental way reducing the risks of running out of project schedule and resources.

In the rest of the paper we go into details of REFRAME in Sect. 2, discuss challenges we meet in Sect. 3, report on current status of the work in Sect. 4, shortly overview related work in Sect. 5 and conclude in Sect. 6.

2 Insights into REFRAME

REFRAME has several distinguishing characteristics we describe in more details in this section.

Eclipse-centric activities. To maximally reuse existing tools at the different stages of outsourcing, REFRAME is centered around Eclipse technology [4]. Eclipse plugins are used for code visualization, browsing, analysis, transformation and change history recording. The annotation language is a combination of Java 5 annotations and Eclipse-specific TODO-like prefixes that can be automatically processed in Eclipse. Tools used for adapter generation are also adjusted to be reused from Eclipse.

Model-discovering code analysis To improve the quality and decrease the risk of highly demanding business model discovery, we combine several sources of evidence to understand existing implementation. We consult (barely available) framework documentation to understand how the framework is designed. We also consult documentation for application developers and interview them to understand how the framework is used and which problems of its usage exist. We investigate massively the framework's source code. Moreover, applications are investigated as well to discover usually missing framework functionality that had to be repeatedly implemented in applications.

Manual investigation is guided by tools for code analysis. We use Eclipse's metrics plugin [5] to find code "bad smells" reported, for example, by high coupling, low cohesion or duplication of code. When we know more about a particular concept or group of concepts, we use .QL [6] (an SQL-like language executable on Java code bases) to query the framework about the concept's implementation.

Modeling in patterns Often concepts and their relations discovered in Phase I can be modeled in several alternative ways. In such cases we tend to think and model in terms of design patterns, either general [2] or specific for the framework's domain. In such cases the annotation language is extended and adjusted (using Hungarian notation [7]) to reflect the patterns in annotations.

Transform by refactorings While restructuring and replacing framework implementation, we want to neither loose important existing framework functionality nor break applications that use the framework. Therefore, whenever to apply a change, we try to express it in terms of *refactorings*—source-to-source program transformations that preserve system behavior [8]. Common examples of refactorings include renaming classes and members to better reflect their meaning, moving members to decrease coupling and increase cohesion, adding new and removing of unused members. Effectively, a framework refactoring modifies only the code's external appearance, but not its functionality that is to be preserved during outsourcing. Although a single refactoring is usually a small structural change, it has been shown that even large architecture changes can be achieved by applying a sequence of refactorings [9].

Since many concepts in Phase I are modeled as design patterns, one of the main activity during framework preparation (Phase II) is refactoring to pattern [10]—structural changes of the implementation to follow a certain design pattern. Refactorings also are the main activity during framework replacement (Phase III). For example, replacing a method call to an equivalent "outsourced" method, refactoring a method's algorithm to use functionality of the new framework or aggregating old plain types into a new framework type can be modeled as refactorings.

Because Eclipse refactoring engine is designed extensible, we can implement a new refactoring operator in case no appropriate operator is available for a particular change. In such a way we end up in an executable refactoring catalog specific for framework outsourcing.

Gentle migration. As mentioned, we cannot migrate existing applications in one big step (big-bang migration), because this would require either to stop them running for the whole time of performing outsourcing or to maintain in parallel the old framework. Moreover, big-bang migration has a high failure risk requiring to apply many complex changes at once. Instead, we keep applications running during outsourcing by providing adapters that shield the applications from the changes in the framework's API. Applications are migrated step-wise, synchronously with the iterative changes to the framework (so-called *gentle* migration [11]).

Because implementing adapters by hand is time-consuming and error-prone, we partially automate their creation. According to Dig and Johnson [12] who studied the evolution of five big software components (including Eclipse), more then 80% of application-breaking API changes are refactorings. Treating refactorings operators as formal specifications of syntactic changes, for all framework API refactorings occuring in Phases II and III we generate compensating adapters [3]. Refactoring history is by default recorded in Eclipse and comes "for free". Applications remain binary-compatible, that

is, they link and run with the new framework version without recompilation. Remaining API changes beyond refactorings are adapted manually.

Once the whole outsourcing is finished, applications can be updated invasively (i.e., changing directly their implementation) to use the latest framework version, so that no adapters are needed anymore. Invasive adaptation can also be considerably automate based on the refactoring history [13].[2]

3 Further Discussion

Although we aim for reducing the costs and risks of outsourcing, it still remains a highly demanding development process.

Outsourcing as white-box modernization Framework outsourcing is white-box modernization requiring inevitably a deep understanding of framework internals and consequent system restructuring. There is always a danger that developers become stuck in code due to its complexity. The system preparation crucial for the subsequent replacement, is also risky requiring complex, often untrivial code transformations. Moreover, it is not always obvious, which pieces of code should be chosen in a particular increment of replacement. It is important therefore that developers are guided by the business model previously discovered in performing further code analysis and transformation. Explicitly representing business model helps further understanding of the system (in analysis) and guides code transformation by demarcating pieces of code with related functionality.

Incremental Deployment Preserving old framework, while deploying and testing new framework versions (parallel deployment) guarantees a fallback mechanism in case of a framework failure. However, parallel deployment is often too expensive for a small company. Alternatively, incremental deployment of framework consisting of old and outsourced parts reduces costs and development time. Moreover, incremental deployment may also increase the framework acceptance by its users (application developers), who are forced to use it immediately and, hence, gradually learn the new system [14, p.236]. The risk of system failure is however high and should be reduced by thoroughly testing new framework versions using regression tests and existing applications as acceptance tests.

Internal Adapters Another implication of incremental development is the need for internal adapters to wire together old and outsourced framework parts. Some of these adapters work as wrappers translating between old and new interfaces. These adapters can be generated based on refactoring information, similarly to the ones that preserve applications (discussed in the previous section). In some case, more sophisticated adapters

[2] We use invasive adaptation also to update existing unit tests possibly invalidated by API refactorings.

are required, for instance, *protocol* adapters to translate between otherwise incompatible protocols (e.g., workflows). In such cases, either specific adaptation approaches are required (e.g., [15]) or adapters must be written by hand.

Architecture resemblance The main drawback of the incremental development is that it may lead to the new system's architecture resembling the old one, potentially inheriting existing design problems. That is why system preparation of Phase II is important: it transforms the existing architecture from what it is to what it should be. Subsequent outsourcing may also improve architecture quality by introducing components of a better quality.

Data migration REFRAME focuses on code outsourcing and do not aim for data migration (i.e., from one database to another, more modern database). However, code outsourcing is hardly possible without affecting the way data are stored, retrieved and processed. If the company does want to touch the database to accommodate it to the framework changes (as in our case), this requirement implies introducing a data access layer to translate between old and new data representations.

4 Current Status

To estimate the feasibility of our main goals, see the frontiers of current technologies and set up an outsourcing environment, we perform currently a pilot case study. We use a medium-size Java framework SalesPoint [16] developed at our department for teaching framework technologies. The framework models a purchase-selling business model with associated concepts, such as customer, shop, catalog, and data item. To realize application-specific requirements (e.g, to implement a bike shop or a ticket shop), students specialize framework types. We use the framework together with its Web extension WebPoint (link not available) that is based on Struts [17].

Two master theses investigate different aspects of source code analysis, annotation and refactoring in Eclipse. There is a prototype annotation language developed that is specific for the domain modeled by the framework. Once related pieces of code are annotated, they can be browsed in Eclipse. In addition, several refactorings specific for framework outsourcing are identified and implemented in Eclipse by extending its refactoring engine. An example of a refactoring from Phase II (framework preparation) identified is pulling up to the framework functionality duplicated across applications. This duplicated functionality is previously identified by comparing application code. An example of a code replacement from Phase III is outsourcing SalesPoints's logging facilities to log4j [18]. Another outsourcing activity currently in development is replacing simple serialization mechanism of SalesPoint by Hibernate [19].[3]

[3] Some details about the current status of the case study and related features of Eclipse are at http://141.76.120.114/ReFrame.

5 Related Work

In [20] Chikofsky and Cross discuss how a design can be recovered from legacy systems and what is implied by the process of program understanding. This work is extended by Tilley at al. [21] toward a systematic analysis of system structure. Brodie and Stonebraker [22] describe in details various strategies of software migration and criteria of choosing appropriate migration strategies. A full example of a legacy system transformation is presented in [23]. For a general description of the state of the art we refer to [14].

Among existing modernization projects, perhaps the most relevant to REFRAME is the ESPRIT-funded project Renaissance [24]. This highly ambitious project aimed for systematic way of migrating from old mainframe technologies to object-oriented systems. Similar to Renaissance, REFRAME elaborates on how to systematically modernize a legacy system. However, with limited resources available for modernization, REFRAME focuses exclusively on the agile modernization of Java programs.

6 Conclusion

In this paper we introduced and described in details REFRAME—an agile strategy for Java framework outsourcing. We believe that the key success of REFRAME is that it promotes, even forces, incremental framework changes. Developers performing outsourcing gradually understand the framework by focusing their activities on its parts in smaller increments. Moreover, they gradually learn how to restructure and replace the framework, often from their own mistakes [14, p.16]. Toward the end of the project, this speeds up outsourcing activities.

Overall, the main project milestones are also achieved gradually. Project's tasks should be prioritized to earlier schedule more important yet less demanding tasks. Incremental outsourcing together with an appropriate project schedule guarantee that, if the project runs unexpectedly out of resources, its main goals will still be fulfilled resulting in an operational framework of an improved quality.

References

1. Robotron Datenbank-Software GmbH homepage. `www.robotron.de`
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts (1995)
3. Şavga, I., Rudolf, M.: Refactoring-based adaptation for binary compatiblity in evolving frameworks. In: GPCE'07: Proceedings of the Sixth International Conference on Generative Programming and Component Engineering, Salzburg, Austria, ACM (October 2007) 175–184
4. Eclipse homepage. `www.eclipse.org`
5. Eclipse metrics project. `metrics.sourceforge.net`
6. Semmle's SemmleCode. `semmle.com`
7. Simonyi, C.: Hungarian notation. `msdn2.microsoft.com/en-us/library/aa260976.aspx` (November 1999)

8. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA (1992)
9. Tokuda, L., Batory, D.: Evolving object-oriented designs with refactorings. Automated Software Engineering **8**(1) (January 2001) 89–120
10. Kerievsky, J.: Refactoring to Patterns. Addison-Wesley (2004)
11. Reussner, R., Hasselbring, W.: Handbuch der Software-Architektur. Dpunkt Verlag (2006)
12. Dig, D., Johnson, R.: The role of refactorings in API evolution. In: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), Washington, DC, USA, IEEE Computer Society (2005) 389–398
13. Henkel, J., Diwan, A.: Catchup!: capturing and replaying refactorings to support API evolution. In: ICSE '05: Proceedings of the 27th international conference on Software engineering, New York, NY, USA, ACM Press (2005) 274–283
14. Seacord, R.C., Plakosh, D., Lewis, G.A.: Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices. Addison-Wesley (2003)
15. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM TOPLAS: ACM Transactions on Programming Languages and Systems **19**(2) (1997) 292–333
16. SalesPoint homepage. `www-st.inf.tu-dresden.de/SalesPoint/v3.1/index.html`
17. Apache Struts homepage. `struts.apache.org`
18. Apache log4j. `logging.apache.org/log4j`
19. Hibernate homepage. `www.hibernate.org`
20. Chikofsky, E.J., Cross, J.H.: Reverse engineering and design recovery: A taxonomy. IEEE Software **7**(1) (January 1990) 13–27
21. Tilley, S., Paul, S., Smith, D.: Toward a framework for program understanding. In: 4th Workshop on Program Comprehension, IEEE Computer Society Press (March 1996) 19–28
22. Brodie, M., Stonebraker, M.: Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach. Morgan Kaufmann (1994)
23. Seng, J.L., Tsai, W.: A structured transformation approach for legacy information systems—a cash receipts/reimbursements example. In: Proceedings of the 32nd Annual Hawaii International Conference on System Sciences. (1999)
24. Renaissance project. `www.comp.lancs.ac.uk/computing/research/cseg/projects/renaissance/RenaissanceWeb/index.html`

PSM-to-PIM, a Pragmatic Way

Lionel Vigier, Andrey Sadovykh

Designer Engineer, Softeam, 78000 Guyancourt, France
{Lionel.Vigier, Andrey.Sadovykh}@softeam.com

Abstract. In many businesses, like banking, insurance and in particular aerospace, the life-cycle of information systems lasts for 5 to 20 years. New scalability, performance requirements and targeted platforms make it necessary to modernize current legacy systems. In this article we briefly present a concrete example of a platform migration problem according to the ADM approach. We particularly concentrate on a model abstraction during PSM-to-PIM transformation relying on UML profiles. Finally, we discuss the applicability of our method and the lessons learned.

Keywords. UML, ADM, PIM, PSM, Model Mining, PIM abstraction., Reverse-engineering

1 Introduction

This study is funded by the European Space Agency (ESA) in the frame of the Round-Trip Engineering for Space Systems project. During this study a methodology for systems modernization is elaborated aiming at the technology migration.

The particular interest for the methodology is recovering from the legacy artifacts the functional architecture also referred to as Platform Independent Model (PIM) [Ref 1] in MDA terminology. Indeed, the Architecture Driven Modernization (ADM) [Ref 2] approach proposes to base the modernization activities on the architectural models rather than code artifacts. Currently the reverse engineering methods and tools (SOFTEAM Objecteering, IBM RSA/RSM, BORLAND Together) allow to obtain a UML model, which we call Platform-Specific Model (PSM) [Ref 1] since it is very close to the code. In this context the PSM-to-PIM method should provide an abstraction transformation from the model obtained from the code into a model representing the functional aspect of the architecture.

PSM to PIM is often quoted as something possible, however hard to do because it is human knowledge related. [Ref 3, 4, 5] Depending on the context and understanding of what the platform is or what the business logic is, a component can be either categorized as part of a PIM or not. In this article we present an instrumented approach for PSM-to-PIM transformation involving human experts.

2 PIM extraction workflow

As it was mentioned before, the existing reverse engineering methods allow obtaining a UML model of the code. It is hard to be used for model understanding since the classes extracted are too platform dependent and often unstructured. For instance, a C++ application with no namespaces could result in a "flat model", that is to say all the classes will be at the same level with no packages.

In our approach, extracting a PIM from a PSM is basically deciding whether each model element[1] is part of the PIM or not. It can be a long lasting human process, because of the numerous decisions to take, sometimes involving long discussions.

For the large systems as we had in our study, this task may be spread over a long period (days, weeks, months), thus it was necessary to allow a possibility to go back and change the decision at any time. Therefore, one of the objectives of our method was to work on the PSM in a non-destructive way. In addition the decisions taken for model change should be easily readable and understandable.

2.1 Workflow overview

In the workflow for PIM extraction [Fig 1. on next page], we defined a 3 step iterative process to be repeated until an acceptable PIM is obtained:

1. Mark the elements of the model that should appear in the PIM (human)
2. Extract a PIM from the "*marked PSM*" (computer)
3. Validate the "*candidate PIM*" (human)

2.2 *Mark PSM* activity

As it was mentioned in the previous section, the first step consists in analyzing and annotating the model. For the model analysis, the question we ask for each element of the PSM UML model is: "Should it belong to the PIM?". We often get one of the three answers below:

- "It is part of the PIM"
- "It doesn't belong to the PIM"
- "I don't know, we'll discuss that later"

A convenient approach could have been to mark the classes that should remain in the PIM and extract them. Unfortunately, we could have forgotten a class had been stated as not belonging to the PIM over time. As we want to keep a trace of the decisions made, the classes that should not remain in the PIM must also be marked. This way we can see the progress of the process.

This process is supported by the *marker* tools developed for this purpose. There are 3 of them: the first one is used to mark an element as part of the PIM, the second one is used to exclude an element from the PIM and the third one removes any mark on any element.

[1] association, attribute, class, operation, package

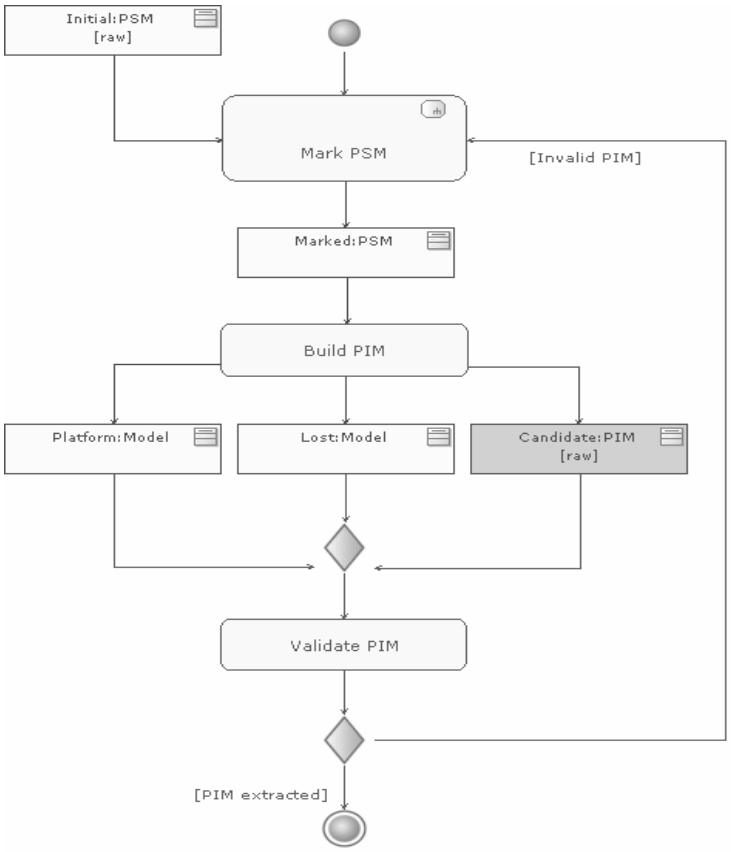


Fig. 1. PIM extraction workflow – UML Activity Diagram

2.3 *Build PIM* activity

The "PIM Builder" function works in a non-destructive mode which means that marked elements from the *initial PSM* are duplicated and not moved to a different model. This way, the *initial PSM* is completely preserved and not modified.

The PIM builder parses the *marked PSM* to generate a *candidate PIM* to be validated further. Two additional models are created: the *platform model* containing the classes to build the original PSM and the *lost model* containing everything that has not been marked, as depicted in Fig. 1. According to the information set with the markers, each element from the *marked PSM* will be created in the appropriate target model. These two additional models are useful to validate the PIM, as explained further.

2.4 *Validate PIM* activity

Obviously the most important model to check is the *candidate PIM*, it's potentially the PIM we will keep in the end. The two other models are here to enhance comprehension about everything that doesn't appear in the PIM. This way, we concretely see collateral damage that could have been missed when looking at the *marked PSM*.

The aspect we found useful in our study for PIM validation are the following ones:

- Check that everything in the *candidate PIM* should belong to it.
- Check that nothing in the *platform/ lost models* should belong to the PIM.
- Analyze relations between the PIM and the two other models.

Of course, the more accurately the model has been marked, the more readable the three models are. In particular, many unmarked associations and multi marked classes can generate a real mess, this is a symptom that we need at least one more iteration.

3 Model markers implementation

We strongly rely on Objecteering customization facilities based on UML2 Profiles. This allows adding icons and semantic information to model elements using stereotypes.

3.1 Colour conventions

As mentioned earlier in this article, some of the native icons will be customized to enhance the readability of the model. We agreed to use colour conventions to visibly mark the model. We distinguish two kinds of elements according to the number of states they can have when marked.

Some elements can allow one and one only mark, that is to say they can be marked as belonging to the PIM or to the *platform model*, but not both. From the model tree coloration point of view, these are the leaves.

Table. 1. Marked elements with 3 possible states

States	Graphic used
1. Not marked	Native icon (yellow)
2. Marked as belonging to the PIM	Blue icon
3. Marked as belonging to the Platform	Purple icon

Some elements allow several marks because of the elements they contain. From the model tree coloration point of view, these are the nodes.

Table. 2. Marked elements with 7 possible states

States	Graphic used
1. Not marked	Native icon (yellow)
2. Marked as belonging to the PIM	Blue icon
3. Marked as belonging to the Platform	Purple icon
4. Marked as belonging to the PIM and to the Platform	Blue and purple icon
5. Marked as belonging to the PIM with remaining unmarked elements.	Blue icon with a warning
6. Marked as belonging to the Platform with remaining unmarked elements.	Purple icon with a warning
7. Marked as belonging to the PIM and to the Platform with remaining unmarked elements.	Blue and purple icon with a warning

Of course, the tricky classes are the ones with multiple marks. In the case study presented later in the article, the classes/packages with multiple marks were, in fact, wrappers for external libraries (like ftp or xml) or system related (such as threads). This is a sign of a good architectural structure. On the contrary, the presence of lots of mix classes (PIM and Platform at one time) throughout the model would have highlighted a poor architecture.

3.2 Rules applied while marking the model

Our approach is to preferably mark packages or classes. We first propagate the mark on the elements under it (recursively), and then update the targeted class or package and its owner, "and so on". Finally, the user has to browse the model and deal with the remaining unmarked associations.

Operations and Attributes. They have 3 possible states [Table 1].

They will most often be processed when marking the class they belong to. However, they can be marked directly. Proceeding this way, the class they belong to is updated ("and so on"). This behaviour is discussed for attributes further in the article (cf. Dealing with attributes).

Classes. They have 7 possible states [Table 2].

Classes are marked according to the elements they contain[2]. When marking a class, we first mark some of the elements it contains (operations and attributes) and then we deduce the appropriate mark as depicted in Table 2. A class can still contain associations that are not marked or marked differently (because of the class they are pointing to). It can even contain an inner class containing associations raising the same problem, "and so on".

[2] attributes, operations, associations and inner classes

Packages. They have 7 possible states [Table 2].

Packages are marked according to the elements they contain[3]. When marking a package, we first mark elements it contains (recursively, according to these rules) and then we deduce the appropriate mark as depicted in Table 2. A package can still contain unmarked elements or marked elements. The elements can be marked as PIM, Platform or both (classes or packages).

Associations. They have 3 possible states [Table 1].

Associations are preferably not marked by default, they should be marked directly. Then the class an association belongs to is updated ("and so on"). It is important that the user decides for each class relation if it is part of the PIM or not. However, there is a special behaviour when marking a package: an association between two classes belonging to the package being marked is marked too.

4 PIM builder implementation

During the model separation, each element is created according to its marks, as explained in section 2. Elements with 3 possible states (see Table 1) are created in the corresponding target models. Elements with multiple marks (see Table 2) on them will be created in each target model they have been marked for. As relations involve two classes, they are processed at the end: when all the classes have been created.

4.1 Rules applied for model separation

Associations. They are marked.

Marked associations are created in one and one only of the three target models. Processing unmarked associations is much more complex because they can involve one or two multi marked classes. In these cases, we create as many associations across the three target models as needed.

Generalizations. They are not marked.

Generalizations can involve classes with different marks. Generalizations are created across the three target models for every class created during the first step of the model separation, but they can involve classes with different marks. Imports are processed the same way.

[3] classes, packages

5 Case study results

The case study was provided by the ESA is the File ARChive (FARC) application. It represents a distributed document sharing facility with an advanced version control functionality. It has been written in C++ and relies on several technologies such as CORBA, FTP, XML and ODBC. The aim of the project was to study MDA technologies for systems modernisation and in particular migration of the FARC systems from C++ to Java.

The modernisation process follows two sequential phases: the first one extracts two separate models (PIM and Platform) from the legacy source code and the second builds the new version of the software after a re-design step at the PIM level and the integration of the new technical platform.

The first phase which could be called “Bottom-Up” realises a reverse operation from C++ source code to get a preliminary PSM model which could be considered as a raw PSM. This is the activity covered by the PIM extraction workflow we describe in this article. The second phase which could be called in the same way “Top-Down” starts with a manual refactoring at the PIM level and continues with the technical or platform migration if needed to provide a final PSM model which will be used in the production of the new source code skeleton. As we notice, these two phases are fully compliant with the ADM and MDA approach and provide a really concrete and pragmatic answer to the software modernisation problem.

The figure below provides a global overview for the software modernisation approach proposed this context.

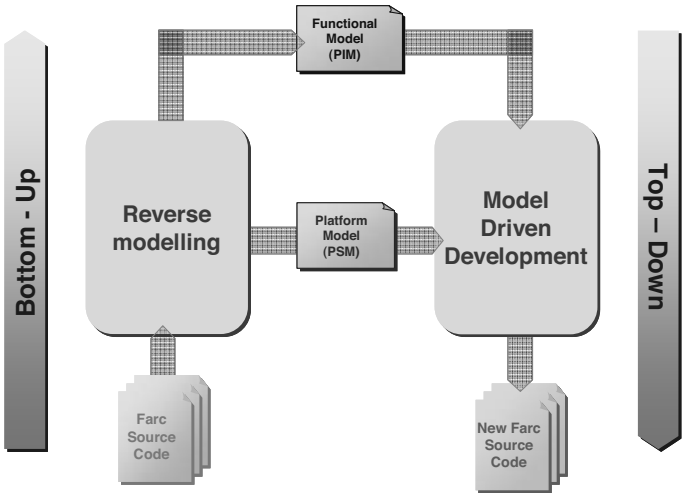


Fig. 2. Modernization Process

The workflow we described above is situated in the Reverse Modelling Phase (c.f. Fig. 2.). We start with the raw PSM model, elaborate marked PSM in order to finally produce the PIM model representing the functional architecture.

The following figures illustrates the results obtained during the method validation. The intermediate marked PSM and the target PIM are depicted below in the Fig. 3.

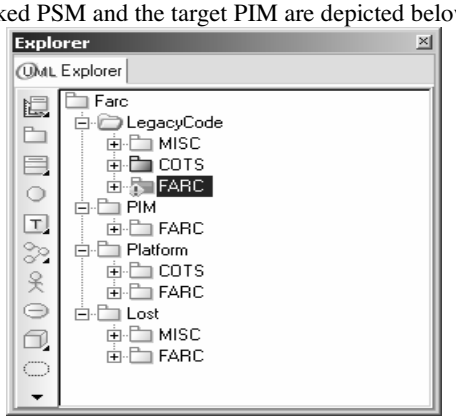


Fig. 3. Case study model, "Farc" application, Objecteering Model Explorer.

The Fig. 3. depicts that the *LegacyCode* package contains the reversed C++ PSM, that is to say the "Marked PSM" described in our workflow. In this package, the *MISC* package is not marked, the *COTS* package is marked as Platform and the *FARC* package containing almost all the application is multiply marked. Consequently, we fond the 3 models generated by the "PIM builder": *PIM*, *Platform* and *Lost*.

The Fig. 4. describes the interpretation of the dependencies in the marking process. The multiple marking is added automatically, since the class depends on other classes included in PIM and Platform.

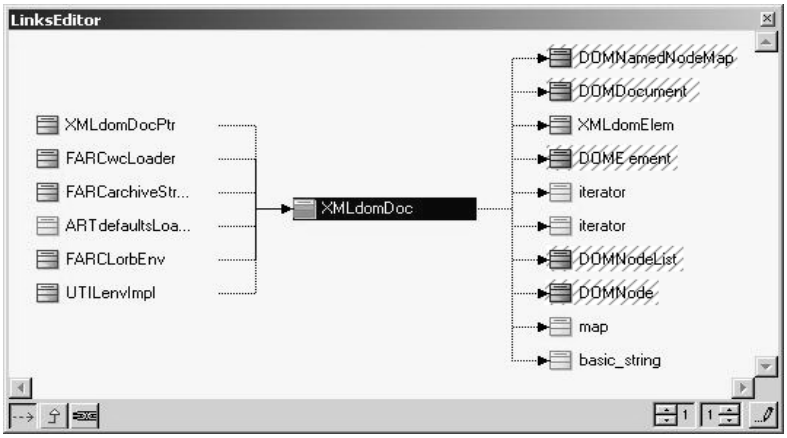


Fig. 4. Dependencies View, Objecteering Fast Link Editor.

The *XMLdomDoc* has multiple marks since it depends on Platform classes (*DOMDocument*, etc.), a PIM class (*XMLdomElem*). There are also unmarked elements to be either lost or marked in the next iteration.

This case study permitted to validate the methodology developed.

6 Feedback

6.1 *Platform model* reusability

The model marking based workflow we describe in this article is part of a larger technological migration funded by the ESA. In this context, they provided a case study which is a C++ application to migrate in Java using ADM and MDA principles.

Applying this PIM extraction workflow, the *platform model* clearly identified the COTS[4] and the Operating System related components[5] used in the application. Having this model speeded up the PSM derivation workflow.

For example, in our study, the C++ XML parser libraries were clearly separated into the platform model. This permitted to identify the need for a substitute library in Java. The same was applicable for the FTP and database connection libraries. In our approach we conserved the links to these libraries as traceability links. This allowed re-linking the new libraries to the extracted PIM classes for migration to Java PSM.

6.2 Reverse engine related improvements

The PSM which is our start point is obtained using Objecteering. Therefore, some points were not considered because of the C++ reverse engine behaviour. Two of them could be enhanced in the future.

The C++ reverse engine creates attributes in the model only when a class contains UML basic types[6] in the code. Otherwise, an association is created in the model. Our approach is to take decisions at a package or at a class level. Therefore, we assumed a basic typed attribute belonging to a PIM class would be part of the PIM too. We should have considered the attributes typed by classes, but we did not, as there were no such attributes in our model. We can still mark an attribute regardless the class it belongs to later. A default behaviour like the one we defined for associations would be more accurate and in the end more reliable (cf. rules applied for the marker).

Having N-ary associations in the PSM has not been considered at all during this study as they are not created by the reverse too. It could change the *marker* tool a bit (technically speaking) but the principle would leave unchanged: the N-ary association would be marked with one and one only mark to have it in the right target model (PIM, Platform or Lost).

[4] Commercial, off-the-shelf (Ftp, ODBC, Xml)
[5] file system, threads, date&time
[6] boolean, char, integer, real, string

6.3 Keeping a trace of the decisions

Marking the model has been enough, for the usage we had. However, we agreed that over time, keeping only the result of decisions (using the last marked PSM) might no longer be enough, because we lose the underlying reasons why it's been marked as part of the PIM. It is especially significant if you consider the model a month later.

Of course, it can be done using descriptions or notes on the model. A dedicated note could have been welcome here, just to record some of the relevant discussions which lead to the PIM extraction. A traceability model might be a solution in this case and we count to consider it for the future studies.

Conclusion

OMG ADM approach is interesting for many businesses, since it allows retrieving not only the implementation sources of the application but also its architecture concepts. Going from the legacy artefacts – source code, documentation, execution logs – to architectural model requires an abstraction transformation. This transformation is a hard topic since it requires expert knowledge about the system and the platform. Thus it is hard to be fully automated.

In this article we presented a pragmatic approach for extraction of PIM model from a C++ PSM model reversed from a source code. The transformation is illustrated with a C++-to-Java migration case study provided by the ESA. The method is implemented with Objecteering CASE tool using UML2 Profiles. In this method some of the parts of the raw C++ PSM model is marked manually, while for the other are automatically marked according to the dependency analysis and rules for semantic annotation.

According the analytic reports in the next 5 years the market of the systems to be modernised will grow dramatically. Our work represents one of the small bricks for the tooling of the ADM concept.

References

1. Object Management Group, Getting Started, Terms and Acronyms http://www.omg.org/gettingstarted/terms_and_acronyms.htm
2. OMG Architecture Driven Modernization Task Force, http://adm.omg.org/
3. Jörg P. Wadsack, Jens H. Jahnke, "Towards Model-Driven Middleware Maintenance", OOPSLA 2002 Conference Proceedings
4. Thijs Reus, Hans Geers, and Arie van Deursen, "Harvesting Software Systems for MDA-Based Reengineering", ECMDA 2006 Conference Proceedings
5. A. Boronat, J. A. Carsi, and I. Ramos, "Automatic reengineering in MDA using rewriting logic as transformation engine", CSMR 2005: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering

Modeling Source Code with Orthogonal Hierarchies

G. Fischer, J. Lusiardi, J. Wolff v. Gudenberg

Department of Computer Science, University of Würzburg
wolff@informatik.uni-wuerzburg.de

Abstract. In this paper we consider various approaches to build up abstraction hierarchies for programming source code. With the help of these models we can transform high-level languages to such an extent that analysis and model driven tools can be applied. The paper thus contributes to the Architecture Driven Modernization framework [12] of the OMG.

1 The Syntax Tree Hierarchy

The source code of a program is not only denoted by a plain text controlled by the concrete syntax of the corresponding language, but it can also be represented as a parse- or syntax-tree. Usually the syntax tree does not contain the particular phrasing of statements or declarations. Identifiers are often kept separate in a symbol table, keywords are replaced by tokens like in compiler construction.
We introduce several layers of abstraction by further reducing language specific information. The top of the hierarchy is a generic, language independent abstract syntax tree GAST.

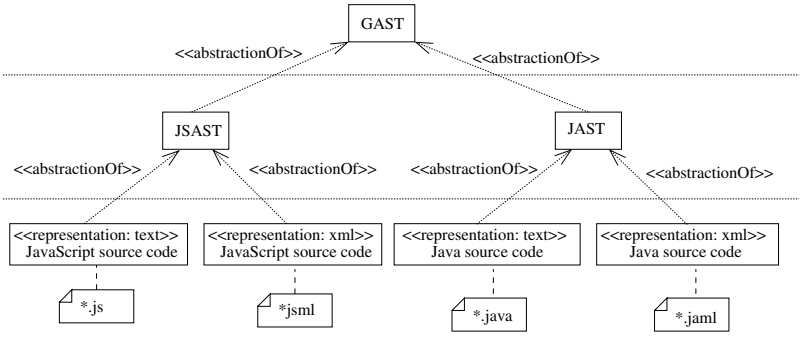


Fig. 1. The Syntax Tree Hierarchy

The three levels correspond to concrete syntax, abstract language specific syntax, and generic language independent syntax. More layers are possible.

An XML representation is commonly used for syntax trees.

1.1 Ehrengast

In recent years we have developed Ehrengast (Eclipse Hosted Representation for Extended Notation of Generic Abstract Syntax Trees), a framework for creation and transformation of XML representations of source texts on different levels of abstraction [3]. We are working with ASTs for the object-oriented paradigm.
Starting from a Java AST obtained by an appropriate parser (eclipse JDT in our case), we produce an XML representation (called JaML) of the source code with layout information [2]. This JaML document is augmented by type and reference information in order to facilitate querying. JaML is already used for highlighting in the Java Online Tutorial [4], and for higher level program analysis [9] e.g.
In a further step, the concrete syntax information is stripped and a Java abstract syntax tree (JAST) is obtained as an XML document.
The currently last abstraction step generalizes the JAST description to a generic abstract syntax tree or GAST definition that can be used as intermediate format for high-level language transformation, at least between object based or object oriented languages. Emphasis is put on the generation of readable and maintainable new source code that will be able to replace legacy codes.
Currently work is in progress on GAST models for JavaScript [6], C++ , C# , and Python.

1.2 Ontological Hierarchy

This syntax hierarchy is an inherent feature for classification in the domain of abstract syntax trees. Following [7] we call it the ontological hierarchy of ASTs. It is one starting point of the ADM (Architecture Driven Modernization of software) initiative coordinated by the OMG (Object Management Group) [13]. One goal of ADM is to provide tools for high-level language transformations in the framework of model driven software development [10]. To reach this goal the hierarchy has to be combined with the model instance hierarchy well known from the UML definition.

2 The Model Instance Hierarchy

Four abstraction layers are introduced in the UML desription. Running programs on instance level M0 are modeled by an M1 instance of a UML diagram that, in turn, is modeled by the M2 UML diagram model. That meta model can be regarded as an instance of the M3 meta meta model MOF. There is a common core that is used by UML as well as by MOF. This common core provides means to divide the definition of the UML into two parts of the library, the infrastructure belonging to MOF as well as the superstructure describing the UML meta-model [11], as known .

3 Combining the Hierarchies

It is attractive to combine the syntax hierarchy with the model instance hierarchy, since the model based definition enriches the set of models that are used in model driven design. A UML like, MOF based model of an AST enables model to model transformations [1] between a UML design model and an AST and, hence supports automatic code generation or application of other MDA tools. The other direction, the generation of a UML model from source code given as an AST produces the documentation from the implementation.
A more direct use of AST is intended in high level language transformation in the field of architecture driven modernization of software. In this framework a MOF based AST model is very helpful. Hence, the OMG initiated a task to define MOF based models for abstract syntax trees [13].

3.1 The OMG Approach

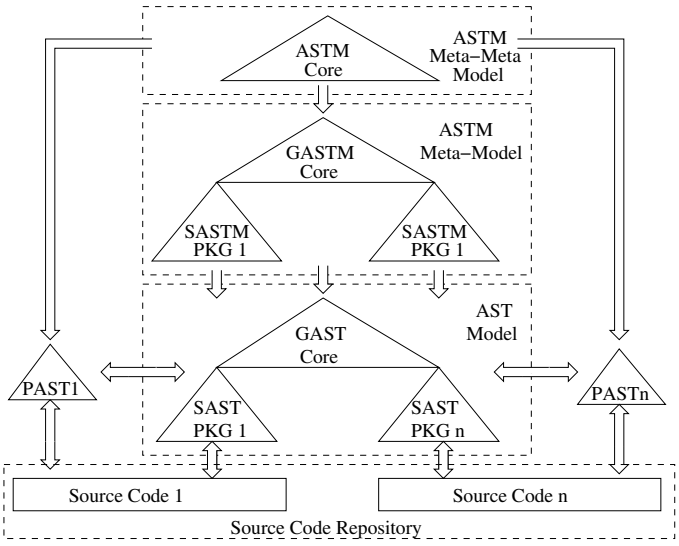


Fig. 2. ASTM modeling framework (from [13])

The OMG approach is condensed in figure 2, that to our knowledge illustrates the current ASTM meta-model proposal [13]. The three top layers describe a modeling framework for abstract syntax trees in analogy to the UML meta-model in object-oriented design. The fourth layer presents a program in plain text representation that, on the same level of abstraction, can be replaced by a

tree like instance of a GAST or SAST instance. The ASTM meta-meta-model on top of the hierarchy is on the same abstraction level as MOF. It could either be replaced by MOF itself, as we do in our approach, or it could itself be modeled as an instance of MOF, using MOF as meta-meta-meta-model.

Layer	Description	ADM examples
Meta-metamodel M3	MOF, i.e. the set of constructs used to define metamodel	MOF Class, MOF Attribute MOF Association, etc.
Metamodel M2	Metamodels, consisting of Instances of MOF constructs	GASTM/UML Profile SAST/UML Profile
Model M1	Models	AST model of COBOL language
Instances (Examples) M0	Objects and data	AST model instances of source code

Table 1. Metamodels for ADM (from [13])

The same proposal [13] contains table 1, that maps the syntax tree hierarchy to the model instance hierarchy. On layer M2 we see that the general GAST model as well as all specific SAST descriptions are constructed as UML profiles. The model layer M1, however, does not contain the source code but an AST model for an arbitrary particular language. In the current table the source code that we consider as a model on level M1 occupies level M0.
Obviously there is an inconsistency. This is due to the fact, that two different things are modeled in the two approaches. One models a running program with its objects and data on layer M0, so that the source code, that is a model for the running program instance is on layer M1. The other models the source code itself, thus placing it on layer M0.

M4			MOF
M3	MOF		ASTM
M2	UML		SAST model
M1	UML-diagram	source code	SAST instance
M0	run. prog.	run. prog.	run. prog.

Table 2. The OMG Approach

If we always consider the running program as the item to be modeled on layer M0, table 2 can be derived. SASTM / GASTM are MOF based meta-meta-models that describe the way to construct language specific / generic ASTs. An instance of such a SAST corresponds to a UML diagram or a program in source code. Note that there is no MOF based model of a source program, but only for UML diagrams or AST instances. These can in turn be transfered into source code by a model to text transformation. In our table 2 those transformations

happen in layer M1 and map column 1 or 3 to column 2. More columns for more languages may be added.
Models and instances for ASTs can generally be divided into a common core part factored out in the GAST model and language specific refinements provided by the SAST model. That means that the SAST model or instance on level M2 or M1 actually consists out of an inheritance hierarchy where the specific SAST part extends and refines a common GAST model. The meta-meta model ASTM, however, is the same for each column.

3.2 Our approach

The syntax tree hierarchy is orthogonal to the model instance hierarchy. From the viewpoint of the model instance hierarchy, the different elements of the syntax tree hierarchy are various representations of the same item. Source code, UML-diagrams, SAST and GAST instances all represent the "program" and are placed on layer M1. The meta-model layer M2 contains the AST models, the descriptions of how to write an AST, either specific SAST or generic GAST.
Note that in our syntax tree hierarchy the GAST is an abstraction from the language specific ASTs and, hence, describes an own language whereas in the OMG approach the GAST is obtained by factoring out common features of SASTs. This abstraction does not leave its corresponding level in the model instance hierarchy. Hence, we appended a separate column in table 3 (as opposed to table 2). Remember that SAST means specific AST, i.e. for each language or variant there is a separate column.

M3	MOF		MOF	MOF
M2	UML		SASTM (model)	GASTM (model)
M1	UML diagram	source code	SAST (instance)	GAST (instance)
M0	run. prog.	run. prog.	run. prog.	run. prog.

Table 3. Our Approach

4 GAST - The Generic Abstract Syntax Tree

As a proof of concept we have defined a Java-like GAST language and provide transformations from and to Java, JavaScript, C++ and C#. The implementation uses Java and XSLT transformations and is developed as an Eclipse plug-in. In section 1 we introduced 3 levels of the syntax tree hierarchy which all belong to the M1 model level of the model instance hierarchy. The most specific layer, the XML representation of a Java source text (JaML), is described in detail in [2]. Its transformation to an abstract Java tree (JAST) and finally to a GAST format is developed in [8]. The languge definition, i.e. the meta-model for our generic

GAST tree is given as a set of UML/MOF diagrams. An alternate description as an XML schema is also available [5, 8].
The meta-models for other, language specific SASTs can be defined in a similar way.
By design the transformation

Java → GAST → Java

works (nearly) without loss of information. That does not hold for the other languages where specific fragments of the language are kept as so-called SAST fragments in a particular element. SAST fragments allow a seamless retransformation back into the source language. But, in the other direction, this will fail, if the target language lacks a similar construct, see 4.4 for more details.

4.1 GAST - Overall Structure

A GAST aggregates a program and an external declaration list, see figure 3.

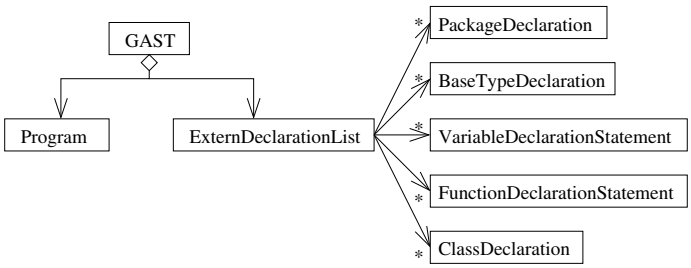


Fig. 3. GAST document

A program consists of several package declarations each of which contains a list of type declarations or statements.

4.2 External Declaration List

A programming language rarely works without any supporting library. So a program usually uses data types, functions, and packages that are not defined internally but belong to the language standard or a library. To cope with this situation we provide the external declaration list. This list holds information about used types and functions from external sources.

4.3 Program

Statements The statements of a program are common to many programming languages. The GAST supports the following statements: block, assignment (ExpressionStatement), one or more kind of conditinal statements (IfStatement, SwitchCaseStatement), one or more kind of loops (ConditionControledLoop, ForeachStatement), declarations (VariableDeclarationStatement, FunctionDeclarationStatement), method calls (FunctionInvocation) or exception handling (ThrowStatement), etc.

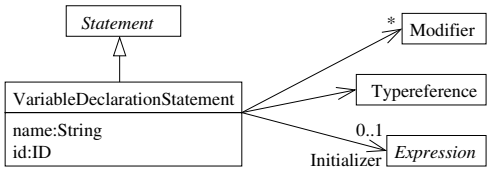


Fig. 4. Variable Declaration

Expressions Many statements impose some sort of constraints for used expressions. The GAST supports the usual arithmetic and logical expressions with unary and binary operators, but not the accumulating assignments which have to be simulated.

Type Declarations Since our GAST describes a statically typed language, each expression and each variable declaration must refer to a type. We provide the type *Type* as base type of all types.
Many programming languages offer some primitive datatypes to represent atomic data units like “int” or “float”. These types are included in the language definition and are therefore declared in the external declaration list. The element “BaseTypeDeclaration” is used for this purpose (see figure 5).
For the treatment of untyped languages a newly defined dynamic type provides generic access to properties as well as a generic invocation of arbitrary methods. Of course, new types can be declared. The complex types (classes or interfaces) may be parametrized (see figure 6).
Summing up we can characterize our GAST language as an object-oriented statically typed language with hooks for adaptation and extension.

4.4 SAST Fragments

To support features that are not directly available in GAST, the element

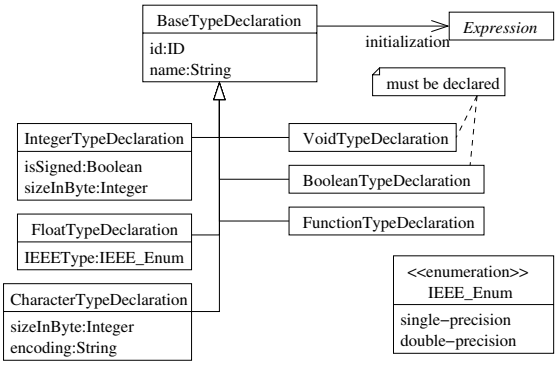


Fig. 5. Basetype Declaration

"SAST_Fragment" was introduced. SAST fragments are peaces of the language specific AST representing an artifact that can not be translated into GAST. This enables the GAST to store arbitrary fragments of various source languages. In this way the information is preserved.

5 Conclusion

We have developed orthogonal hierarchies for abstract syntax trees. In figure 7 the concretization – abstraction hierarchy that relates multiple source languages (here Java and Javascript) with one common generic language is drawn against the model instance hierarchy.
The paper supports the MDA approach by establishing a set of platform independent or platform dependent models of "how the statements of a software asset are structured and thus reflect the grammar of the particular programming language. An AST is a model of the formal structure, but not the language specific form of expression of the program statements." (From [13])
The set of models is enriched. Since all models are offspring of MOF standard MDA tools like mappers or generators can be applied, There is a common meta model facilitating model to model transformations.
With the pragmatic approach of the SAST fragments an intermediate language is defined that supports high-level source code transformations. Its focus certainly lies on object oriented or object based languages, but imperative or even functional languages may also be treated. The uniform XML representation enables tools for language independent program analysis to work. We have, e.g., implemented a metrics tool which will be used to identify plagiarism in students' homework.
The specificaton of an ADM parser as a tool that "generates an ASTM model and exports the GASTM model via XMI" [13] is realized by our transformation from

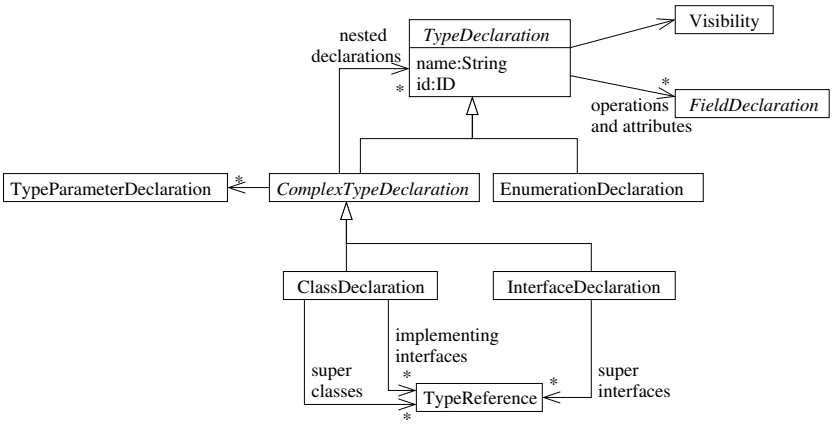


Fig. 6. TypeDeclaration

Java or another source language to GAST as an MDA model. In our approach, however; a proprietary XML representation is generated. We further consider the re-transformation form GAST to target languages.
The definition of XML representations for program source code with increasing degree of abstraction opens new ways of modeling and transforming software.

Acknowledgement: The authors want to thank J.Boldt (OMG) for providing valuable information.

References

1. Jean Bézivin. From object composition to model transformation with the mda. In *TOOLS (39)*, pages 350–354. IEEE Computer Society, 2001.
2. G. Fischer and J. Lusiardi. JaML XML Representation of Java source code. Technical report, University of Wuerzburg, 2008.
3. G. Fischer, J. Lusiardi, and J. Wolff v. Gudenberg. Abstract syntax trees and their role in model driven software development. In *ICSEA online proceedings*. IEEE, 2007.
4. G. Fischer and J. Wolff v. Gudenberg. Improving the quality of programming education by online assessment. In *Proceedings of PPPJ-06*, pages 208–212. Mannheim, 2006.
5. P. Grube. Modeltransformation between an Abstract Syntax Tree and UML. Master's thesis, University of Wuerzburg, 2007.
6. M. Hepp. Abstrakte Syntaxbäume zur Sprachtransformation mit Javascript. Master's thesis, University of Wuerzburg, 2006.
7. T. Kühne. What is a model? In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*, 2004.

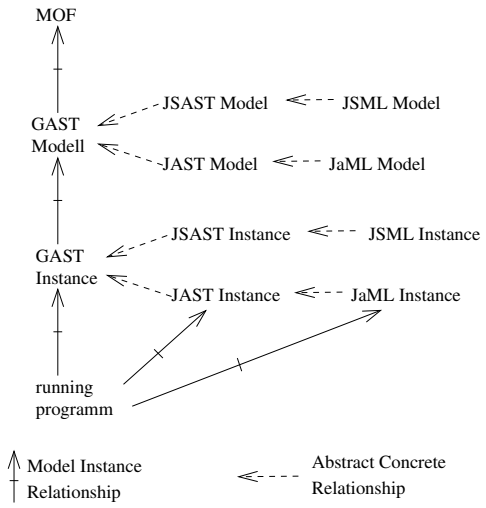


Fig. 7. orthogonal hierarchies

8. J. Lusiardi. Ein Metamodell für abstrakte Syntaxbäume zum Einsatz in der Software-Modernisierung. Master's thesis, University of Wuerzburg, 2006.
9. D. Seipel M. Hopfner and J. Wolff v. Gudenberg. Comprehending and visualizing software based on xml-representations and call graphs. In *11th Int. Workshop on Program Comprehension*. IEEE, 2003.
10. OMG. MDA Guide version 1.0.1. Technical report, Object Management Group, www.omg.org/docs/omg/03-06-01.pdf, 2003.
11. OMG. Uml superstructure, version 2.0. Specification, Object Management Group, 2005.
12. OMG. Architecture-Driven Modernization (ADM). Homepage, Object Management Group, http://adm.omd.org, 2008.
13. OMG. Architecture-Driven Modernization (ADM): Abstract Syntax Tree Metamodel (ASTM). Draft specification, Object Management Group, 2008.

Modernising industrial systems with models

Sébastien Truchat, Thilo Tetzner, Jan Vollmar, Adrian Köhlein

Siemens AG
Corporate Research and Technologies, Software & Engineering, CT SE 5
Günther-Scharowsky-Str. 1, 91058 Erlangen, Germany
{sebastien.truchat}{thilo.tetzner}{jan.vollmar}{adrian.koehlein}@siemens.com

Abstract. An integrated plant model is a core asset of solution providers. In order to benefit of best practice experience, reference structures should enhance re-use. Within the framework of the MOMOCS[1] project, we are investigating a component oriented modelling method, laying an emphasis on domain specific views and dependencies in order to support re-engineering activities. This method is applied in a case study about airport logistics.

Keywords: modelling, systems engineering, industrial systems.

1 Motivation

Any business operations, where a solution provider creates an operative and customer-specific solution to a customer's problem under the terms of a project contract, can be called a solution business. In our context, we will consider applications of this concept for the industrial domains such as the primary, manufacturing and logistics industries. Here, solution providers are primarily concerned with the creation of production and manufacturing facilities (industrial plants). The industrial solutions business faces some major challenges involving time, budget, technical as well as economical aspects. The lifecycle of a plant in the solution business can be divided in the initial plant construction phase (2-5 years) and a plant operation phase (up to 40 years). The budgets and timeframes are decreasing while the requirements of the customer for features, functionality and quality are increasing. Different technical disciplines are involved in the creation of industrial plants. There are less greenfield projects (new plants) and more modernisation projects.

One way to answer to this challenge and doing a successful business is a strategic reuse of information (e.g. concepts, modules) during the execution of projects. On the other hand, for modernising existing plants there is a need for re-engineering them for reasons of maintenance or performance improvement. During these re-engineering phases, engineers may be confronted with huge amounts of heterogeneous, unstructured information to understand the complex system. To manage the two

[1] The work presented in this paper is partly funded by the European Commission through the MOMOCS STREP. The authors are solely responsible for the paper's content.

approaches (reuse and re-engineering) we propose a modeling method for industrial plants based on a component oriented paradigm that lays an emphasis on the dependencies governing the whole system, as well as on domain specific views.

Building a model is by orders of magnitude faster, cheaper and less risky than building a real system, while actual design specifications can be derived from models later on. Models can be designed to analyze specific aspects (model views), e.g. optimisation of specific technological KPIs (Key Performance Indicators). Effects of critical design decisions on the plant can be evaluated based on models. Management of system/plant/ product data over the entire lifecycle requires an integrated information model as well as integration of data, tools and enterprise-level IT.

This leads us to our central statement: "An integrated plant model is the core asset of solution providers/system integrators". Currently, we investigate a case study concerning airport logistics within the framework of the project MOMOCS (Model driven Modernisation of Complex Systems) [1].

In the following part, the concept of technological reference structure, representing the considered paradigm of our department, will first be presented. This leads to the explanation of what model driven modernisation of complex systems means in our industrial context. Then, after shortly introducing our case study in the framework of airport logistics, related works as well as conclusions and further works are discussed.

2 Technological Reference Structure in industrial plant business

The stakeholders in the industrial plant business have to deal with very specific and complex challenges, for example:

- In most cases a plant is a long term investment which is amortized only after several years. That means that the solution provider must ensure a long-term service liability and makes also very high demands at the quality of the solution.
- A further challenge is the involving of many disciplines, (like e.g. mechanical engineering, electrical engineering, automation engineering etc.) with their different structures and workflows. Those are mostly not compatible with each other. Based on this fact, a high integration effort of the working results often follows in the late project phases.
- Solutions within an industry are often very individually aligned to the needs of the end customer. That leads to the fact, that each project must be restarted regarding the planning and realization of the technical contents without a possibility to reuse the experiences of previous projects. The reuse of the project solution is in this case technically extremely complex and usually also economically unprofitable.

The conclusion is: The project execution in the industrial plant business is associated with high risks.

So what the industrial plant business needs today is a better systematization. That does not only refer to the formal lifecycle management but also to the engineering aspects [2][3]. It is necessary to have an overall view regarding the project / plant lifecycle as well as all involved disciplines. This view should be oriented towards the

end customer view. Equally important is the integration of the different project tools (for calculation, ordering, engineering etc).

So, improvements must always be seen in the total context of all aspects of the industrial solution projects and the plant lifecycle.

Figure 1 shows the typical approach today - the project and/or discipline-driven structuring over the phases of the project lifecycle. The phases "Bid Preparation", "Basic Engineering", "Detailed Engineering" and "Commissioning" are only an abstract illustration.

The different structures of the disciplines results of the bid, and engineering phases culminate in late integration of the results and raise difficulties during the integration. The relationship between the different disciplines and their structures are often not clear with regard to:

- Interfaces
- Dependencies
- Functional co-operation

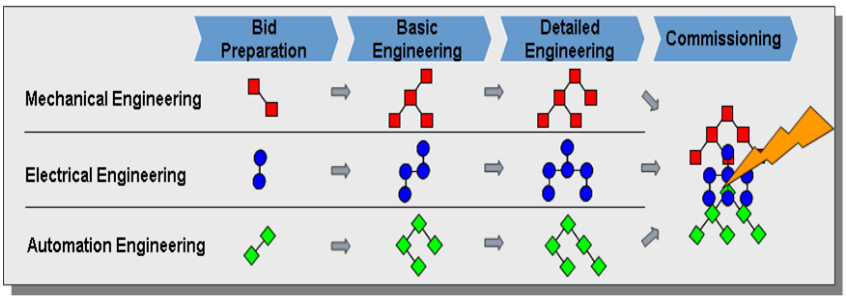


Fig1. Discipline-driven structuring in industrial projects today

The missing continuity within the single phases of the project lifecycle constrains the coordination and the integration of the discipline-specific results.

An example of a methodical approach is the Technological Reference Structure.

It is a paradigm for the modeling of industrial plants and systems and a concept to save and provide consistent information over the whole lifecycle of a plant. Amongst others, proven techniques will be used from the software development (e.g. object-oriented and aspect-oriented concepts) and adapted to the specific needs of plant engineering activities.

Core of the Technological Reference Structure is the Technological Plant Structure (see figure 2). It is built of physical components that are hierarchically arranged according to the technological aspects of the real plant.

There are two types of components: basic components and aggregated components. While the basic components describe the leaves of the trees (smallest technological functional units), the aggregated components will be formed according to the principle: component X consists of sub component X_1', X_2' ... X_n'.

The information will be assigned to the components of the technological plant structure over the entire lifecycle of a project and/or the plant. This information (in

the form of electronic data) concerns all aspects necessary for the planning, realization and the operation of a plant.

In order to get a better handling of this information a classification is introduced. Thus information views are formed, which can be of different kind and complexity, e.g. discipline-specific views on the plant data.

In addition to the information views of the involved disciplines like the "mechanical engineering", "electrical engineering", "automation engineering" etc. there may be information views such as "functional requirements" and "economic aspects" to make wide use of the concept of the Technological Reference Structure (see also figure 3).

The Technological Reference Structure, developed in such a way, can be described with the help of three dimensions:

- Time, i.e. lifecycle of a project / plant
- Structure, detailing along the phases of the project / plant lifecycle
- Information, connecting to the components, augmentation over the phases of the project / plants lifecycle

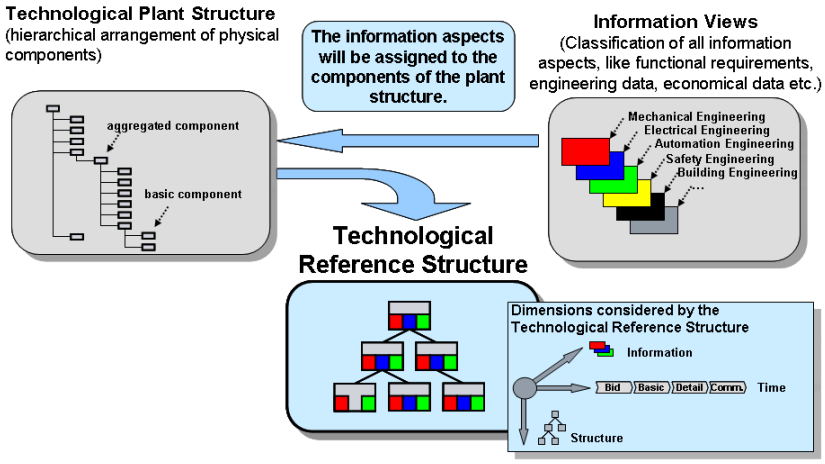


Fig2. Technological Reference Structure

The Technological Reference Structure offers thus a comprehensive project / plant view which corresponds to the customer view. It provides a uniform structure for all disciplines, helps for a better systematization of the Industrial Solution Business, supports the continuity of the disciplines in the project phases promotes the integration also already in the early project phases and increases transparency with regards to interfaces, dependencies, and functional co-operation.

The real benefit of the Technological Reference Structure appears particularly today but also in future in the support of specific tasks as they are shown in figure 3.

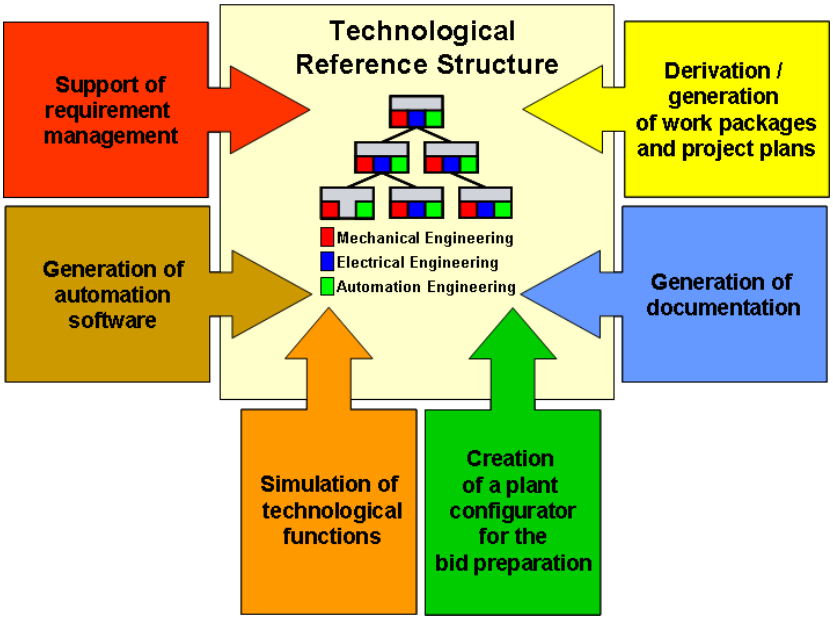


Fig.3 Examples for applications of the Technological Reference Structure

As an example for the use of the Technological Reference Structure the task requirements management can be supported. The benefit here is:

- Enable a better tracing of the requirements (effects of changes and/or new requirements)
- Make a close connection between the requirements and the technological view of the customer
- Enable a better evaluation of requirements completeness

A further example is the support of the generation of automation software. The benefit is among other things the:

- Reduction of manual mass data handling
- Avoidance of systematic errors
- Improvement of the automation software quality and stability

3 Model driven modernisation of complex systems

From an industrial point of view, model driven modernisation of complex systems can be explained by considering following terminological definitions:

- System: a technological system made of one or several cooperating technological objects (static aspect). The technological system realises a technological process by its behaviour (dynamic aspect).

- Complex system: a technological system that is so complex, that it is hard to understand it on the whole just by considering all its technological objects at once. For a better understanding, one can group several technological objects according to the functionality reached by their cooperation. This composite structure can be called an industrial component, and the reached functionality is the technological process associated to it. After applying this composite structure abstraction mechanism one or several times, the complex system can be represented in an understandable way by a few industrial components.
- Modernisation: re-engineering of a technological system in order to achieve improvements and/or new functionalities.
- Model: a document that describes a technological system. A model describes the architecture and the behaviour of a technological system. Depending on the focus, a model may have several abstraction layers, until a certain depth. The goal of a model may be to provide a human understandable information container for engineers, in which case particular attention will be paid to a clear and simple representation. Sometimes the goal of a model may be to provide a formal specification in order to automate some tasks, in which case the emphasis lays on an exhaustive formal representation. These two goals are not always compatible with each other.

From this point of view, model driven modernisation of a complex system means: the model supports the re-engineering of a technological system in order to overcome its complexity.

Our goal is to develop a modelling technique that supports engineers during the re-engineering of technological systems (especially industrial plants). This modelling technique relies on a component oriented paradigm with a special emphasis on views and dependencies [4].

4 Modernising airport logistics

4.1 The modernisation idea

In the following, the focus will be on an empty tray store of an airport logistics scenario. In an airport, the baggage is transported on plastic boards called trays for more care and higher transportation velocity. Each baggage is put on its tray at either the check-in or the transfer to the airplane. At the destination, the baggage is tilted of its tray. Temporarily not needed trays are kept in the empty tray store. Arriving trays are simply stored by queuing up on a belt conveyor. After requesting more trays in the system, the trays are reinducted by using the LIFO principle (see fig. 4 le The need to save time and space is a possible reason to modernise the current empty tray store. The modernisation idea is to install a stacker in the empty tray store in order to put trays in a pile. When trays are piled onto each other, a considerable amount of space is saved, thus augmenting the capacity of the empty tray store. Only when the maximum height of a pile is reached, is the pile conveyed one tray length further on

the belt conveyor. Additionally, the belt conveyor is split in two belt conveyors: one conveyor hosting the stacker and one buffering conveyor before it.

First, all trays pass the scanner storing their IDs in order to know which trays are temporarily not used.

Afterwards, the arriving trays are buffered and step-by-step transported to the processing position by the stacker's belt conveyor, underneath the stacker. The stacker will lift the first tray, hold the tray in this upper position and remain in this position until the next tray arrives. After putting down the first arrived tray onto the second tray, the stacker moves its grippers to the second tray and lifts both trays in the upper position (see fig.4). The stacker will continue with the other trays in the same way until the maximum number of stackable trays is reached.

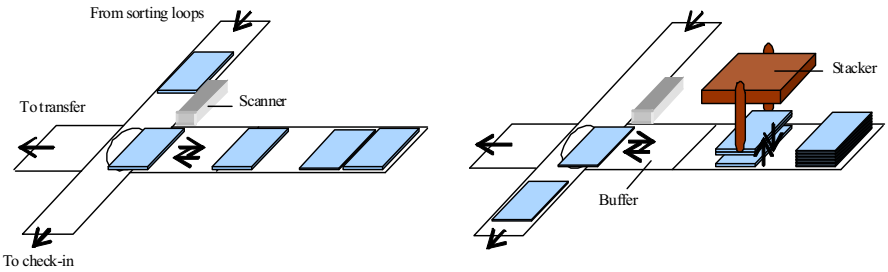


Fig.4 ETS before (left) and after (right) modernisation

4.2 The modelling method

In our work, we investigate a methodology about modelling industrial systems that relies on a component oriented paradigm, the focus of which is on domain specific views and dependencies related to the system. This methodology aims at being as simple as possible in order to sustain intuitively a re-engineering (e.g. modernisation) process across all the engineering domains. In a first step, the system that should be modernised has to be modelled according to these principles. Then, the consequences of every potential modernisation step can be foreseen thanks to the known dependencies of the system, and domain specific views ease the understanding of the various engineers.

First of all, a component oriented model of the system, according to this method, has to be elaborated. Then, the considered components involved in the modernisation process have to be refined, thus elaborating another component oriented model one abstraction layer deeper. These steps are reiterated until the desired abstraction depth is reached. Let us consider the layer of an empty tray store for normal size trays, represented in figure 5. Following abbreviations are used: PC for Photo Cell, BC for Belt Conveyor, ETS for Empty Tray Store, PLC for Programmable Logic Controller. Components are represented by rectangles, and their connectors (i.e. their interfaces)

by little filled rectangles on their edges. Connectors are connected by connections (represented by solid lines).

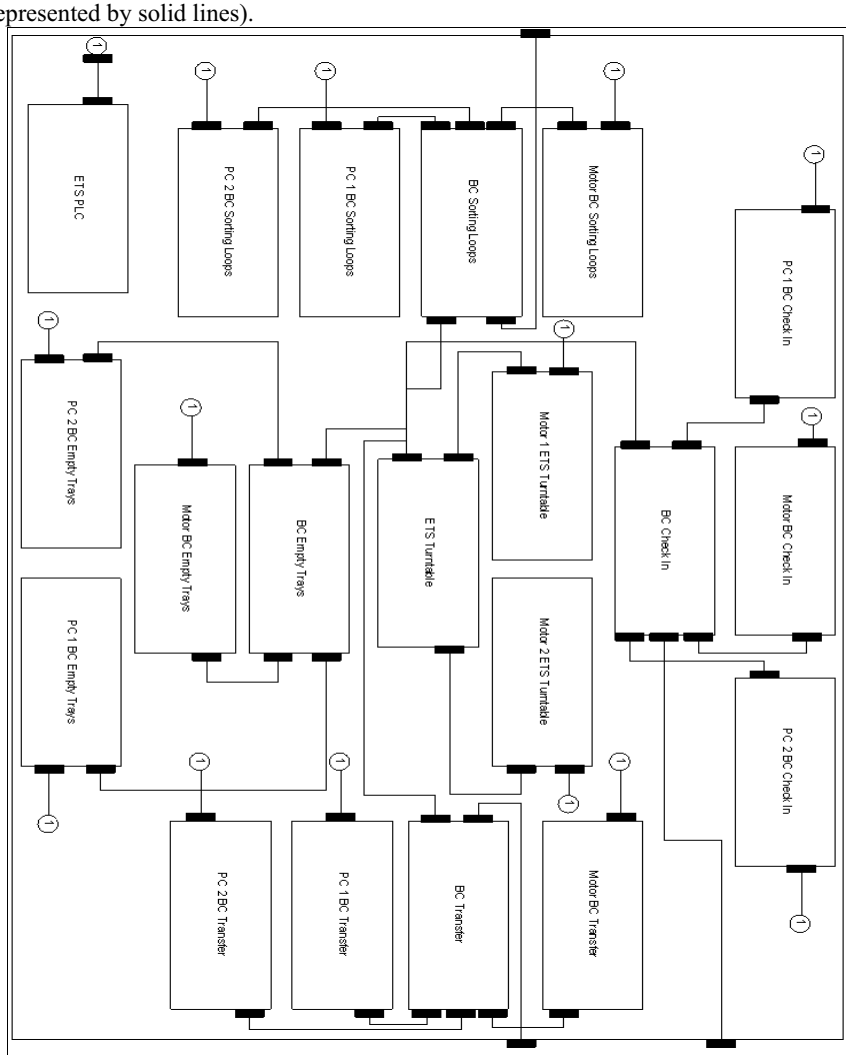


Fig. 5 Empty Tray Store for normal size trays

The goal of the model is to store and represent information about the real system. To this end, we divided the data structure of the model in four tables. The first table contains information about the components, including the mother-component (i.e. the higher abstraction level), the component ID (i.e. its unique identifier in the model), and its type (every concrete component is an instance of a component type). Table 1 represents the example of the component ETS Turntable.

Mother	Component ID	Type
ETS for Normal size trays	ETS Turntable	Turntable

Table 1: ETS Turntable

The second table represents the attributes of the components, which are the concrete information associated with the components. This includes the attribute ID (i.e. its unique identifier in the model), the owner (i.e. the component to which it belongs), the type, the content, and some other elements such as comments or links to documentation. Furthermore, a column can be added for each concerned engineering discipline, so a "flag" can be put in it, whether the considered discipline is involved or not. Table 2 represents one attribute of ETS Turntable.

Attribute ID	Owner	Type	Content	Comment
ETS Turntable Att1	ETS Turntable	Integer	3	Baggage flow in tray per min.

Table 2: Attribute 1 of ETS Turntable

The third table is needed to describe the connectors. As a connector is an interface of a component, the information describing the connector is already stored in some attribute of the component. The goal of this table is thus to put them in relation. This table includes the connector ID (i.e. its unique identifier in the model), the owner (i.e. the component to which the connector belongs), and a list of attributes containing information about this connector. Table 3 represents one connector of ETS Turntable.

Connector ID	Owner	Attributes
ETS Turntable Con1	ETS Turntable	ETS Turntable Att1

Table 3: Connector 1 of ETS Turntable

The fourth table describes the connections of the model. As a connection is a couple of connectors, this table has only two columns for the two identifiers of the connectors. Table 4 represents the connection between a connector of the Turntable and a connector of the belt conveyor from the sorting loops.

Connector 1	Connector 2
ETS Turntable Con1	BC Sorting Loops Con1

Table 4: A connection

The goal of these tables is to provide the needed information in a structured way. For example, a query on table 1 concerning one specific mother component will provide all the components of a considered abstraction layer. A query on table 2 concerning all the flags of a specific engineering discipline may help to highlight all the concerned attributes in order to generate a specific view on the model.

After that, an important point is to define the constraints related to some technological processes (e.g. minimum speed that has to be reached by a belt conveyor) and the dependencies between the components (e.g. linear speed of a belt conveyor is directly dependent on the circular speed of an electrical motor). These constraints and dependencies should be written as equations/inequations involving the contents of attributes, so that they can be machine readable and verifiable.

This way, a modernisation scenario of the system can be partially tested on the model, being aware of all the constraints and dependencies. At the end of some dependency chains, some software parameter may be involved, thus reducing the risk of wrong software re-use: considering the dependencies, the parameters can be adapted by the engineer.

5. Related topics, conclusions, and further work

Component oriented paradigms in order to use models of industrial systems, for example for simulation [5] purposes, have already been used. Works such as [6] also consider the need to integrate system views emerging from different engineering cultures. The focus of our current work is rather to sustain re-engineering phases in the simplest possible way (i.e. usable with no special modelling knowledge) for industrial plant engineering. Even SysML [7] is still a too mighty and general modelling language in this approach. One of our goals is to explore the advantages of defining a UML profile for our modelling method to be as simple as possible to use. The core asset of solution providers is an integrated plant model, where our work wants to put an emphasis on domain specific model-views, and dependency awareness in order to foresee the consequences of re-engineering. A further reason why a UML profile seems attractive is that our methodology could be a valuable complement to existing engineering tools, if there was a way to exchange data with them. At this point, the XMI description of the model combined with the OCL description of dependencies could provide a standardised machine readable basis for interoperation.

References

1. Methodology and related tools for fast reengineering of complex systems. Available (March 2007): http://www.momocs.org/
2. Ulrich Löwen, Rüdiger Bertsch, Birthe Böhm, Simone Prummer und Thilo Tetzner; Siemens AG. "Systematisierung des Engineerings von Industrieanlagen". Zeitschrift : atp – Automatisierungstechnische Praxis / Ausgabe 04 - 2005.
3. Dr. Ulrich Löwen. "Ringvorlesung: Verfahren in der Softwaretechnik, Universität Stuttgart, Institut für Automatisierungs-und Softwaretechnik. 22.11.2006 „Industrielle Anlagen der Zukunft“." Available (May2008): http://www.ias.uni-stuttgart.de/vorlesungen/rv/vorlesung /WS0708/4_Ringvorlesung_Loewen.pdf.
4. Sébastien Truchat, Jan Vollmar, Adrian Köhlein, Ulrich Löwen. "Towards model induced support for engineering industrial systems". Proceedings of 2007 IEEE International Conference on Systems, Man and Cybernetics, October 7-10, 2007, Montréal, Quebec, Canada, pp.2697-2703.
5. K.Wöllhaf and R.Rosen "A Component-Oriented Simulation Approach for Industrial Plant Models: The PlantSim Simulation Tools" ESM 2000, Ghent, May 2000, p. 291-295.
6. Andreas Peukert, Ulrich Walter. "Integrating System Views Emerging from Different Engineering Cultures". INCOSE INSIGHT, Vol 10 Issue 1, January 2007.
7. OMG Systems Modeling Language. Available (March 2007): http://www.omgsysml.org/.

Facility in Complex System Modernization, a Case Study of CRM Modernization

Etienne Brosse and Andrey Sadovykh

SOFTEAM,
Immeuble Le Jupiter, 8 parc Ariane
78284 Guyancourt Cedex, France
{etienne.brosse, andrey.sadovykh}@softeam.fr

Abstract. The complex systems modernization problem concerns many business domains and in particular CRM. This topic is especially challenging due to the lack of an appropriate modernisation methodology and corresponding tooling. OMG ADM proposes a model-driven concept for system modernization. We advocate for the XIRUP methodology based on OMG ADM, which in addition proposed several tools. One of these tools for XIRUP system model editing and analysis is presented in this article and illustrated with a Travel Agency CRM example. Finally, we discuss the benefits and drawbacks of the current approach.

Keywords: ADM, OCL, UML2 Profile, XIRUP, DSM, CRM, MOMOCS.

1 Introduction

This work is performed in the frame of the Model Driven Modernization of Complex Systems (MOMOCS) project which is co-funded by the European Commission [1]. This project develops *eXtreme end-User dRiven Process* (XIRUP) modernization methodology. In this methodology it is proposed to follow the model driven approach for systems modernization. The methodology describes the ways to rebuild the system starting from the legacy artifacts such as source code, documentation and execution logs. The important step is building of the system model, which helps to understand the current system, to specify the modernization goals and which guides the forward MDD process for building the modernized system. For the modelling a highly abstract metamodel is proposed. It allows to hide the implementation complexity and to concentrate on the business logic of the application being modernized. This metamodel is component oriented providing some additional facilities for specification of metrics and constraints. These facilities are necessary for model analysis, which is applied for different steps of the methodology. The example presented in this article illustrates modelling and analysis in a context of the modernization problem.

In this article is dedicated to the XIRUP System Model Editor and Analysis tool and its usage in the XIRUP methodology.

The article is structured as follows hereafter. This section gives an overview of the XIRUP methodology and the metamodel being developed in the MOMOCS project. In the next section 2, we present the Editing and Analysis tool which used for creating XIRUP models and their analysis. In section 3, we explain in details how our tool can help user during the application of modernization methodologies, which is illustrated with a case study. Finally, section 4 describes our future work and provides the conclusions.

1.1 XIRUP Methodology

eXtreme end-User dRiven Process (XIRUP) follows the OMG *Architecture Driven Modernization* (ADM) concept [5]. In this concept, it is suggested to base modernisation on the architectural model. Therefore, the first step is to discover the architecture of the system using available legacy artefacts - documentation, source code, configuration files and/or execution logs. Once the architecture discovered, the analysis phase can start using the benefits of the model-based approaches – hiding of system complexity, separation of concerns, formalism; as well as dedicated Model Drive Development (MDD) tools – model queering, model verification, simulation and testing. Once the modernization goals and methods are identified, the process follows the forward MDD approach – model refinement, transformation, code generation.

This process is summarized on the Fig. 1.

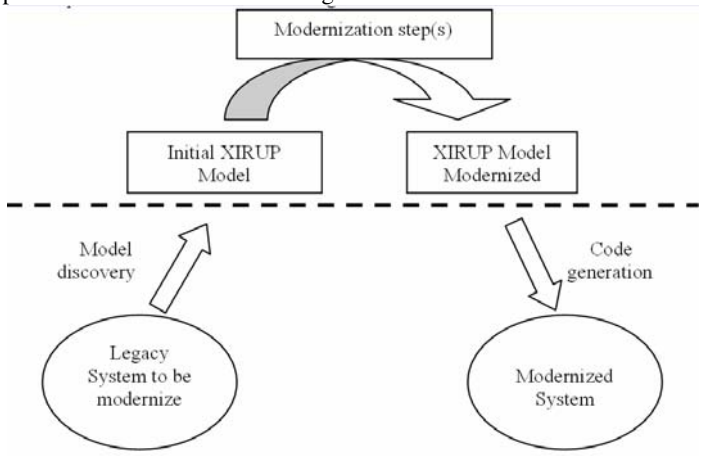


Fig. 1. XIRUP Methodology

Indeed, the system model is discovered and expressed using XIRUP metamodel terms. The modernization steps transform initial system model into a modernized system model. Several consecutive forward MDD transformations should be performed in order to obtain the system implementation.

While the model discovery and model transformations are not trivial steps, the models obtained promise a possibility to reuse them any time the modernization is required again. The MDD techniques and tools allow for numerous benefits such as: platform-independence, traceability of changes and transformations, keeping the documentation up to date thanks to the model-to-text generations, while links between generated code and models on different layers(code-PSM-PIM-CIM) are extremely helpful during the maintenance tasks.

1.2 XIRUP Metamodel

Following the demand from the industrial users, the MOMOCS project developed the XIRUP metamodel. The objectives were to provide:

- Simple and limited number of modelling concepts in order to be comprehensive for large number of users and to be applicable for several engineering/industrial domains (e.g. system/software and telecom/airport building);
- Platform independent view of the system;
- Component orientation;
- Static and behavioural views;
- Support of constraints and metrics for advanced analysis.

The achieved metamodel, inspired by UML2, restricts and extends the concepts of component and activity diagrams. The additional elements (Constraints, Metrics, Tags) are introduced in order to provide more support for model analysis and semantical queering.

Table 1. List of XIRUP main metamodel elements

Static View	Behavioral View	Analysis Support
ComponentInstance	Activity	Constraint
ComponentType	Sub-Activity	ConstraintTemplate
ComponentPart	Decision	ConstraintParameter
Interface	Merge	Metric
Attribute	Fork	Tag
Method	Join	TagType
Table		

Table 1. lists the XIRUP metamodel elements. For more information please refer to [1].

2 Editing and Analysis Tool

The XIRUP System Model Analysis and Editing Tool (EA) is part of the MOMOCS workbench [1], which also contains a Transformation Tool and a Knowledge Base

Repository Tool. These tools are depicted on the Fig. 2. presenting the architecture overview.

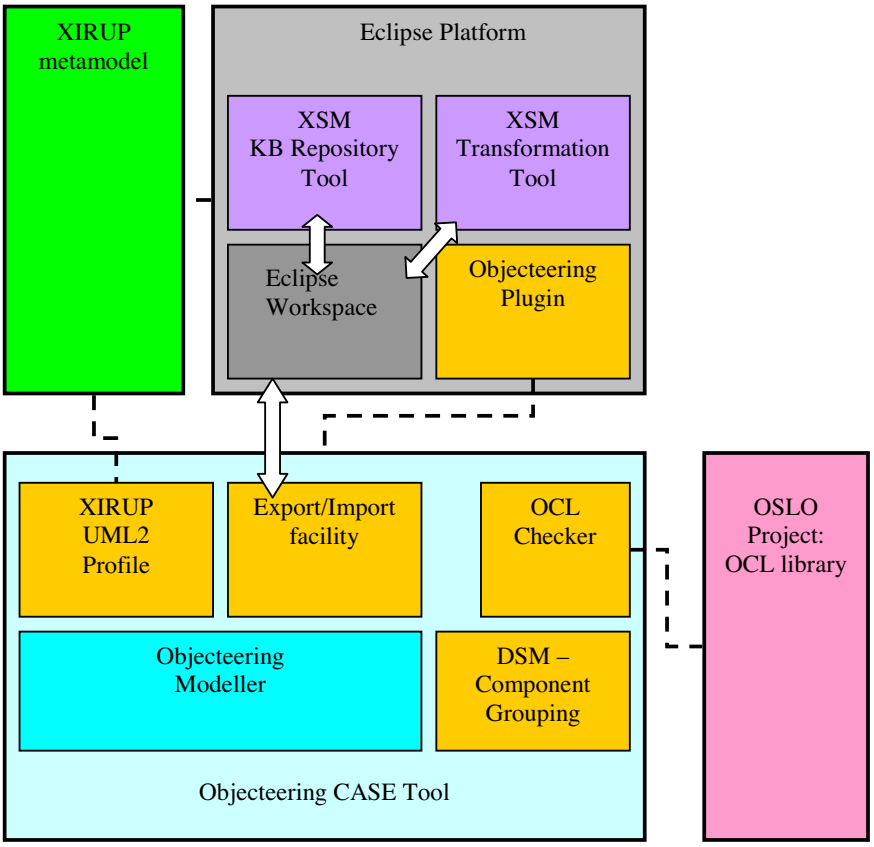


Fig. 2. MOMOCS Workbench Architecture Overview

The Eclipse Platform was used for tools integration. The EA tool is stand-alone, since it is run by Objecteering CASE Tool. However the Objecteering plug-in allows a seamless integration into the Eclipse.

The XIRUP metamodel is shared between tools and represents the commonly used data format. The tools communicate via the Eclipse Workspace by exporting/importing XSM models.

The EA tool is composed of several components:

- XIRUP UML2 Profile – representation of the XIRUP metamodel using UML2 concepts.
- Export/Import facility allowing transformation of the internal object model into the XIRUP metamodel format.

- OCL Checker applying the OCL verification techniques for model analysis. This module is powered by OSLO Project's [3] OCL library.
- and a Design Structure Matrix (DSM) Component Grouping facility.

These features illustrated in details in section 3.

One of the particularly interesting features of the EA tool is the component grouping facility. It is described in the next section.

2.1 DSM Component Grouping

When discovering model from the legacy code (Fig. 1) using reverse engineering, the recovered models are often unstructured presenting all components on the same level without functional grouping or packaging (e.g. visualisation, business logic, and persistency). The grouping facility allows discovering of the functional groups of components by analysing the components dependencies. Thus, this feature can help user to have a more understandable model.

This functionality is based on the DSM formal method. "DSM is a System Analysis: provides a compact and clear representation of a complex system and a capture method for the interactions / interdependencies / interfaces between system elements." [**Erreur ! Source du renvoi introuvable.**]

In DSM context, the system component dependencies are represented in a form of a matrix. Using operations matrix operations groups of the tightly coupled components are identified. Based on this information we create specific composite components for each of the groups. The links between components belonging to different groups are translated into the links between corresponding composites.

3 Travel Agency Case Study

For illustration of our methodology and the EA tool usage we developed an example. This case study consists in a Customer Relationship Management (CRM) system evolution and more precisely in travel agency web application modernization.

A travel agency deployed a CRM web-application for managing travel catalogue, customers database and travel orders. After some years of exploitation, due to increased number of customers, the performance problems appeared. In this way the modernization was required in order to cope with the new demand.

The XIRUP methodology proposes to work on the component level – identify the component architecture of the system, identify how the modernisation goals can be fulfilled using COTS components or what components should be rewritten.

In the scenario we describe, we demonstrate how to:

- create XIRUP System Model (XSM) using the EA tool;
- specify the modernisation goals using formal languages like OCL;
- identify the component that are not satisfying these goals;
- find the most appropriate replacement components;
- and how to automatically integrate them ensuring the consistency of dependencies.

In this way, the travel agency scenario steps are the following ones:
1. Create Travel Agency application model;
2. Specify modernization goal as OCL constraints;
3. Verify model conformance to the modernization goals;
4. Identify Components to be replaced;
5. Find replacement components with OCL;
6. Replace the components ensuring all dependencies;
7. Verify the model validity and its conformance to the modernization goals.

Each step is detailed in the following sections showing tool usage.
The EA tool is involved on the model modernization level (Fig. 1. – top part). That is why the forward MDD process for code generation is not covered by the following description.

3.1 Travel Agency Application Model Creation

There are several ways possible in order to create the travel agency application model: automatically by reverse engineering of the source code or manually by analysing the legacy artefacts like documentation and execution logs.

In our case study we used the manual editing by means of the EA Tool, which provides a simple GUI interface. The Fig. 3. depicts a travel order process captured from the web application configuration files.

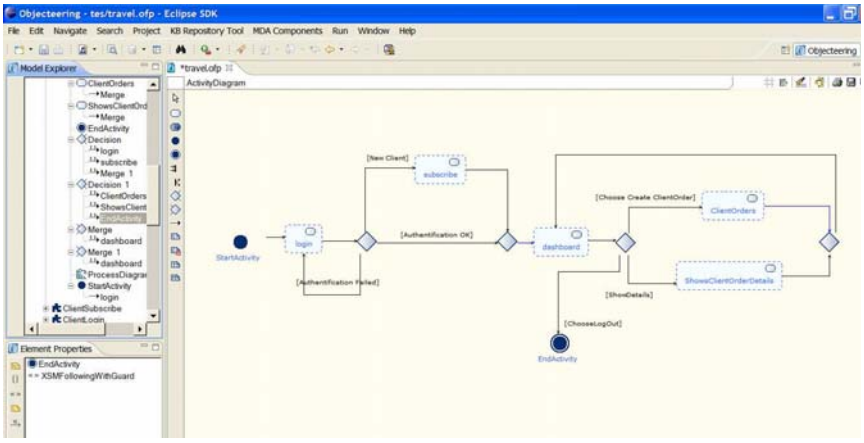


Fig. 3. Client navigation diagram

There is also a possibility to use external reverse engineering tools like MODISCO [6] In this case, the grouping method mentioned before can be helpful.

Our travel agency web application is composed of many pages. Some are dedicated to customers and the others to administration. Each page is represented in our model by a XIRUP component. If our model is created by a reverse engineering tool which makes a correspondence between pages and XIRUP components, the system model will be composed of several components – all on the same level. For the model

understanding, discovering of the functional groups of the components may be necessary. In our example, the components used by customers have many interactions between each other. In the same way the components concerning the administration are also strongly linked.

The use of grouping method in this kind of generated model would create two components – which are named ClientPart and AdminPart and are depicted in Fig. 4.

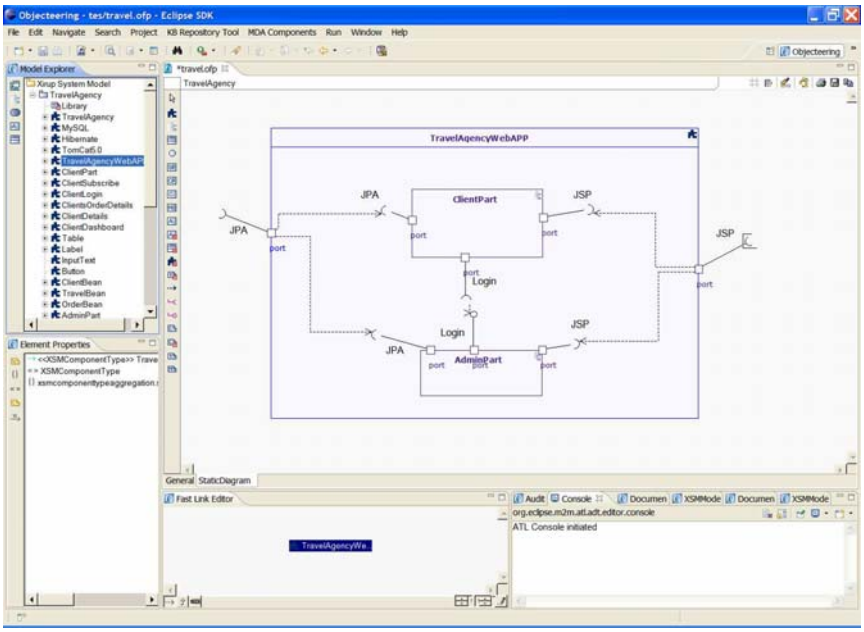


Fig. 4. Grouping Results

3.2 Modernisation Goals Specifications

After the model creation the analysis may begin. In order to automate the analysis, we specify the modernisation goals in the form of OCL Constraints.

In the Travel Agency example we identified the following goals:

- Cost of the complete system should not exceed a predefine value.
- The application performance should be improved allowing more customers to connect simultaneously.

Using the metrics elements all components were annotated with “cost” and “number of connections” metrics - representing the component cost and supported number of connection accordingly. The constraints specified queries the model for metrics and make the operations on metrics. In this way, the goals were translated into the constraints as follows:

- Retrieve “cost” metric of all components and sum up their values. The sum should not exceed the “maximum cost”;
- For each component retrieve the “number of connections” metric and compare with the “minimum number of connections”. This constraint allows discovering a component, which doesn’t satisfy the new demand.

3.3 Automated Model Verification

The OCL checker is the main feature used in this step of our scenario. Once the metrics are associated with the correct XIRUP elements and modernization goals are translated in OCL expressions, the model can be automatically verified by using OCL checker.

The Fig. 5. is an example of OCL checking GUI in the EA tool. In this figure three OCL constraints are shown. The constraint named “Overall cost” validates the fact that the sum of all “cost” metric is under the value of “maximum cost” metric. The constraint named “Number connection” tests if all travel agency web application subcomponents support a number of connections greater than the value predefined in the “minimum connection” metric.

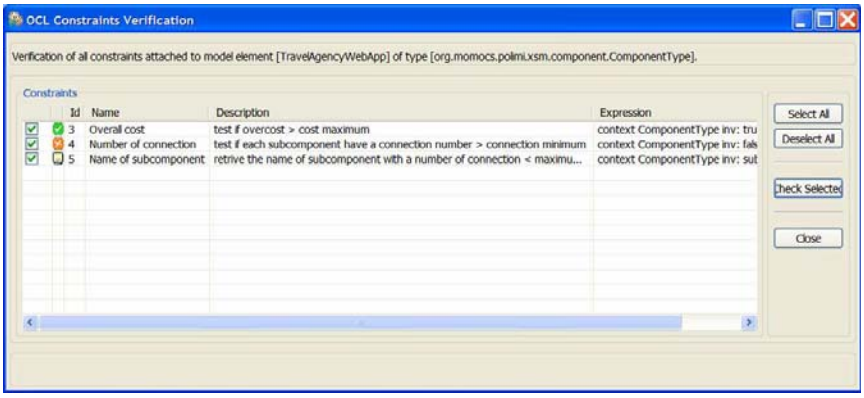


Fig. 5. Constraint checker vie

3.4 Component Identification

The components identification can be done by hand by scanning the entire model, or it can also be done by apply a correct OCL constraint. A constraint is able to retrieve - from the model - the relevant information, like the name of the component which does not satisfy the constraints and thus is to be replaced.

In our scenario, it is the scope of the constraint named “Name of subcomponent”. This constraint returns components name which break the “Number of connection”

constraint. For the Travel Agency the data base component represented by MySQL was identified as a component to be replaced.

3.5 Finding Replacement Component

One of the ways to modernize the Travel Agency application is to find a replacement component from available COTS. At this point of our scenario, we must find a component element in order to replace the one which does not satisfy "Number of connection" constraint. The substitution component finding is done by using the search facilities provided by KB Repository (KBR) Tool – shown in Fig. 2.

3.6 Component Replacement

Once we have found a list of potential replacement components – thanks KBR tool, we must replace the old component by a new one. Of course this replacement can be made by using our tool, but the component replacement is not only a cut and paste of the new component in our model. This replacement implies redefinition of all the links existing between old component and other model elements. In order to do this in a good way and avoid errors, we used the XSM Transformation Tool (TT).

3.7 Modernized Model Verification

Finally, the step described in the section 3.3 should be repeated in order to ensure the model consistency and validity according to the constraints we defined. If the new component breaks any constraint it is possible to get back and to restart the modernization steps once again.

4 Conclusions

In this paper, we presented the results of the effort for development of the XIRUP System Model Editing and Analysis Tool. The tool usage is demonstrated in the context of the Travel Agency example dealing with a modernisation of a CRM application. The ways for translation of the modernization goals into OCL constraint were presented, which allowed automatic model analysis and identification of replacement component.

It was shown that the EA tool plays an important role in XIRUP methodology. The methodology and tools are accepted for the evaluation with case studies from MOMOCS partners – SIEMENS and Telefonica.

For the future work, we will particularly concentrate on the component grouping improvement. For this we will analyze the results the method usage in industrial case studies in order to refine the DSM cost function.

References

1. MOMOCS project web site, http://www.momocs.org
2. Objecteering CASE Tool by SOFTEAM, http://www.objecteering.com
3. Open Source Library for OCL (OSLO), http://oslo-project.berlios.de/
4. The Design Structure Matrix (DSM)Web Site, http://www.dsmweb.org/
5. OMG Architecture Driven Modernization Task Forces, http://adm.omg.org
6. MoDisco Home page, http://www.eclipse.org/gmt/modisco/

Telco case modernization patterns in MOMOCS

Rubén Fuentes-Fernández[1], Francisco Garijo[2], Juan Pavón[1]

[1]Facultad de Informática, Universidad Complutense Madrid
Ciudad Universitaria s/n, 28040 Madrid, Spain
[2]Telefónica I+D
c/ Emilio Vargas 6, 28043 Madrid, Spain
ruben@fdi.ucm.es, fgarijo@tid.es, jpavon@fdi.ucm.es

Abstract. MOMOCS is an European research project that considers the modernization of complex software systems, which can be made of heterogeneous components running on different platforms. The methodological approach to cope with such complexity consists on using models and transformations in an iterative process that is driven by features. Each feature addresses a particular modernization requirement. Putting this into practice with one of the project's case studies, the Telco case study, we have identified several modernization patterns associated to typical features. A modernization pattern describes a typical modernization setting, identifying the structures and characteristics that must appear in the original system, the intended modernization, and the target platform in order to provide a set of automated transformations to achieve the purpose of that scenario. These patterns are of a higher level of abstraction than classical design patterns, as they are defined in MOMOCS at the level of platform independent models and transformations. As such, they can be useful to cope with the difficulty in defining target models and transformations for modernization projects. This paper presents some of these modernization patterns.

1 Introduction

MOMOCS is an European Union research project [11] whose aim is the definition of a model-driven methodology and supporting tools for modernization of complex systems [2]. The MOMOCS methodology, called XIRUP (eXtreme end-User dRiven Process) is a highly iterative process feature-driven [14] structured in four main phases (stages):

- *Preliminary Evaluation*, whose purpose is to decide whether or not the modernization is going to be undertaken/done/planned/performed.
- *Understanding*, to gather knowledge on the existing system and its transformation to the modernized system that meets the requirements. Information about the existing system includes which components are related with the features under study, and their relationships. This information drives to identify components and relationships for the modernized system according to the constraints of the modernization process.

- *Building*, which can be achieved by establishing transformations from components in the existing system to components of the modernized system.
- *Migration*, which involves deployment on specific platforms, considering issues such as the transition from the old system to the new one, including data migration, users' acceptance, and the coexistence of components of old and modernized systems.

Each of these phases involves evaluation activities in order to assess the progress of the modernization and the fulfilment of system requirements. These activities consider specific metrics developed for the XIRUP methodology and inspired in state of the art proposals for modernization projects [5, 16].

A distinctive characteristic of the XIRUP modernization approach is the consideration that the target system will be component-based. Currently, most systems are implemented on some component-based framework, and MOMOCS assumes that this can be applied as a general principle. XIRUP is also in line with the Model Driven Engineering (MDE) approach [10, 15]. It considers that models are the main product of the software process, and that the final running systems should be obtained through automated transformations of these models. MDE has already been considered for modernization projects [6], although in more general settings that the MOMOCS one.

XIRUP takes advantage of these trends in the context of its modernization projects to build models and transformations with primitives that rely on component-based platforms. MOMOCS has defined its own modelling language called XSM, which is defined using Ecore [12], Ecore is a tool-supported language for the definition of languages with the EMF framework. Nevertheless, this support is still quite rudimentary, and it mainly provides tree like editors for Ecore languages. Some examples of this kind of tools are available in the website of the project at http://www.eclipse.org/modeling/emf/. The situation with transformations is quite similar. MOMOCS relies on the ATLAS Transformation Language (ATL) [3], which is currently the most widely used for this purpose. Again, the tool support for ATL is limited [3, 4] and the specification of transformations can be cumbersome and prone to errors. There are some attempts in MOMOCS to produce assistant tools that help in the specification of models and transformations, but they only work for simple cases. Therefore, the success of a typical case, such as the Telco case in MOMOCS, currently relies mainly in the expertise of practitioners in these projects and their related technologies. The validation of XIRUP in MOMOCS can provide here valuable information to increase the knowledge available for these practitioners.

MOMOCS methods and tools are validated with two case studies: one industrial application and one telecommunications (Telco) application. Whilst the first considers a typical control system, with real time constraints and low-level interaction with hardware, the second is related with the distribution of services. The development of these case studies has raised several issues on the application of models and transformations in the modernization process. Particularly interesting for the applicability of the MOMOCS model-driven approach is the identification of modernization patterns, which would facilitate building system models and specification of transformations. These patterns rise from experimentation.

A modernization pattern identifies a recurring transformation from certain structures in the existing system to other in the modernized one. These structures are characterized by the components appearing in them, their relationships, and additional characteristics like the availability of certain properties or methods. Besides, the pattern considers the purpose of the transformation between both structures. With this information, the pattern provides a set of predefined transformations for this setting. These patterns facilitate the identification of the elements to be taken into account when specifying modernization features and the management of their transformations to build the intended modernized system. This information would be still useful in modernization projects even with improved tool support, as it happens with design patterns in common Software Engineering. Here, it must be noticed that these modernization patterns are of higher level than classical design patterns, as they are defined in MOMOCS at the level of platform independent models and transformations.

This paper presents some of the modernization patterns identified in the Telco case study. Section 2 introduces this case study. Then, section 3 provides an informal description of the features that are considered in the Telco case and which modernization patterns have been identified when dealing with these features. Section 4 proposes a way to describe, in a systematic way, modernization patterns, and illustrates this with one particular modernization pattern. Finally, section 5 discusses the advantages of using this kind of patterns and identifies relevant issues to facilitate their application.

2 Telco case study description

The Telco case study considers a Small and Medium Enterprise (SME), with a computing infrastructure of isolated PCs for managing client list, employees, clients accounting, and billing. They have also a traditional telecom infrastructure of fixed and mobile phones. The purpose of modernization is to adapt this infrastructure and services to an integrated framework that is provided and supported by a Solutions Provider (SP). The SP has an infrastructure that allows adapting generic components to customers needs. There are generic components for secure access, Customers Relationship Management (CRM), Enterprise WorkForce Management (EWFM), location, and communications. The SP framework is facilitated by a service architecture, which is made up of software components and based on architectural patterns, at different levels: organization, control, resource, and basic components. This is described in [9].

In order to show the iterative and feature oriented approach of the XIRUP methodology, and to consider modernization in different settings, three concrete scenarios are considered in the Telco case study. These scenarios illustrate three relevant features that can be addressed in iterations of the modernization process.

The first scenario takes into account a concrete application running on SME premises for billing This application is JBilling (see http://www.jbilling.com/), which is open software, so its source code is available. There are no billing components in the SP platform, and taken into consideration the characteristics of the application, the decision of the modernization engineer is to wrap this application to integrate it in the

SP framework as a resource. Therefore, the scenario considers how to extract a model of billing application services and how to build a wrapper according to the SP framework as a resource. The input is the code of the billing software and the SP resource specification. The output is code to wrap the billing software. Data do not need to be modified in this case.

The second scenario takes one of the SP components, whose logic is currently build on a rule engine called ILOG JRules (see http://www.ilog.com/products/jrules/), and modernize it to an Open Source product called Drools (or JBoss Rules, see http://labs.jboss.com/portal/jbossrules). This scenario implies the substitution of the old rule engine for the new one and the adaptation of the reasoning rules to the particular features of the new component. Furthermore, during its implementation, some refactoring has been performed in order to improve code structure.

The third scenario is the more complex as it considers several independent applications of the SME and tries to make an integrated access system for them. For this, it takes SP framework access components and configures these in order to fulfil the requirements from different applications. It provides a role based access system to different services of the modernized system. This scenario can be further decomposed to iterate on simpler features, like the validation of the access credentials or the effective control of the access to components.

By considering these three use cases with concrete scenarios, it has been possible to assess different aspects of MOMOCS methods and tools in a common case study. In particular, these scenarios have been implemented by Telefonica I+D (TID), the Telco partner of MOMOCS, by applying common software engineering practices and with the XIRUP methodology and tools. This has allowed discovering recurrent modernization patterns. These patterns describe specific modernization requirements that when considered in settings with certain features, both for the existing and modernized system, can be solved with similar transformations.

3 Identifying patterns from modernization scenarios

Each scenario deals with some specific modernization feature. To address each of these features, the modernization engineers need to define models of the existing and target (modernized) systems, and also the transformations to convert the existing system into the modernized one. The tasks of specifying these models and transformations are not easy, and they require specific knowledge and experience to make the proper design and implementation decisions. The application of the XIRUP methodology for the Telco case has shown that there are some repetitive tasks and designs in this process, and this has driven to the identification of some modernization patterns that can be found when dealing with typical modernization scenarios. These patterns define models for the existing and target systems, and also the transformations between them. These transformations are usually model-to-model and model-to-text, although some patterns also include text-to-text transformations. In the Telco case, the following patterns were identified, each one associated to some particular features that are relevant for each scenario:

- First scenario: the billing application is integrated in the modernized system as a component of the target platform. Initially, the billing application is running independently of the rest of the applications in SME premises. This application is Open Source, available at http://www.jbilling.com/. The purpose is to integrate this with the rest of the system by wrapping this application as a Resource of the SP framework. The SP framework has a Resource Manager, which is the component responsible for controlling all the resources in the system. Each resource has a management interface, which is used by the Resource Manager, and a usage interface, with specific operations for the clients of the resource. Therefore, the wrapper for the billing application will have to implement both interfaces. This means that the integration is quite simple as only there is a need to identify operations of usage classes in the JBilling application and add them to the usage interface of the wrapper.
    - Modernization pattern 1: *Modernization by wrapping existing software.* This is one of the most typical ways to modernize a system into a target component framework, where all components have to comply with certain interfaces. This happens in frameworks like the one used in this case, the ICARO-T framework [9], and others such as EJB [1].
- Second scenario: Changing the implementation of a component to adapt to a new library or platform. In this scenario, some of the components of the SP framework are implemented with a proprietary tool, ILog JRules (http://www.ilog.com/products/jrules/). The purpose of this scenario is to consider the migration of the code (in principle, the rules, but also associated Java classes) to an implementation with Drools (or JBoss Rules, see http://labs.jboss.com/portal/jbossrules). This problem consists basically of defining transformations from rules in ILog JRules to rules in Drools, and analyzing how objects that are referenced in the rules can be invoked in rules for the target system with Drools. When considering the change of platform, repetitive patterns appear on how to rewrite code from the initial system to the target. Therefore, it should be possible to identify transformation rules for making the modernization process, using the XIRUP methodology and MOMOCS tools to provide some automated support. Additionally, in the process of doing this migration from one platform to another, some improvement in the code can be performed, specifically by refactoring, regrouping, and providing interfaces in order to implement known design patterns that address the intended issues. This is a second feature for this scenario.
    - Modernization pattern 2: *Modernization by transformation and migration of code from one platform to other*. This case works at code level and implies essentially the definition of rules for transforming some code templates to other code templates. Differently from the previous patterns that only imply model-to-model and model-to-text transformations, this one just implies text-to-text transformations.
    - Modernization pattern 3: *Refactoring by applying object-oriented design patterns*. This happens when analyzing the structure of the original code, that some bugs or anti-patterns are identified. Code de-

sign can then be refactored to get better structure by applying some well-known object-oriented design patterns. This requires the availability of specifications of the antipatterns to identify, the corresponding intended patterns, and the transformation rules to make the refactoring from antipatterns to patterns.

- Third scenario: The purpose is to make an integrated access control for all services in the modernized system. As initially the SME has several independent applications, the different information and processes at each one have to be used to configure and feed with data the access controller of the modernized system. In the target platform, the SP framework contains an Access Controller component that manages the user access to the system. When the system starts running, the Access Controller is invoked to carry out the user authentication. This control component uses the Access Visualization Resource to show users a login window where they can write their username, password, and profile. This component also requires the use of the Persistency Resource to verify this data and authenticate users. This involves several features and their patterns:
    - Modernization pattern 4: *Controller-Resource Feature Achievement*. Combination of functionality of several legacy components into a target component managed by a controller. The purpose is simplifying the use of the existing components while providing the interfaces and behaviours expected in the target platform through the controller.
    - Modernization pattern 5: *Resource based component's persistency*. Implementation of the persistency of a legacy component as an independent resource. With this pattern, the resource is isolated from the specific persistence mechanism, what improves its reusability. At the same time, the wide use of this pattern makes that the whole target system becomes more flexible about persistence. It eases the substitution of the persistency resource and the implementation of new services and policies about it.
    - Modernization pattern 6: *Data migration from one context/representation to another*. This pattern is usually present as part of the other modernization patterns. It is achieved through model-to-text transformations.
    - Modernization pattern 7: *Asynchronous controller-view user interface*. Transformation of a legacy user interface into an asynchronous controller-view-model using controller components and resource-view components.

4 Modernization patterns description

As stated in the introduction, a modernization pattern describes a typical modernization setting in terms of the components that should appear in the existing system and the elements to add/modify/remove in the modernized system. The information that characterizes these components includes their structure as classes and relationships,

properties as attributes of these elements, and constraints about applicability. Knowing this is possible to have predefined templates of automated transformations for this setting that avoid, or at least reduce, the burden of the implementation of the modifications. Besides, issues such as the purpose of the pattern, examples, or other patterns related with that setting can be useful for the modernization engineer. This description of modernization pattern includes information similar to the one proposed for *social patterns* in [7], which is inspired by the classical work about design patterns of Gamma et al [8]. This work extends the structure proposed for social patterns to address the specific needs of modernization. The resulting template appears in Fig. 1.

<table>
<tr><th colspan="3">Pattern name</th></tr>
<tr><td colspan="3">1. Aim</td></tr>
<tr><td colspan="3">2. Use</td></tr>
<tr><td>• Applicability
Participants
Structure
Collaboration
Constraints
Format: XSM + Text</td><td>• Transformation
Additions
Modifications
Deletions

Format: ATL + Text</td><td>• Target
Participants
Structure
Collaboration
Constraints
Format: XSM + Text</td></tr>
<tr><td colspan="3">(More use triples with Applicability + Transformation + Solution)</td></tr>
<tr><td colspan="3">3. Process</td></tr>
<tr><td colspan="3">4. Consequences</td></tr>
<tr><td colspan="3">5. Examples</td></tr>
<tr><td colspan="3">6. Related patterns</td></tr>
</table>

Fig. 1. Template for the description of modernization patterns.

A *modernization pattern* is uniquely identified by its *pattern name*. The *aim* is a textual description of the motivation to use the pattern. It describes the kind of expected features that the application of the pattern can bring to the modernization project or the target system.

The *use* section of the template is concerned with when the pattern is applicable, how to use it, and its results. These elements are described as triples of *Applicability*, *Transformation*, and *Target*. The applicability describes the characteristics that the existing system or part of it must present if the modernization engineer wants to apply the pattern. The transformation establishes the modifications to carry out in the existing system that produce the structures described in the target for the modernized system. Given that these patterns are going to be used in modernization projects, probably through support tools, it is important to provide a description of these elements usable in an automated way. For this purpose, the MOMOCS project proposes the use of the XSM language (for information about it see the website of the project at http://www.viewzone.org/momocs) and OCL [13] to specify the applicability and target elements of a use triple. The XSM language is defined with the Ecore [12] metamodeling language and includes specific primitives for modernization projects with component-based systems. This language is at the level of the platform independent models of the Model-Driven Architecture [15]. The XSM diagrams in the use

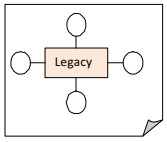
Wrapper

1. Aim.

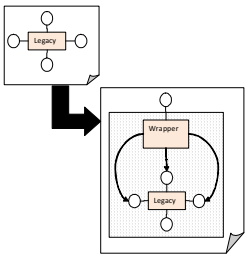
This pattern is intended for modernization by wrapping existing software. The functionalities to be modernized are hidden by a set of new components in the target system. These new components are compliant with the requirements of the modernization project and the target platform. At the same time, they provide services equivalent to those of the original software but implemented on top of that legacy software.

2. Use

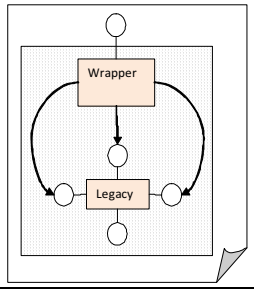
- Applicability



- Transformation



- Target



3. Process

1. Identify external operations of original components.
2. Apply transformations for defining operations in target component, according to target framework constraints.
3. Apply transformations for defining code (essentially, delegation of invocation to corresponding operation in original component) for each new operation in target component.
4. Test target component.

4. Consequences

This pattern is intended to provide functionality existing in the legacy system through interfaces compliant with the target platform. In this sense, the need of adding code for new business rules is usually very low in the settings where this pattern is used. As drawback, this pattern increases the activation chain, though the cost is usually negligible. If performance is a key issue other patterns should be considered.

5. Examples

JBilling case study from the Telco case.

6. Related patterns

1. Component integration. It integrates the access to a set of related services in a common point. As the wrapper, a new component is built on top of the existing ones, but here the main concern is to provide a unique entry point, for instance for security reasons.

2. Intermediary. It is intended to add new business rules over the existing ones. Its organization is similar to the wrapper one. However, in the wrapper the new classes do not add new logics, while the ones in the intermediary do. Hence, the effort in design and implementation of the new classes is higher.

Fig. 2. Template for the description of modernization patterns.

triples include both values and variables, what allows specifying generic conditions of applicability and the results of the pattern. The definition of the transformations uses ATL [3] to describe how the elements on the applicability XSM diagrams must be modified to obtain the elements appearing in the target XSM diagrams. If all these elements are properly described, the pattern can be automatically applied, just matching the source pattern of the transformation, which corresponds to the applicability component, against the system to modernize. Textual descriptions complement the previous diagrams. In fact, there is an intended redundancy of information between the applicability component and the source patterns of the transformations, and the textual and XSM-ATL descriptions. MOMOCS considers that these different views ease the specification, understating, and use of the patterns in different situations, increasing their usefulness. For instance, the requirements of information for an automated transformation are different from those of an engineer studying patterns or of a programmer implementing new transformation rules.

The *process* section provides a sequence of steps for applying the pattern. Initially, it can be described as a list of actions, which are expressed in natural language. Ideally, this would be specified with some formalism (e.g., an activity diagram or other).

The *consequences* section includes the expected results of the application of the pattern, besides its trade-offs and drawbacks. There is also a section of *examples* of usage. This section contains use triples instantiated from real projects and their explanation. Their current examples are extracted from the Telco case. In addition, a list of *related patterns* includes other potentially useful patterns for the same modernization setting and their differences with the considered pattern.

As an example of the use of this template, Fig. 2 shows an excerpt of the description of the wrapper pattern already introduced as modernization pattern 1 in section 3.

5 Conclusions

The identification of modernization patterns can improve the modernization process in several ways. First, it facilitates the definition of models and transformations, which is usually a cumbersome task when following a model-driven approach. Patterns guide the modernization engineer to identify relevant elements to build the models of both the existing and modernized systems, and provide templates for transformations. Second, modernization patterns identify issues that should be taken into account by supporting tools, so these can increase their usability and the productivity of the modernization teams. Third, they are a mechanism to establish a knowledge base for modernization processes. Each modernization pattern reflects some successful experience and ways to model and transform particular modernization features.

The modernization patterns that illustrate this paper are just a starting point to build a knowledge base for the modernization engineer. The structure for the definition of the modernization patterns in this paper is a first proposal that can be improved with comments and specific needs revealed by other practitioners. Another open issue for improving modernization practices is the consideration of tools that guide in the selection and configuration of the modernization patterns.

Acknowledgments

This work has been funded as EU Information Society Technologies Project IST-2006-034466.

References

1. Alur, D, Crupy, J., Malks, D.: *Core J2EE Patterns: Best practices and Design Strategies*. Prentice Hall (2001).
2. Architecture-Driven Modernization Task Force: *Architecture-Driven Modernization: Transforming the Enterprise*. Technical report, (2007). http://adm.omg.org. Accessed on 04/01/2008.
3. Bézivin, J., et al: *The Atlas Transformation Language (ATL)*, (2008). http://www.sciences.univ-nantes.fr/lina/atl/. Accessed on 04/01/2008.
4. Biermann, E., Ehrig, K., Köler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, LNCS, vol. 4199, pp. 452-439. Springer, Heidelberg (2006).
5. Comsys Information Technology Services Inc.: *Comsys-TIM System Metric: The metrics Guide*, (2002). http://www.comsysprojects.com/SystemTransformation/tmmetricguide.htm. Accessed on 04/01/2008.
6. Fleurey, F., Breton, E., Baudry, B., Nicolas, A., Jézéquel, J. M.: Model Driven Engineering for Software Migration in a Large Industrial Context. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, LNCS, vol. 4735, pp. 482-497. Springer, Heidelberg (2007).
7. Fuentes. R., Gómez-Sanz, J.J., Pavón, J.: Model Driven Development of Multi-Agent Systems with Repositories of Social Patterns. In *Engineering Societies in the Agents World VI*, LNAI, vol. 4457, pp. 126–142. Springer, Heidelberg (2007).
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series (1995).
9. Garijo, F.: *ICARO-T Users Manual*. Telefonica I+D Report (2008).
10. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA. In *1st International Conference ICGT 2002*, LNCS, vol. 2505, pp. 90-105, Springer, Heidelberg (2005).
11. MOdel driven MOdernisation of Complex Systems (MOMOCS), EU Information Society Technologies Project IST-2006-034466, (2007). http://www.momocs.org. Accessed on 04/01/2008.
12. Moore, B., Dean, D., Gerber, A., Wagenknecht, G., Vanderheyden, P.:,*Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks (2004).
13. Object Management Group: *Object Constraint Language Specification, Version 2.0*, (May 2006). http://www.omg.org. Accessed on 04/01/2008.
14. Palmer, S. R., Felsing, J. M.: *A Practical Guide to Feature-Driven Development*. Prentice Hall PTR (2002).
15. Schmidt, D.C.: Model Driven Engineering. *IEEE Computer* 39(2), 25-31 (2006).
16. Ulrich, W.: A status on OMG architecture-driven modernization task force. In *Proceedings EDOC Workshop on Model-Driven Evolution of Legacy Systems (MELS)*. IEEE Computer Society Digital Library (2004).

Knowledge Base Repository to foster knowledge sharing in Model Driven Modernisation of complex systems

Jesús Gorroñogoitia, Luis Quijada, Clara Pezuela

Atos Origin S.A.E., Calle Albarracín 25, 28037 Madrid, Spain;
{jesus.gorronogoitia,luis.quijada,clara.pezuela}@atosorigin.com

Abstract. One of the current challenges for IT companies is to face with the increasing complexity of the IT systems. As time goes forth, IT systems requires to be modernised in order to face with: interoperability with other systems, or alignment with new architectural patterns or new development paradigms which may improve, for instance, the system performance, reliability, and other non-functional requirements such as modernization cost. In addition, to cope with the modernization of complex systems, Model Driven Development techniques have proven quite useful. In this context, MOMOCS, an EC STREP project aims at developing a new modernization methodology, named XIRUP, and its supporting tooling. XIRUP methodology proposes an iterative modernization process that encourages the re-usage and sharing of main artefacts involved: XSM models, ATL transformations and other XIRUP artefacts, through a common centralized knowledge base repository, instantiated by the Knowledge Base Repository tool, which rationale, features, architecture and usage scenarios are described in this paper.

A. Introduction

The wealth of IT companies and their clients demanding IT solutions consists of the knowledge gained along with the development of those IT systems. In many cases, IT systems development comprises long lasting development iterations that leverages on previous ones, relying on the availability of some pre-existing system knowledge. In other cases, new IT systems require their integration with pre-existing company IT assets, which also requires the availability of knowledge describing that legacy IT infrastructure. For that reason, IT companies should have access to the knowledge acquired along the life time, so new IT systems developments could rely on the re-usage of all that expertise.

Aforementioned ones are concrete scenarios of the modernization of complex IT systems, which may require a preliminary knowledge on the company IT legacy infrastructure. In this context, MOMOCS, an ongoing EC founded STREP project [1], is aiming at at studying a methodology and related tools for fast reengineering of complex IT systems making extensive usage of Model Driven Development (MDD) techniques. Modernization is meant by MOMOCS as the process by which complex

legacy IT systems (known by MOMOCS as To Be Modernized Systems, TBMS) are transformed to make them compatible with new programming paradigms, software frameworks, architectural patterns, etc. or interoperable with new applications, components, and so on. MOMOCS has provided some main results: a) one methodology named XIRUP (eXtreme end-User dRiven Process) [2] to guide the modernization process, b) a XIRUP meta-model used to instantiate complex system models, and c) a MOMOCS Tools Suite, partial tooling support for this methodology.

MOMOCS leverages OMG modelling standards, and the model-driven development, to propose a holistic iterative modernization process for complex software systems: XIRUP. The ultimate goal is to define models, model-based reasoning, and transformation techniques to help engineers understand existing systems and move towards modernized ones.

XIRUP is heavily based on meta-model in charge of defining the modernization elements, their semantics, and also their mutual constraints. Given the peculiarities of the modernization of a complex system, we cannot reuse an existing meta-model, say pure UML, but we need a new solution: XIRUP meta-model. XIRUP meta-model instantiations, XIRUP System Models (XSMs) are extensively used by the MOMOCS tools during the modernization process.

MOMOCS also provides partial support for most of the phases and activities of the XIRUP methodology. This support is partial since we have to consider: a) MOMOCS project resources availability constraints, and b) some XIRUP activities can be pretty well covered by some third party tools.

MOMOCS has envisaged the need for having a centralized repository that supports the storage and retrieval of artifacts (like, for instance: models, transformation rules sets, transformation mappings, etc.) created at different phases of XIRUP modernization process, encouraging their re-usage, whenever possible. This repository constitutes by its own a complete knowledge base acquired during the modernization experiences. Thereby, MOMOCS has developed a Knowledge Base Repository Tool (KBR), that instantiates this knowledge base. This paper describes the KBR tool, its rationale, features, architecture and usage scenarios in the context of MOMOCS modernization process.

This paper is organized as follows: section B describes the rationale to develop KBR in the context of XIRUP methodology and MOMOCS project, section C describes KBR, concretely: section C.1 describes the main KBR features, section C.2 explains KBR architecture, section C.3 justifies the usage of KBR in the context of MOMOCS case studies and section C.4 describes some enhancements envisaged as future work. Finally, section D summarizes the main paper conclusions.

B. Motivation

An important percentage of the IT investments that companies afford is dedicated to evolve TBMSs towards modernized ones. This is the main target case study of the MOMOCS project. As aforementioned, MOMOCS aims at improving this modernization process by offering, on the one hand, an specialized methodology, XIRUP, to cope with the inherent complexity of the modernization process, and, on

the other hand, a tooling supporting this methodology, consisting of a XIRUP meta-model and the MOMOCS Tools Suite, providing partial support to the modernization life-cycle [3].

XIRUP modernization methodology is an iterative methodology consisting of some well established phases organized into activities and tasks that provides concrete guidelines to help XIRUP analysts (XAs) to cope with the complexity inherent to the modernization of TBMS. This process requires, in most of the real scenarios the concurrent participation of different XAs, with different expertise, focusing on different aspects of the modernization process and/or on different parts of the complex system. Besides, a common modernization process requires, in general, several iterations of the XIRUP methodology, approaching step by step to the final modernized system (MS).

Modernization process activities need to be fed with input artifacts describing, for instance, the original TBMS or with output artifacts produced by previous iterations, or maybe with artifacts resulting of other modernization processes under similar conditions.

Hence, the whole process will produce a lot of artifacts, provided and consumed by different XAs, at different iterations and phases of the modernization process. Those artifacts may describe the TBMS and MS, at different levels of abstractness, offering several views of the system. This set of artifacts could therefore be considered as part of the main knowledge acquired and consumed by IT departments, and therefore, suitable to be collected within a centralized Knowledge Base, fostering their subsequent re-usage.

In consequence, Knowledge Base (KB) plays an important role assisting XAs during the modernization process, providing them support to consume and provide required artifacts from/to the KB. This KB grows with the number of modernization processes accomplished, becoming into the facto know-how of IT companies from which to rely on when seeking for previous expertise.

Afore explained reasons justify the need of having a KB as part of the MOMOCS Tools Suite, that we can summary as: a) fostering of the re-usage of common artifacts among modernization processes, b) availability of a common infrastructure to support the concurrent sharing of common artifacts among modernization process participants, c) integration facilities between other MOMOCS tools since it offers a common repository of shared artifacts d) acquisition of the historic know how belonged to IT departments or companies.

Now, it raises the question whether or not developing a new KB Repository tool (KBR) or, in contrast, to use (likely with some modifications and add-ons) another one developed by a third party, if available. A set of functional requirements for KBR were collected, coming from a deep analysis of MOMOCS case studies requirements [4]. Among those requirements, we can remark: a) KBR should be an Eclipse [5] plugin tool, compatible, at least, at Eclipse workspace sharing level with other MOMOCS tools also based on the EPD[1] framework, b) KBR should provide concurrent access to central common KB repository, c) support for repository structure management, d) support for repository content sorting and filtering, e)

1 Eclipse Plugin Development

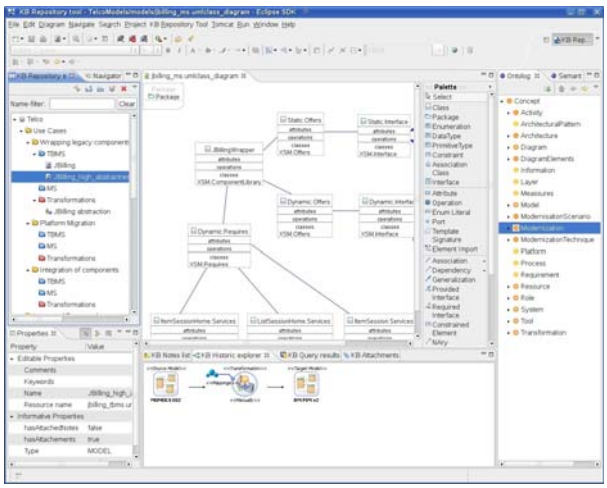
support for repository content search facilities, f) support for repository content historic management, and so on.

A first glimpse, natural election is to rely on the Eclipse workspace and extend Eclipse Explorer View with additional KBR features, using Eclipse extension mechanism. However, Eclipse workspace fails on satisfying those requirements since: a) it can not be shared concurrently by several XAs working with KBR, since several Eclipse instances can not share the same workspace, b) Eclipse Explorer view does not offer adequate search facilities to cope with KBR artifacts like XSMs, because their XMI internal structure makes difficult to rely on a Eclipse keyword text and it lacks of semantic based search support, c) Eclipse Explorer does not provide support for artifacts versioning (historic support) by its own, requiring external supporting plugins (like SVN, CVS plugins) which rely on local workspace copies, but don t support common concurrent repositories. Whereby this election was abandoned. A survey on existing MDD Eclipse based repositories did not give the expected results, whereby it was decided to develop one KBR tool from scratch providing those features specially conceived for a suitable support of the XIRUP modernization process knowledge sharing.

C. MOMOCS Knowledge Base Repository Tool

This section introduces the KBR tool by describing its main features [6][7] in the context of MOMOCS modernization of complex systems and its architecture.

KBR tool provides concurrent access to a common knowledge base repository of XIRUP modernization process artefacts, offering features to reuse the expertise accumulated during different modernisation processes, track the evolution of specific artefacts, artefacts provisioning and retrieval, fast and easy managing of the repository structure, search facilities, and more. A KBR tool snapshot is shown below. KBR tool is part of the MOMOCS Tools Suite.



C.1. Features

KBR tool provide some remarkable features to assist a XA to manage the common knowledge base shared during the modernization process, to store, search and retrieve artifacts and to track artifacts life cycle. A summary of those features follows:

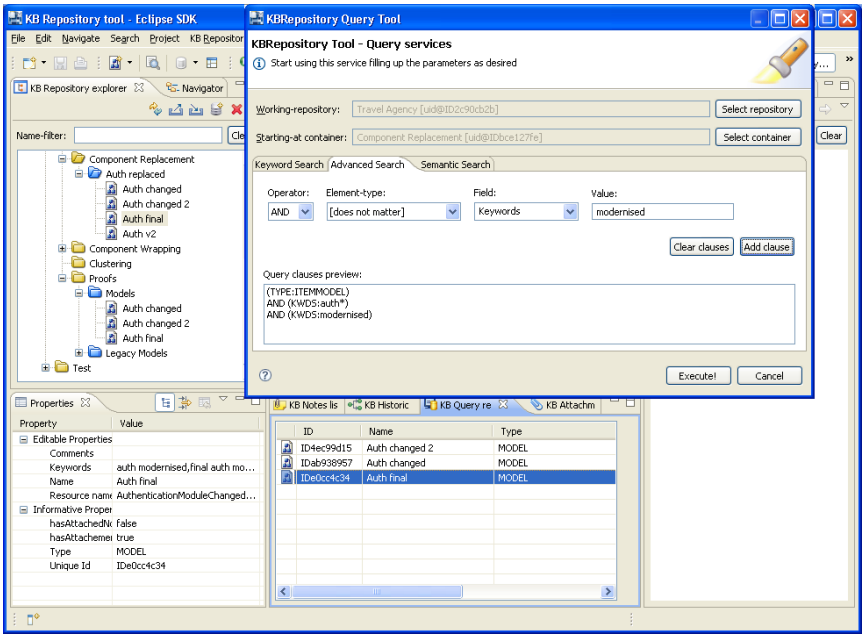
1- User-friendly UI: due to the integration of the KBR within the Eclipse platform and the efforts applied on UI analysis, the tool provides a simple and easy to use UI. Considering the vast number of models, transformations, and other artifacts involved during the modernization, the XA must cope with a very large repository structure which complexity the KBR attempts to reduce. KBR usage is focused in two principles: i) help the user to easily manage the repository structure by applying filters to the management views, hiding/showing different sort of elements, colored typifying, etc; ii) guide the user through the non-trivial task of search-and-find by providing powerful tools to achieve it.

2- Central repository: the modernization of complex systems it is not an easy task and could involve several engineers to work together, even concurrently. Hence, it is mandatory a central repository where to place the domain knowledge and the result of a modernization process. KBR allows XAs to share their expertise, work together, exchange knowledge, track the overall modernization process, and definitively reduce time and effort.

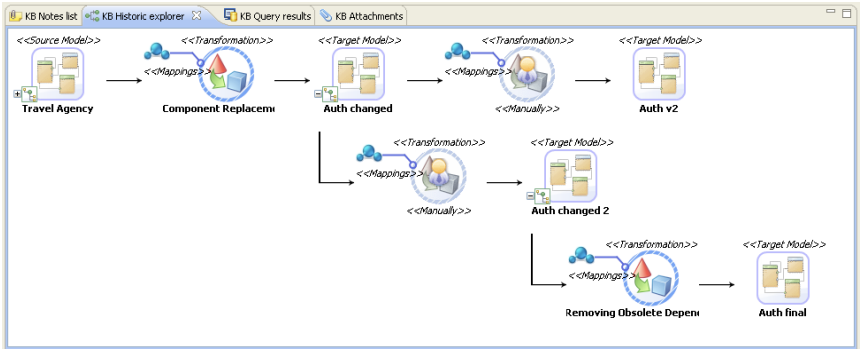
3- Artifacts annotation capabilities: the proposed XIRUP methodology aims to reduce costs and efforts in the modernization process by reusing as much as possible the knowledge acquired. Models, components types, and the rest of the modernization artefacts are no meaningful by themselves. Only establishing the semantics of the given artifact and setting the relationships between them, the XA is able to build a real and reusable knowledge base. For doing that, the KBR provides both a classic metadata annotation (keywords) and a semantic annotation mechanisms. The semantic annotation mechanism allows the XA to describe the semantics of the different artefacts and the relationships between them, constraints and requirements. In this way KBR offers the possibility to query and find those artefacts satisfying some architectural or functional constraints. Provide complete and concise metadata is a key factor for the reusability of the artefacts stored within the knowledge base. The KBR semantic engine makes usage of two domain ontologies that describe the domains of the proposed modernization scenarios in the context of the MOMOCS project : Telco and Industrial use cases. Semantic annotation is provided manually by the XA who selects concepts from those ontologies, assisted by a KBR ontology browser facility.

4- Search facilities: indeed, completing the previous feature, the KBR supplies search services necessary to find out previously annotated artefacts. To accomplish this non-trivial task of searching artefacts and even more, to discover a best candidate artefact (i.e. a valid components models to use in a transformation pattern), both a keyword and ontology based search facilities are provided. Keyword search facility is implemented by using existing well-known OSS libraries (Apache Lucene) for indexing and searching, but improving and customizing the existing analysis algorithms to fit the KBR structure and XAs needs. Besides, KBR supplies a wizard to create advance keyword matching queries. Semantic search facility provides an

implementation for a search engine that applies a ranking algorithm (based on semantic similarity) upon a semantic knowledge base, reported in the scientific literature [11][12]. Both search facilities are complementary, and according with those references, can be applied jointly to improved individual precision, although it has not been tested yet in our scenario. Picture below shows the keyword matching advance searching facility wizard and backward the query results view.



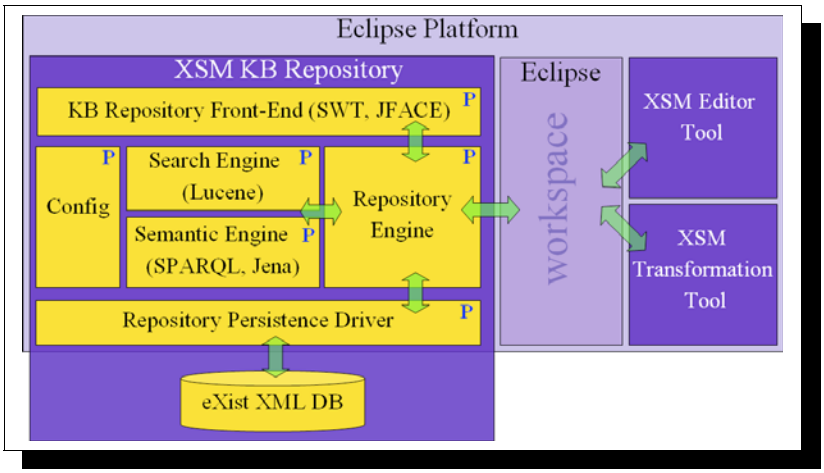
5- Modernization graphs: one of the most important features of KBR in the context of MOMOCS modernization is a feature that, graphically, tracks the evolution of a specific artefact at each modernization step. Picture bellow shows an example of this artifacts evolution tracking feature where the evolution of a XSM for Travel Agency scenario is displayed, showing the transformation suffered and the way to the final proposed modernized model.



C.2. Architecture

The KB Repository tool is built over the Eclipse Platform in the form of an Eclipse Feature . In Eclipse terminology, a feature is a set of bundles (plugins) that can be installed and uninstalled as independent pieces in the overall platform architecture. The election of the Eclipse platform as the main framework for the MOMOCS Tool Suite was motivated because it is the most widely used IDE, provides a robust platform architecture and facilitates the integrations among tools using OSGI [8] technology.

Inside the KBR feature there are seven bundles, separated by functionality. The design of the tool follows a multi tier MVC (Model-View-Controller) architectural pattern. The different plugins that compose the tool interoperates each other in a loosely-coupled way by implementing the Observable / Observer pattern. Figure below shows the overall KBR architecture.



Below it is remarked the KBR components in charge of each role in the MVC pattern:

Model

Represented in the KBR by the Repository Engine, with its configuration plugin, the Search Engine and Semantic Engine plugins and the Repository Persistence Driver with the eXist data storage engine implementation.

View

KB Repository Front-End plugin, based in SWT and JFACE graphic components and controllers.

Controller

It is played by the Eclipse platform itself by configuring its extension-points.

The interoperability between the KB Repository tool and the rest of the tools of the MOMOCS Suite is achieved by working on Eclipse Resource exchange mechanism making use of the Eclipse Workspace.

C.3 Scenarios of usage

MOMOCS project includes the validation of XIRUP methodology and its supporting tools within two case studies: a Telco case [9] and an Industrial case study [10].

Both case studies use the XIRUP modernization methodology to accomplish their modernization goal upon their existing legacy complex scenarios. The Telco case study is applied to the modernization of some software based TBMS, while the industrial case study is applied to the modernization of an airport logistics infrastructure. Both cases are suitable to be modeled using MDD techniques and therefore modernized following the XIRUP methodology.

A real scenario like one of those aforementioned is complex enough to require several XIRUP analysts working concurrently on the same modernization process, maybe on the project as a whole, or on some specific subsystems, to cope with its complexity. That requires concurrent access to common KB repositories to share knowledge and artifacts to accomplish the modernization process. Here is when KBR comes into to play a role.

Let s apply the XIRUP methodology to one of the TBMS of the Telco case study. XAs start the Preliminary Evaluation XIRUP phase from TBMS code sources. TBMS source code needs to be abstracted as much as possible, and translated into a normative format, like one instantiation of XIRUP, XSM, before undertaken this phase. This process can be achieved by translating from source code into UML using some available reverse engineering tools. UML model to XSM can be achieved, for instance, for J2EE systems, like Telco case, by applying a tool developed by MOMOCS team. One obtained, the XSM describing the TBMS are stored within the KBR for future usage.

During the Preliminary Evaluation XIRUP phase, different XAs are working on different specific areas of the TBMS, providing different views with higher levels of abstractness. Manually, different versions of those XSMs are created and stored within the KBR which tracks the changes historic for each.

Once the TBMS is enough understood, XAs try to create preliminary XSMs of the MS from those XSMs stored within the KBR describing the TBMS. This is done at very high level, considering only component models. This process requires a component library describing the technical framework used to modernize the TBMS. A component library that collects a set of XSMs describing the technical framework is available from the KBR, since some XAs designed those component models previously, maybe during another previous modernization process or during a previous iteration of the same Preliminary Evaluation modernization phase. Therefore XAs can use the KBR search facilities to retrieve candidate components (XSMs) from the knowledge base. Best component candidates can be incorporated into the XSMs under construction representing an evolution of the TBMS XSM towards the MS version. Again, those evolutive XSM are stored and track within the KBR to be later consumed by the same or another XA during a next iteration of the same modernization phase or in a subsequent one.

Same approach using KBR can be followed during subsequent XIRUP modernization phases, like in the Understanding and Building phases, fostering the sharing and re-usage of the modernization artifacts between those XAs participating in the modernization process. This approach is complemented by other features provided by KBR as follows:

a) At any time during the modernization process, XAs can use KBR artifact historic tracking facilities for a better understanding about the modernization process and for getting a wider vision of it.

b) As the knowledge base repository for Telco case study gets bigger, KBR offers filtering and sorting capabilities that complement the searching facilities to look for particular artifacts and to cope with repository complexity.

This combine usage of KBR together with the rest of MOMOCS tools supporting the XIRUP methodology speed up the modernization of complex systems, since it fosters the re-usage of artifacts proven useful in other previous modernization experiences

C.4. Future Work

KBR tool is a prototype developed within MOMOCS project that offers important features, but still requires some enhancements to address some of the missing features expected in such type of repository. This section describes some of them considered by authors for future work.

Current prototype of KBR tool does not provide complete support for artifact searching, both in keyword matching and semantic search. Regarding keyword matching, we want to explore additional ranking algorithms beyond that provided by default by Lucene, which can consider the hierarchical keyword metadata organization inherited from the KBR artifacts structure. Regarding semantic search we also want to explorer additional ranking algorithms and leverage search results on semantic reasoning, so relationships and constraints imposed between ontology concepts used to annotate artifacts may restrict the search results, improving its precision.

Both search approaches depend on a previous artifact annotation process, which is done manually in our current prototype. We also want to explorer current semi-automatic annotation experiences reported in the literature [13][14] to instantiate a semi-automatic annotation engine to annotate XSM by inspecting directly into the XMI format, extracting information from the XIRUP meta-model main instances.

Regarding artifacts versioning, KBR versioning support is some restricted and oriented for the convenience of XIRUP modernization process visualization. In order to enhance this support we are exploring how to connect KBR with existing versioning supporting tools (for instance SVN) providing additional versioning features.

The concurrent access support in our current KBR prototype is based on the subscription/publication pattern, which offers poor repository consistency in case of real concurrent access. Improving this concurrent support, leveraging on those additional versioning features scheduled for future work, is another enhancement considered by authors.

D. Conclusions

The wealth of IT companies and their clients demanding IT solutions consists of all the knowledge gained along the development of those IT systems. That knowledge can be collected together in a central knowledge base repository, leveraging on MDD practices, for its sharing and further usage in future IT projects, like those aiming at modernizing complex systems.

KBR tool introduce in this paper an instantiation of that knowledge base repository, in the context of MOMOCS modernization process. KBR offers features supporting the reuse the expertise accumulated during different modernisation processes, the tracking of the evolution of specific created artefacts, the artefacts provisioning and retrieval, the fast and easy management of the repository structure, search facilities, and more.

KBR plays a central role in XIRUP modernization process as a concurrent central repository of artefacts created and/or consumed during that process, integrating at data-flow level the rest of MOMOCS tools.

E. Acknowledgements

The work described in this paper was initiated and motivated by Momocs Project (MOdel driven MOdernisation of Complex Systems) funded by European Commission in FP6 under contract IST- FP6-IP-034466.

Authors wish to thank very much MOMOCS team for the fruitful discussions, comments, suggestions, and so on, which have supported us to produce this work.

Authors wish also thank Atos Research & Innovation (ARI) team, for their support during the MOMOCS project life time.

References

[1] MOMOCS Project, www.momocs.org
[2] MOMOCS consortium. XIRUP Methodology Specifications . Technical Report, MOMOS Project, June 2007
[3] Alessandra Bagnato et al. MOMOCS: MDE for the modernization of complex systems , Neptune Conference, 2008
[4] MOMOCS consortium. XIRUP Supporting Tools Requirements . Technical Report, MOMOS Project, December 2006
[5] Eclipse project, www.eclipse.org
[6] MOMOCS consortium. XIRUP Supporting Tools Specifications . Technical Report, MOMOS Project, July 2007
[7] MOMOCS consortium. Data Modernization Tool . Technical Report, MOMOS Project, February 2008
[8] OSGI Alliance, www.osgi.org
[9] Rubén Fuentes, Francisco Garijo, Juan Pavón . "Telco case modernization patterns in MOMOCS". Submitted to Workshop *Model-Driven Tool- & Process Integration* In conjunction with ECMDA, 2008, Berlin/Germany.
[10] ST. Truchat, JV. Vollmar, AK Kohlein, LU Ulrich. Towards model induced support for engineering industrial systems Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on 7-10 Oct. 2007 Montreal
[11] P. Castells, M. Fernández, and D. Vallet, An Adaptation of the Vector-Space Model for Ontology-Based Information Retrieval , IEEE Transactions on Knowledge and Data Engineering, Feb 2007
[12] D. Vallet, M. Fernández, and P. Castells, An Ontology-Based Information Retrieval Model, Proc. Second European Semantic Web Conf. (ESWC '05), 2005.
[13] Kiryakov, A., Popov, B., Terziev, I., Manov, and D., Ognyanoff, D.: Semantic Annotation, Indexing, and Retrieval. Journal of Web Semantics 2, Issue 1, Elsevier (2004) 49-79
[14] Kiryakov, A., Popov, B., Terziev, I., Manov, and D., Ognyanoff, D.: Semantic Annotation, Indexing, and Retrieval. Journal of Web Semantics 2, Issue 1, Elsevier (2004) 49-79

A Component-oriented Metamodel for the Modernization of Software Applications

Luciano Baresi and Matteo Miraz

Politecnico di Milano – Dipartimento di Elettronica e Informazione
via Golgi 40, 20133 Milano, Italy
baresi|miraz@elet.polimi.it

Abstract. The modernization of a software system is a complex and expensive task and requires a deep understanding of the existing system. The capability of re-factoring a complex application into some high-level views is mandatory to elicit its structure and start localize possible changes. The high number of different implementation technologies imposes a *model-based*, neutral approach to reconstruct the structure and hide unnecessary details. OMG supports this view and proposes KDM (Knowledge Discovery Metamodel) as means to describe software systems in detail, but unfortunately KDM supports a component-oriented decomposition of the system of interest only partially.
To bypass this limitation, the paper proposes the COMO (Component-Oriented MOdernization) metamodel to extend KDM, by borrowing recurring concepts from component-based development and software architectures, and to support a proper componentization of the system we want to modernize. The paper presents the main elements of the COMO metamodel and exemplifies them on a simple case study.

1 Introduction

Many software systems are becoming aged, and their maintenance costs higher and higher. Generally speaking, this is because the older a system becomes, the more expensive (and painful) changes are, but it is also because often developers are forced to create ad-hoc patches that, even if seem to satisfy the new requirements, ball up the entire structure of the application, and violate the original design decisions. On the other end, software applications are important assets for any modern company, which is very reluctant to get rid of its existing systems and introduce completely new ones. The trend is to keep software systems alive as long as possible, and also try to reuse them (their components) in a variety of different business situations.

Componentisation is key to reuse and is also a means to ease maintenance, but components —and their interactions— are too often tangled within the system. The lack of a clear software architecture [1] hampers the actual reuse of the different parts, which are not shaped as fully independent entities. Unclear interactions complicate the redesign, and replacement, of a single part of the system since we need to understand its functionality, the components it depends on, and

also those that depend on it. This is why, even if there is clearly a point above which the cost of maintaining a component becomes higher than its value [2], we often tend to postpone its substitution. The clear and full understanding of what we want to change, usually not supported by available documentation, is often a barrier against more radical changes.

Traditional maintenance activities (corrective, perfective, and adaptive) are usually devoted to correct faults, to improve systems' performance, or to adapt to changed contexts. However, when the scope of these requirements becomes wide, a thorough evolution of the system —often called *modernization*— becomes mandatory. Even after years of experience in programming languages, abstractions, and software processes, modernization is often carried out by reasoning at the level of the source code, maybe because of the scarce quality of available documentation, usually not aligned with implemented components. There is no real attempt to conceive a global —and sufficiently abstract— view of the system before starting to change it.

Recently, OMG wanted to fill this gap by proposing the ADM (Architecture Driven Modernization [3]) initiative, as means to address all the main aspects of the modernization of complex software systems. ADM proposes KDM (Knowledge Discovery Metamodel) as reference metamodel to produce significant models from any software artifact and get rid of implementation details. Unfortunately, KDM works at a quite low level, and does not provide constructs to conveniently render the architecture —components and connections— of a complex software system. In contrast, we think that modernization requires that the components and interfaces, which constitute the system of the interest, be properly identified.

Our assumption is that modernization must start by reconciling, and rendering, the system's structure at architectural level. To this end, the paper proposes the COMO (Component-Oriented MOdernization) metamodel to extend KDM with the concepts of component and interface. By lifting the abstraction level (w.r.t. KDM), we can produce an adequate view of the entire system, and leverage the boundaries between components to better focus the subsequent modernization effort. The compatibility with KDM ensures that each component be associated with the concrete set of programming elements that constitute its implementation and changes be propagated to the corresponding source code.

The COMO metamodel is validated by reconstructing and rendering the architecture of an open-source billing system (widely used in the telco domain). The first results witness that the metamodel fully supports the elicitation of the software architecture, facilitates the understanding of the system, and thus ease its modernization.

The rest of the paper is organized as follows. Section 2 briefly presents KDM, which is the basis of our work, while Section 3 introduces the case study and elicits the requirements behind the modernization of a complex system. Section 4 introduces the COMO metamodel, and Section 5 compares it with the state of the art. Section 6 concludes the paper.

2 Knowledge Discovery Metamodel

The Knowledge Discovery Metamodel (KDM, [4, 5]) defines a metamodel for the representation of existing software applications, the relationships between its parts, and those with the environment. KDM supports both object-oriented and procedural artifacts and provides them with a homogenous low-level representation. It also fosters modernization through model-based transformations, and ensures that all changes be easily propagated to the source code, whose relationships with the model are kept through traceability links.

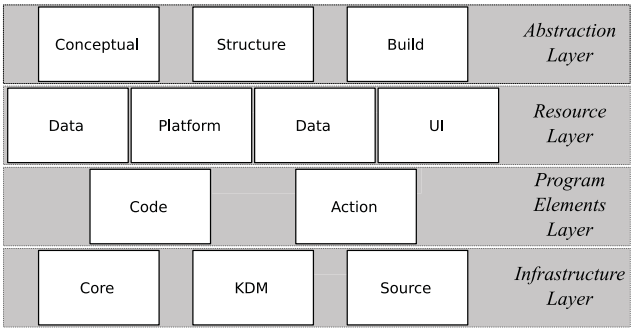


Fig. 1. KDM organization.

The KDM metamodel is organized around the four layers of Figure 1, where each layer describes the application at a higher abstraction level. The first layer, called *Infrastructure Layer*, provides the root (abstract) elements, along with their common properties, for all KDM concepts: a `KDMModel` comprises `KDMElements` and `KDMRelationships`, which make any KDM model resemble an entity-relationship diagram. Moreover, each `KDMElement` can be linked to its corresponding physical artifacts (e.g., source files, images, or configuration options) to set the relationships between model elements and source code regions, and be able to propagate model-based transformations onto the relevant physical elements.

The *Program Elements Layer* specializes the elements above to describe the actual implementation of the system and provides means to render the program elements and their behavior. It abstracts from specific programming languages, and represent the low-level details of the system in terms of data types, callable units, and shared variables; their behavior is rendered by means of a sort of abstract syntax tree.

The last two layers enrich the basic model. The *Resource Layer* deals with the runtime resources used by a system. It supplies the elements to represent the runtime operative environment, which sometimes hinder the comprehension of the overall behavior of the system. Modeling explicitly the environment that

hosts the system eases the understanding of subtle interactions between its parts: for example, a neat model of the locking system would allow us to detect possible deadlocks. This layer is also in charge of modeling the user interfaces, including their possible compositions and allowed sequences of operations, and how data are organized. This way, we can accurately model complex data repositories, like record files, relational databases, or XML schemas, along with the possible operations on them. The behavior of (parts of) the system is rendered by using state machines, which properly model the abstract states associated with the different resources.

The last layer (*Abstractions Layer*) provides the elements to represent domain- and application-specific abstractions. We can address the building phase of the system by accurately describing how to compile the source files and the artifacts the process generates. It is also possible to render the structure of the system by grouping basic KDM elements into layers, subsystems, and components[1]. The last abstraction supported by KDM borrows its key elements from *SBVR* (Semantics of Business Vocabulary and Business Rules) to represent the conceptual model of a system in terms of *terms*, *facts*, and *rules*.

If we think of how (modernization) tools can exploit KDM, each tool can claim to be *L0* KDM-compliant. This means it must be able to understand the first two layers, that is, it must be able to cope with the overall structure, on one side, and the low-level programming constructs, along with their behavior, on the other side. The capability of understanding part of the other elements leads to *L1* compatibility, while if the tool is able to deal with all the higher-level constructs, we have *L2* compatibility. All tools must preserve the parts they do not understand.

3 JBilling

As example of software application we want to modernize, in this paper we use *JBilling*, which is an open-source billing system[2]. This is an interesting example since its nature allows us to consider it a good representative of "average" (in terms of both quality and dimension) software projects and its size —even if not gigantic— needs attention. The *vanilla* version comprises 581 classes (91 KLOC) and 227 JSPs (14 KLOC). Moreover, it leverages the services provided by the J2EE application server that runs it, and the comprehension of the system is further hampered since the interactions among classes are mediated by it.

KDM allows us to create an integrated model of the whole system to precisely describe each class and the mediation provided by the application server. Figure 2 shows a UML-like simplified representation of the KDM model of *JBilling*. The high-level view comprises three parts: **Database**, which stores both the rules for calculating rates and users' bills, **Server**, which hosts the billing logic and is

[1] A KDM component is more a container for lower level entities than an element of the software architecture of the system.

[2] Available at `http://www.jbilling.com`.

in charge of the actual computations, and **Client**, which offers a Web-based administration console. Moreover, these three parts are implemented using different technologies: the Web-based subsystem uses STRUTS, the database storage is a layer of entity beans, and the logic is realized with session beans.

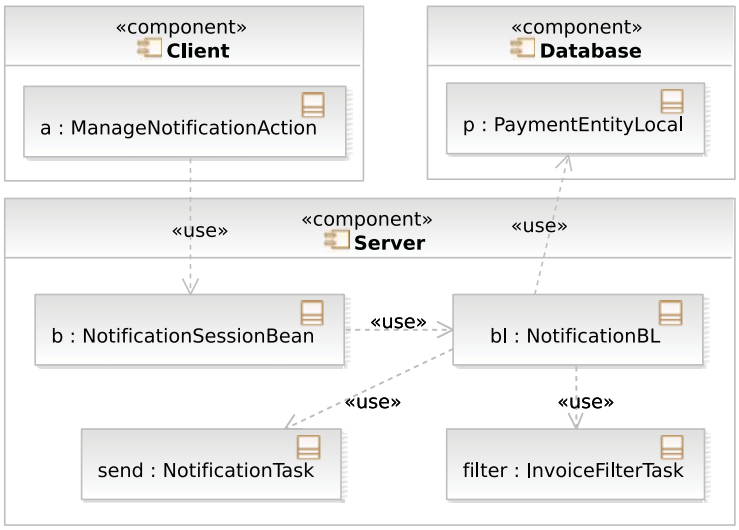


Fig. 2. Partial view of JBilling's architecture

Even if there are different features provided by JBilling, they all are organized around these three layers and follow a similar pattern. Each feature can be seen as a vertical subsystem (with respect to the three parts): for example, the figure concentrates on the internals of the notification subsystem. The user manages it through the Web-based administration client by using the `ManageNotificationAction` class, which in turn uses `NotificationSessionBean`. This is a facade to the notification business logic (`NotificationBL`) in charge of the real business decisions. This class uses `NotificationTask` and `InvoiceFilterTask` to perform some special-purpose operations, and `PaymentEntityLocal` to access the database.

This is all we can get from KDM: its high-level components are more boxes than real architectural components. As soon as the number of relationships between the elements contained in the different boxes increases, the readability and usefulness of such a model are quickly compromised. Even if technically feasible, it is unrealistic to start reasoning on the application by analyzing such a big, flat, and fine-graded model. KDM does not provide well-defined boundaries among its components (containers), which do not declare any interface, be it offered or required.

KDM provides a hierarchical representation of the system, but low-level `use` relationships can cross the boundaries of the different components the way they want. Too tangled dependencies hinder the comprehension of the system, and also precludes the reuse of the different parts. In contrast, a more component-oriented view would facilitate the analysis of the system (at the architectural level), and help better scope changes. If we were able to reason in terms of cohesive elements, with precise offered and required interfaces, we could easily identify changes, predict their impact, and reason on substitutability and dependency mismatches. At the same time, components and interfaces must be linked to the concrete programming elements that form them. These links ensure that all architectural changes be directly reflected onto the source code.

4 COMO metamodel

The COMO metamodel aims to overcome the limitations described above by combining KDM, that is, the state of the art metamodel for program modernization, with traditional concepts borrowed from software architectures. The former is good at creating the structural view of existing systems, while the latter are good at modeling the overall functional organization. Our metamodel combines the benefits of the two domains: it enriches KDM with the modeling elements required to fully render the software architecture of the system, and exploits it to correlate model-based changes to those on the source code.

As already said, KDM supports low-level programming constructs, domain-specific concepts, and high-level views. As for the latter, it offers the elements of Figure 3. `StructureModel` allows for the creation of different "structures" of the system. `KDMEntities` can be organized into a set of `StructureElements` by using the `groups` relation. Each `StructureElement` represents a set of `KDMEntities` that has a precise role in the overall organization of the system. KDM also supports the hierarchical partitioning of the system by means of a ownership relation between different structural elements.

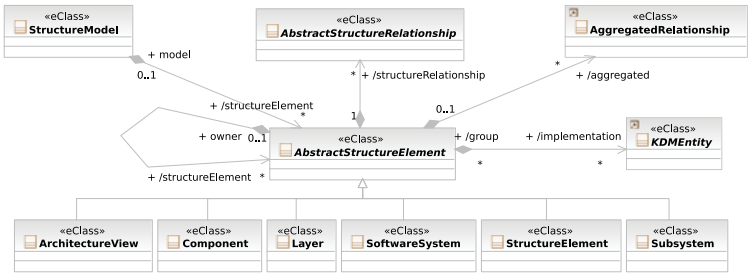


Fig. 3. KDM structural elements.

Natively, KDM provides several kind of structure elements (e.g., subsystems, components, layers, etc.) to model complex structures by mixing them adequately. However, these elements are nothing but named containers, able to host and group together low-level elements. The meaning of these structure elements is misleading since they provide a hierarchical organization, but do not support any high-level abstraction since the relationships among them are defined in terms of low-level dependencies among low-level elements.

These structural views also come with severe limitations that hinder their actual usability. Since the structural division must be non-overlapping and strictly hierarchical, the reuse of a component is forbidden, but this is clearly against the idea of reuse, which suggests to use and share the same elements as many times as possible. Moreover, KDM does not explicitly define how different structure elements interact, and thus we cannot clearly detect what a component offers or what it requires from the others. For this reason, when we want to modify something, KDM forces us to perform a complete check of the system model, losing the benefits of a more disciplined and constrained approach.

To avoid all these limitations, the COMO metamodel enhances KDM with well-known forward engineering concepts to better accommodate the functional organization (architecture) of a system.

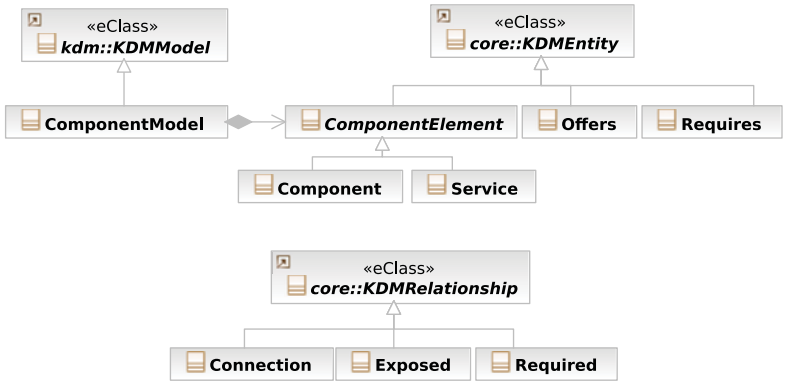


Fig. 4. COMO main elements.

Figure 4 presents all the elements we introduce and specifies their KDM types (super classes). A `ComponentModel` helps create a model able to store the architectural views on the system. For this purpose, it has a reference to all its `ComponentElements`, which can be either `Components` or `Services`. The former renders component, while the latter expresses the idea of a functionality that can be offered or required by a component. There are also new kinds of relationships to enable a component to expose or require some services (through entities

`Offers` and `Requires`, respectively), or to interconnect components to form a composite component (relationships `Exposed`, `Connection`, and `Required`, respectively).

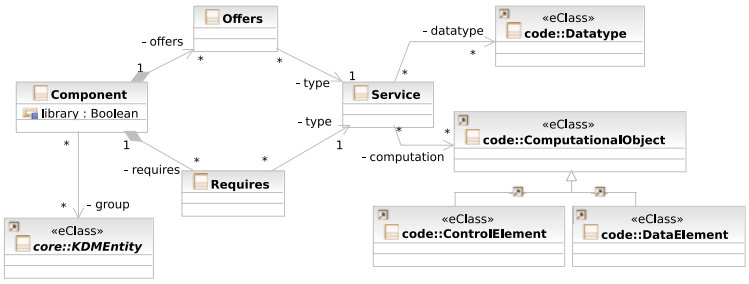


Fig. 5. COMO Components.

Figure 5 concentrates on COMO components. A `Component` is a group of other KDM elements, thus it is able to host standard KDM entities such as code elements and runtime resources. Technically it is able also to host other components (since `Component` specializes `KDMEntity` in Figure 4), but to create composite components it is preferable to use our composition mechanism, detailed below. A component may be flagged as *library*, to mean it is provided by a third party and thus we cannot modify its internal structure: for example, components bought from other companies, which group libraries and other binary artefacts, or components managed by a third party, which are used remotely, like Web services.

A `Service` renders a feature offered or required by a `Component`. `Offers` is used to model a `Component` that offers a `Service`. At implementation level, we must provide both the `DataTypes` and `ComputationalObjects` declared by the `Service`. Dually, `Requires` is used to model the fact that a `Component` needs a `Service`, thus depending on some of its `DataTypes` and `ComputationalObjects`. A `Service` can contain particular data types, modeled in KDM by means of proper `DataType` elements, callable methods (`ControlElement`), and shared variables (`DataElement`). This way we can distinguish what a component offers publicly, with respect to its internals, and impose that when it is changed, we trigger an adequate check on the whole system.

Figure 6 shows how components can be composed of other components. This composition mechanism should allow one to reuse the same component while forming different composite components, as well as permit to use multiple instances of the same component while forming the composite one. For example, consider the creation of a graphical user interfaces with predefined components: the component able to render a text field should be reused in all forms that requires it, and within a single form it is possible to have multiple text fields.

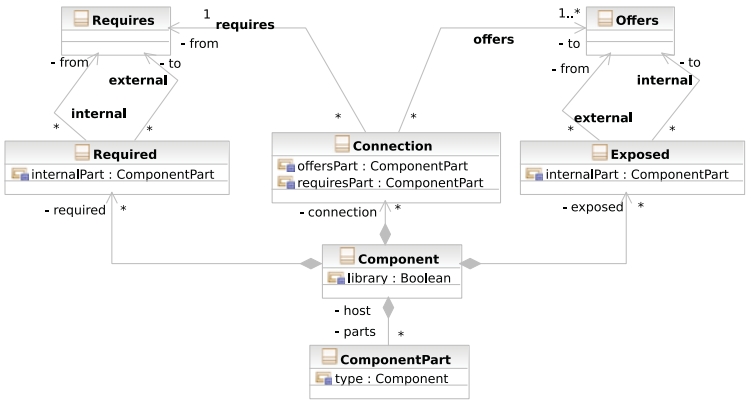


Fig. 6. COMO composite components.

The `ComponentPart` element enables both these kind of reuse. Each part represents the usage of a component, referenced by its type, inside another component. This structure fosters the reuse of components that can be hosted by different components as needed. Obviously, we must avoid cycles in the composition to avoid strange situations (e.g., component `A` is composed of `B` that in turn is composed of `A`), but we can create multiple parts with the same type inside the same composite component. Notice that a composite component is a component, and thus it can also hosts arbitrary KDM entities. This is important to be able to mix (internal) components with additional code, and thus provide more advanced services than those provided separately by the parts. In this way it is possible to model the code that it is necessary to glue together the internal components, able for example to coordinate them and to adapt the different data formats. Real-world systems often use such a mechanism, and the metamodel must render it.

Once composite components are created, we can handle the services exposed or required by its internal components. In particular, it is mandatory to satisfy all services required by internal components, and it is also possible to export the services they provide. These relationships can be materialized in three different ways:

– An internal component requires a service not provided by another internal component. The request is propagated outside the composite component by using a `Required` relationship that connects the internal request with the external one. Since we can have several internal components of the same type within the same composite component, we univocally select the one of interest by adequately setting the `internalParts` attribute.

– A service offered by the composite component is provided by an internal component. To render this situation, we use the `Exposed` relationship by connecting the external offer to the actual internal provision. Again, `internalParts` univocally identify the internal component that provides the service.
– An internal component requires a service that is provided by another internal component. In this case, we need to connect the `Offers` and `Requires` entities involved by means of a `Connection`. This way, we univocally identify how the interaction happens.

The elements presented so far model the static structure of the system, define boundaries among components, and specify that all possible interactions must pass through the predefined services (i.e., their interfaces). Now, we have a structure that better fits the needs of modernization efforts. The well-defined boundaries allow us to easily add new components, change existing ones (by paying attention to offered and required services), and replace those parts of the system that do not satisfy our requirements anymore. Moreover, since everything is tightly integrated into KDM, it is guaranteed that these modifications are directly mapped onto the real system (low-level entities)

The COMO metamodel also supports standard and widely used constraints languages, like OCL [6], to express constraints imposed by the architectural style. For example, we can easily define a layered architecture by specifying that components at layer n can only use services provided by layer $n-1$. Similarly, we can define more advanced constraints on the system like those, for example, that govern the interactions behind the Model-View-Controller pattern. Each constraint is associated with its severity (e.g., strong and weak severity). The idea behind this distinction is that strong constraints must always hold at each modernization step (i.e, after any transformation), while weak constraints can be broken temporarily to enable further transformations of the system.

Finally, we propose to specify the behavior of components by using extended finite state automata. In particular, we propose to use them to model the reactions of components to method/operation invocations by specifying the effects on both their internal states and the other components (contacted through the required services). This way, we enable advanced reasoning techniques and simulations to better support component substitutability. If the whole system were modeled through a set of cooperating automata, we would be able to simulate its behavior and easily identify possible problems, even before performing the actual changes. Moreover, to find an alternative to a given component, we can reason on both syntactic properties (i.e., the set of offered/required interfaces), and desired behavior to foresee potential integration issues.

4.1 Preliminary evaluation

To demonstrate the applicability, and usefulness, of our approach, we implemented the COMO metamodel using EMF (Eclipse Modeling Framework [7]), and created a simple tool, on top of it, able to infer the COMO-like architecture

of J2EE applications. The tool heavily leverages the strongly-typed J2EE architecture to understand the boundaries of each element, and proposes a first draft architecture. Unfortunately, and to the best of our knowledge, no tool is able to analyze the source files of an application and produce the equivalent KDM model. Since we were mainly interested in assessing our extensions, and due to this lack, we decided to derive the high-level elements provided by the COMO metamodel from the source code directly, but the same approach (algorithm) can be rewritten by using information gathered from KDM. The result would be easier and neater, but out of the scope of our experiments.

This simple tool was to assess the validity of our proposal by automatically creating the structural model of JBilling. Even if the prototype is raw, the results are interesting. The overall model of *JBilling* is represented, using a UML-like formalism, in Figure 7. The outermost component represents the overall JBilling application, as a customer can download from the web site. Since the component has unsolved dependencies, this means that it cannot be used as-is, and other components —able to solve those dependencies— are required. In particular, JBilling requires the *JPA* service, able to manage the persistence of Java objects, the *Session* service, which manages the lifecycle of session beans, and a *SQL* interface for accessing a relational database. Analyzing these unsolved services, the user can find an adequate set of components able to satisfy the dependencies, enabling the *JBilling* component to work properly, providing the *Bill* service.

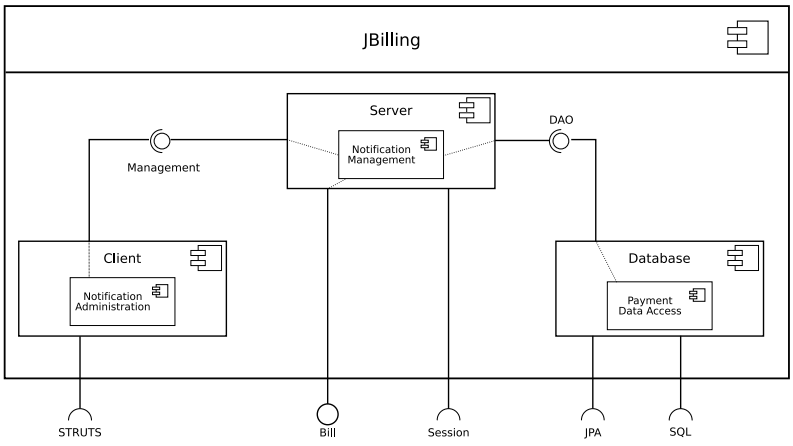


Fig. 7. Component-based view of the case study

Digging down into the internal structure, JBilling is composed of a `Client` part, a `Server` part, and a `Database` part. The diagram also explains the interactions among these components, since they are explicitly declared using the

services *DAO*, for the data access, and *Management*, for administering the Server. The `Server` requires a service to manage session beans, provided by someone outside JBilling, and a service for the data access, provided by the `Database` part. If such services are available, the application successfully provides both the *Bill* service and the *Management* service. Notice that the metamodel groups low-level dependencies among the internal elements of each component into high-level cohesive requirements, and thus simplifies the overall model.

We can also better define the internal structure of each part, and identify the sub-components able to manage the notification. Each internal component contribute to the overall services that the containing component offers or requires; this is rendered in the diagram through dotted lines. For example, if we consider the `Server` component, part of its `Management` interface is provided by the `Notification Management` component.

Using this model, it is possible to effectively tackle the modernization problem at a higher level, narrowing the changes into well-defined boundaries and reasoning in terms of which component should be modified to satisfy the final requirements. The aim is to foster a component-oriented modernization approach[3]. Initially users exploit the COMO metamodel, along with its KDM-compliant supporting tools, to define a first draft architecture of the application they want to modernize. Supporting tools let users refine automatically produced models by hand. All the results are stored in a knowledge repository, which provides versioning facilities, enhanced model search using ontologies, and a transformation library to support the actual modernization of the system.

After creating the initial model, ATL [8] transformations help modernize the application. These transformations help reassemble the structure of the system by merging, splitting, or deleting components. They can both work on the low-level KDM elements and on our components and interfaces. For example, it is possible to adapt the JBilling's `database` layer to the data abstraction currently used in the telephone company.

Users are also free to modify models by hand: the metamodel, and its contraints, ensure the consistency of produced results. Moreover, transformations can check whether these manual modifications on some components affect other neighbor elements. For example, we can easily query the repository for a component able to provide the required `session` interface, and find a proper application server. Since the retrieved component also provides the `JPA` interface, the rule checks the consistency among the parts and easily establishes the connection.

5 Related Work

The proposals related to our COMO metamodel can be grouped into two main groups: model-based techniques and architectural-oriented solutions.

As for the first class, Strein et. al. [9] propose an extensible metamodel for program analysis, able also to serve as basis for a flexible modernization frame-

[3] The modernization approach is the key research goal of the European STREP project MOMOCS (MOdel driven MOdernisation of Complex Systems, no. 034466).

work. They figure out that, when we are required to analyze a system, there is a set of principal tasks that must be accomplished: we need to extract information from the system, analyze it, and either show the results or perform a refactoring. This is true regardless the precise kind of analysis to perform; it only varies the level of abstraction of extracted information.

Their proposal leverages the loose coupling among front-ends for programming languages, analysis techniques, and refactoring tools. Each part can be plugged independently of the others, allowing good reuse, and also ensuring that the system is able to adapt itself to unforeseen requirements. For these reasons, they propose a metamodel, inspired by *abstract syntax trees*, composed of three parts: a *front-end specific* model is produced by ad-hoc information extractors, a language-independent *common model* is achieved by mapping front-end specific constructs by means of common equivalences, and some analysis-dependent *views* are generated from the common model.

Obviously, we can have several concrete front-end models, one for each input language we understand, related to a common model of the system, on which each analysis can calculate its own view. The authors show the feasibility of a model-driven approach to foster the modernization of existing systems based on a language-transparent metamodel. They also demonstrate the efficiency and scalability of the proposal by allowing the resulting framework to incrementally update models. Nevertheless, this work does not help developers outline the structure of the system: their focus is to provide a generic metamodel, usable as basis for creating new analysis tool on systems. Moreover, those result of such analysis will not be integrated with the base systems, and thus when the system is modified, (part of) the result of the analysis must be regenerated. For these reasons, it does not fit well with architectural models, which requires a more careful management. Our approach integrates the architectural model with the model of the system, allowing a better evolution of both.

Similarly, software architecture tries to lift the abstraction level on software systems, and shifts the focus from lines of code to coarse-grained architectural elements. Even if there is no universally accepted definition of what a software architecture is, Garlan and Shaw [10] say that:

> Software architecture involves the description of elements from which the systems are build, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

Architectural description languages (ADLs) provide conceptual means to let designers focus on the conceptual architecture of their system, without implementation details. We can use ADLs in different contexts, from the communication and understanding of a system, to advanced analysis techniques able to identify, for example, possible bottlenecks of the system. These different requirements led to the creation of many ADLs, surveyed in [11]. Garlan et al. tried to merge all main concepts in ACME [12], which is able to map architectural specifications from one ADL to another. The OMG introduced architectural concepts in UML 2.0 [13], and now they are widely spread and used in many

software design activities. These concepts also motivated *component-based development* [14] that aims to build systems by composing preexistent components. Reasoning in terms of offered and required interfaces allows us to glue together existing components and (easily) release new applications.

Typically these architectural views on the system are successfully used in forward engineering. For example, Baresi et al. [15] present an approach to verify whether an architectural style is adequate to satisfy the elicited requirements, and if it can guarantee an adequate level of flexibility. Moreover, *ArchJava* [16] permits us to embody architectural models of the system within the actual implementation, and use high-level operations to modify the current structure of the system. Finally, Roshandel et al. [17] present a way to manage the architectural evolution of the system by combining a configuration management system and typical architectural concepts.

Recently, the research in this domain tries to infer the architecture from existing systems. One of the most successful approach is DiscoTect [18, 1], which is able to identify the architectural structure of a system by knowing its architectural style and by monitoring its runtime behavior. The main problem of those approaches is their limited ability to handle the traditional backward-engineering issues.

6 Conclusions and Future Work

This paper proposes the COMO metamodel as enhancement of the OMG's *Knowledge Discovery Metamodel* to fully support component-oriented modernization. We start eliciting the main requirements behind the modernization of (complex) software applications, strengthening our idea to tackle this problem at component level, and then we introduce our metamodel.

Albeit the overall KDM structure is promising, we found that it can be improved with state of the art concepts borrowed from component-oriented development and software architecture. For this reason, the COMO meta-model is heavily based on components and interfaces and suggets how to represent all important information pieces. We suggest the declaration of reusable components that cooperate via well-defined interfaces.

Preliminary results are promising since the metamodel allows us to model properly the key structural elements of the case study. Moreover the identification of precise interfaces, which mediate the interactions among components, allows us to adopt a higher point of view on the system, simulate its behavior, and reason in terms of component substitutability. For the future, we plan to further elaborate general analysis techniques, able to semi-automatically identify components and interfaces, and better support the whole modernization process.

References

1. Yan, H., Garlan, D., Schmerl, B., Aldrich, J., Kazman, R.: Discotect: A system for discovering architectures from running systems. In: Proceedings of the 26th In-

ternational Conference on Software Engineering, Edinburgh, Scotland (23-28 May 2004)
2. Tan, Y., Mookerjee, V.: Comparing uniform and flexible policies for software maintenance and replacement. IEEE Transactions on Software Engineering (2005)
3. Object Management Group: Architecture Driven Modernization (ADM). `http://adm.omg.org/` (January 2006)
4. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. Software, IEEE **20**(5) (2003) 36–41
5. Gerbery, A., Glynnz, E., MacDonaldz, A., Lawleyy, M., Raymondy, K.: Modelling for knowledge discovery
6. OMG: Object Constraint Language (OCL). `http://www.omg.org/technology/documents/formal/ocl.htm`
7. : Eclipse modeling framework (emf). `http://www.eclipse.org/modeling/emf/`
8. Bzivin, J., Valduriez, P., Jouault, F.: ATL : Atlas Transformation Language. `http://www.eclipse.org/m2m/atl/`
9. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)-Volume 00 (2006) 380–390
10. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)
11. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Software Eng. **26**(1) (2000) 70–93
12. Garlan, D., Monroe, R.T., Wile, D.: Acme: An architecture description interchange language. In: Proceedings of CASCON'97, Toronto, Ontario (November 1997) 169–183
13. Object Management Group: Unified Modeling Language (UML), Version 2.0 (2005)
14. Lau, K.K., Wang, Z.: Software component models. IEEE Trans. Software Eng. **33**(10) (2007) 709–724
15. Baresi, L., Heckel, R., Thöne, S., Varró, D.: Modeling and validation of service-oriented architectures: application vs. style. In: ESEC / SIGSOFT FSE. (2003) 68–77
16. Aldrich, J., Chambers, C., Notkin, D.: Archjava: connecting software architecture to implementation. In: ICSE, ACM (2002) 187–197
17. Roshandel, R., van der Hoek, A., Mikic-Rakic, M., Medvidovic, N.: Mae - a system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. **13**(2) (2004) 240–276
18. Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., Yan, H.: Discovering architectures from running systems. IEEE Transactions on Software Engineering

Representing Legacy System Interoperability by Extending KDM

Vegard Dehlen[1], Frédéric Madiot[2] and Hugo Bruneliere[3]

[1] SINTEF ICT
P.O.Box 124 Blindern,
N-0314 Oslo, Norway
vegard.dehlen@sintef.no

[2] MIA-SOFTWARE
4, rue du chateau de l'Eraudière BP 72 438
44324 Nantes cedex 3
fmadiot@mia-software.com

[3] ATLAS (INRIA & LINA) - University of Nantes
2, rue de la Houssinière BP 92208 - 44322, Nantes, France
hugo.bruneliere@univ-nantes.fr

Abstract. The complexity of software systems is continuously growing. An important part of this complexity issue concerns the interoperability between existing systems (i.e. legacy systems), where problems often occur due to heterogeneity in e.g. data, involved technologies or models. The Knowledge Discovery Metamodel (KDM) standardised by the Object Management Group (OMG) facilitates representation of existing systems, allowing them to be treated in a homogenous way at the model abstraction level. This paper defines a language suitable for modelling interoperability between these systems by extending KDM and introducing concepts that are specifically aimed at representing relevant interoperability information.

Keywords: interoperability, knowledge discovery, KDM, MDE

1 Introduction

The complexity of specifying interoperation between legacy systems is still a widely open issue. Their heterogeneity and distribution characteristics make them often difficult to monitor, analyse and understand. As a consequence, interoperability is difficult to represent and thus to handle There are currently many projects addressing this issue, such as the EU-funded Modelplex project which this work is part of [5]. The aim of this paper is to focus on the representation of the legacy system interoperability and to show how the Model-Driven Engineering (MDE) approach (and more especially metamodelling) is used to provide a solution in this particular context. The overall idea behind this is to be able to go from the usually

heterogeneous world of systems to the homogeneous world of models in order to deal more easily with complexity.

This paper is organized as follows. Section 2 describes the main system interoperability characteristics. Section 3 is a precise definition our KDM extension for modelling systems interoperability. Section 4 concludes on the benefits of such a metamodel.

2 System Interoperability

IEEE defines interoperability as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged" [1]. Research on enterprise and service interoperability has received considerable attention as reflected in projects such as IDEAS[1], INTEROP[2] and ATHENA[3]. Information & Communications Technologies.

Previous research has acknowledged that a system's specification should be separated from its implementation. An *interface specification* serves both as a contract the system implementation must adhere to as a well as telling other systems how to interoperate with it. Traditionally, two main levels of interoperability have been distinguished [4]:

- **Signature level**. The signature (or *static*) level simply deals with the signatures of operations, i.e. names, parameters and return types.
- **Semantic level**. The more complex semantic (or *dynamic*) level deals with the meaning of operations, i.e. operational semantics, pre/post conditions, behavioural aspects of systems etc.

In addition to these two levels, the way complex systems interoperate generally relates to a combination of interoperation types. Some combinations conform to *patterns* that can be described independently of the technology. For instance, [2] have described 65 of these patterns related to integration of applications.

Within this paper, we are going to consider both the expression of *interoperability patterns* and the representation of the *interoperability relationships* corresponding to their application.

---

[1] IDEAS - Interoperability Development for Enterprise Applications and Software, IST-2001-37368

[2] INTEROP Network of Excellence - Interoperability Research for Networked Enterprise Applications and Software, FP6 508011. URL: http://interop-vlab.eu/

3 ATHENA - Advanced Technologies for Interoperability of Heterogeneous Enterprise Networks and their Application. URL: http://www.athena-ip.org/

3 The Interoperability Knowledge Discovery Metamodel (IKDM)

3.1 Identified Interoperability Concerns

As a part of a generic MDE solution to the described problem, this article proposes Interoperability Knowledge Discovery Metamodel (IKDM). The metamodel has been defined as an extension of the OMG Knowledge Discovery Metamodel (KDM), which is a MOF-compliant metamodel for "representing information related to existing software assets and their operational environment" [3]. Thus, the goal of KDM is to provide a common structure that facilitates interchange of data and models of legacy systems.

Based on the different types of interoperability described in Section 2 and communication with the industry partners in the Modelplex project, three main concerns to focus on have been identified:

1. *The interoperability relations between parts of each system*: the parts of the systems involved in the interaction and the role they play.
2. *The decomposition of these interoperability relations*: the sub-relations describing a relation at a lower level of abstraction.
3. *Definition of interoperability patterns*: relations between systems concerning different interoperability patterns.

By crossing the two first concerns, each kind of interoperability relationship between the top level components can be represented, and each relationship can be navigated in order to reach the primary dependencies. As a result of the third concern, the interactions between elements can then be described through a relation between the participants in the existing systems and the pattern they match with. This way of describing interoperability is independent of the kind of system and generic enough to be able to express many different possible kinds of interoperability.

3.2 The metamodel

To allow describing a large number of interoperability relationships between elements of an existing system, the concept of interoperability relationship patterns has been introduced. Such a pattern can be defined in IKDM by using three classes:

- **InteroperabilityPattern** defines a generic kind of interoperability relationship (File Transfer, Method Invocation, etc). An interoperability pattern is composed of roles and sub-patterns. Sub-patterns allow refining the pattern by decomposition.
- **InteroperabilityRole** defines the role that elements of an existing system can play in an interoperability pattern (sender, receiver, etc).
- **InteroperabilityProperty** defines properties that can characterize patterns and roles (filename, size, index, etc).

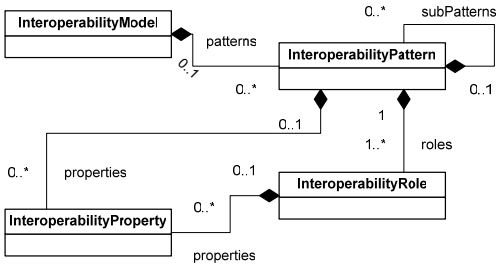


Fig. 1. IKDM Metamodel ("Pattern Expression" Part)

The interoperability patterns are used to characterize the way elements of a KDM model participate to an interoperability relationship. This kind of relationship is described with three classes:

- **InteroperabilityRelationship** defines a relationship between elements of an existing system according to a defined pattern. The aim of this class is to describe relationships at a higher level of abstraction than KDM relationships. Generally, they will be deduced from those natively described in the existing system.
- **InteroperabilityRelationshipEnd** defines the end of an interoperability relationship. It references an instance of the model element representing an aspect of the existing system (described with KDM) and an instance of *InteroperabilityRole* defining the role played by this element in the interoperability pattern. The *InteroperabilityRelationShipEnd* instances are the interface with the model of the existing systems.
- **InteroperabilityPropertyValue** defines the value for a given property set to a relationship or an end.

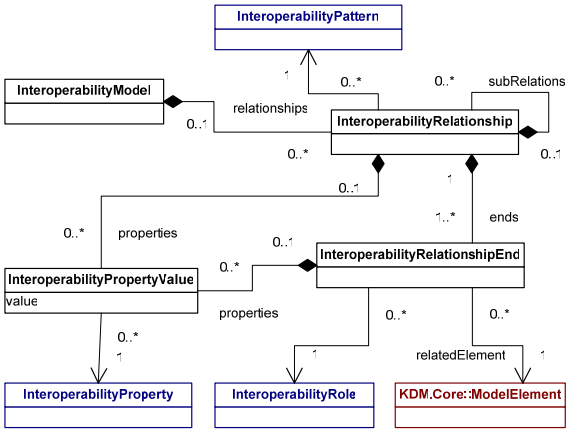


Fig. 2. IKDM Metamodel ("Interoperability Representation" Part)

IKDM is intended to support the main characteristics of interoperability, as described in Section 2. The concept of interoperability pattern (as defined within this section) has been designed to be generic enough in order to allow the description of multiple interoperability dimensions.

4 Conclusion

The work presented in this paper describes the Interoperability Knowledge Discovery Metamodel (IKDM), a KDM metamodel extension dedicated to modelling interoperability between existing systems (i.e. legacy systems). The main characteristics of system interoperability were first summarized and the IKDM metamodel, as a proposed answer to them, was then detailed.

The main contribution brought by this paper and its underlying work is the developed interoperability-specific KDM metamodel extension called IKDM. This metamodel, based on the KDM OMG standard, is particularly useful when modelling interoperability patterns and the results of their applications on real legacy systems, i.e. when modelling the interactions within a given complex system or between different systems.

The main benefit of IKDM is the possibility to define tools, based on or supporting this generic metamodel, that have the ability to manipulate models (i.e. IKDM models) independently of the type of interoperability they represent. In addition, it will be possible to identify the main interoperability patterns and express them as models in order to provide reusable libraries of such patterns.

The work will be continued during the coming months of the second phase of the Modelplex project, mainly by implementing a concrete scenario involving semi-automatic discovery of IKDM models from legacy system.

Acknowledgments. The work presented in this paper has been done in the context of the EU FP 6 project Modelplex, funded by the European Commission.

References

1. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. Institute of Electrical and Electronics Engineers (1990)
2. G. Hohpe and B. Woolf: Enterprise Integration Patterns. Addison-Wesley (2004)
3. KDM Final Adopted Specification ptc/06-06-07
4. A. Vallecillo, J. Hernández and J. M. Troya: Component interoperability. Tech. Rep. ITI-2000-37, Departmento de Lenguajes y Ciencias de la Computacion, University of Malaga (2000)
5. Modelplex – MODELling solution for comPLEX software systems, IST 34081. Official website: www.modelplex-ist.org

In the MOMOCS Tool Suite: a XIRUP Transformation Tool to achieve complex system modernization

Alessandra Bagnato[1], Marco Serina[2]

[1] TXT e-solutions
Salita San Barborino 23r Genoa, Italy, 16100
alessandra.bagnato@txt.it

[2] TXT e-solutions
Via Frigia 26 Milano
Marco.serina@txt.it

Abstract: The research project MOMOCS aims at studying a methodology and related tools for fast reengineering of complex systems. A complex system is characterized by an interconnection of hardware, software, user interfaces, firmware, business and production processes. MOMOCS will study how a complex system can be modernized with a focus on the software portion of it, with the goal of keeping up with a very fast changing business and technical environment taking human beings as the centre of the interaction. This paper describes how TXT e-solutions has implemented a Transformation Tool for modernisazation problems based on the MOMOCS XIRUP methodology [XIRUP]. Fist we introduce the as-is situation in ATL Transformation rules. Then we discuss the results produced by the research.

1 Introduction

Modernization rapidly gains increasing importance for software development [NEPTUNE].

MOMOCS aims at studying a methodology and related tools for fast reengineering of complex systems. A complex system is characterized by an interconnection of hardware, software, user interfaces, firmware, business and production processes.

MOMOCS will study how a complex system can be modernized with a focus on the software portion of it, with the goal of keeping up with a very fast changing business and technical environment taking human beings as the centre of the interaction

This paper discusses the approach of transformation modernization in the MOMOCS project [MO08], it first outlines how the Transformation Tool has been designed in the MOMOCS context and its expected benefits then it describes a practical XIRUP Designer Use Example for called and matched transformation rules [ATL] and then it details the XIRUP2XIRUP Transformations and the identified transformation patterns.

2 The Transformation Tool in the MOMOCS Context

The XSM Transformation tool is based upon the language and functionalities provided by the Eclipse M2M ATL project [ATL]. Its main goal is encapsulating and improving those functionalities in order to transform "To Be Modernized" models into suitable "Modernized" models which are compliant to user's modernization requirements. For more references and documentation, please also refer to ATL documentation [ATL_DOC]

The ATL language and the XIRUP metamodel are in the Transformation Tool two instruments to be put together.

ATL (ATLAS Transformation Language) is a model transformation language and toolkit developed by the ATLAS Group (INRIA & LINA). In the field of Model-Driven Engineering (MDE), ATL provides ways to produce a set of target models from a set of source models. [ATL].

XIRUP is the methodology studied in the MOMOCS project to solve the dilemma between rigorous and bureaucratic methodologies and agile and unstructured ones by building an eXtreme end-User dRiven Process (XIRUP), and related supporting tools. In particular, the Xirup metamodel is the reference MOMOCS metamodel, providing a component -based view of complex systems with the purpose of raising the abstraction level and make their comprehension easier.

The usefulness of the Transformation Tool in the MOMOCS context is to aid the evolution of a TBMS model into a MS model performed by following the Xirup methodology [MOMOCS].

The tool guides the user through the transformation processes and takes as an input:

* A XIRUP System Model (XSM) describing the To Be Modernised System (TBMS) or a partial modernized version of it

* The reference XIRUP Metamodel

The output of the Transformation Tool will be:

* A set of transformations applicable to the input XSM

* An XSM which can be a fully or partially "Modernized System" (MS) model, according to the complexity of the modernization process and the way it is faced.

3 Transformation Tool Approach and Expected Benefits

The XSM Transformation Tool provides a set of functionalities to improve the editing of ATL transformations. It mainly addresses the activities related to the creation and definition of transformations and rules among models which are compliant to the MOMOCS reference metamodel.

Main contributions concens features for:

- Graphically designing "matched" and called "rules" for the Xirup metamodel
- Adding ATL syntax templates
- Quickly using ATL operations
- Commenting a transformation

3.1 Graphically designing matched and called rules for the Xirup metamodel

The XSM Transformation Tool provides a graphical features to initialize both *matched* and *called* rules to be used with Xirup metamodel elements. These rules correspond to the two different ATL programming modes: declarative and imperative ones.

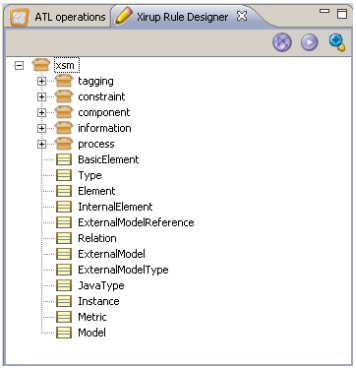


Figure 1: XIRUP Rule Designer View

The Xirup designer view shows the set of Xirup metamodel elements in a tree-structured way.
By clicking on the displayed elements the user can select the items that will be the object of the rule; the designer allows initializing a called rule for one element and a matched rule for both one and two elements.

A two-level detail description is also implemented: while setting up the rule, a user can decide if structural features such as attributes and references are going to be managed in the rule itself or not. This is achieved by clicking on the proper "detail" toolbar button.

3.2 Adding ATL syntax templates

The XSM Transformation Tool also provides a set of ready-to-use templates in order to quickly inserting main ATL language constructs. These constructs are:

- Helper
- Matched rule
- Light Matched rule – reduced pattern with the most used keywords only
- Called rule
- Light Called rule – reduced pattern with the most used keywords only
- Imperative expressions:
  - If
  - For
- OCL declarative expressions:
  - If
  - Let

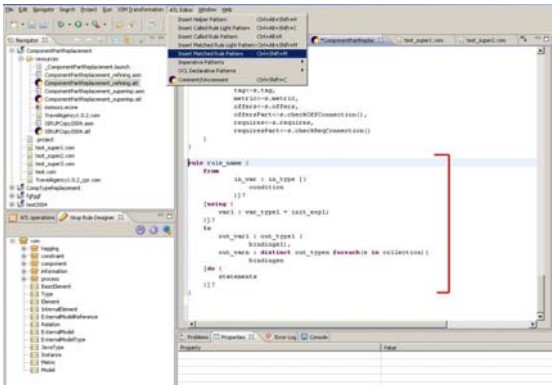


Figure 2: Using construct templates

They can be easily inserted thanks to menu actions to which also key-binding features are associated. This way, users (especially the first times they're using ATL transformation) don't have to keep on checking manuals and documentation to get references for right syntax.

3.3 Quickly using ATL operations

Apart from general constructs used to express both declarative and imperative instructions, the ATL language defines a set of both OCL-derived and characteristic operations. They are organized as follows:

- OclAny operations
- Boolean Data Type operations
- String Data Type operations
- Numerical Data Type operations
- Operations on collections
- Sequence Data Type operations
- Set Data Type operations
- OrderedSet Data Type operations
- Bag Data Type operations (defined by OCL specification but not still available with current ATL implementation)

The XSM Transformation Tool provides a tree-structured Eclipse view showing all main ATL operations organized according to the separation presented in the bullets above. Each item is associated to a tip describing the item itself and thus giving information about the operation it represents: user's double-click will trigger the insertion of the correct operation syntax into the ATL file which is actually in editing mode. This view can be thus considered an "active" manual reference provided by the tool to make the process of creating and editing transformations faster.

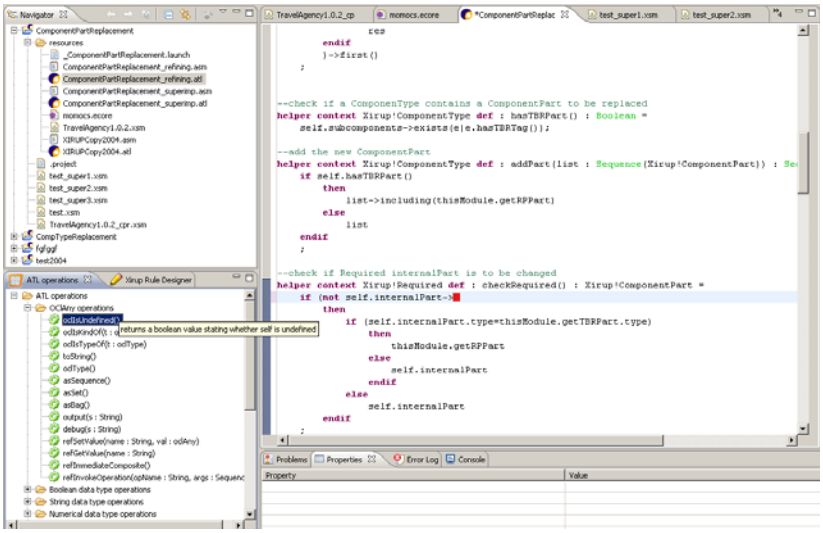


Figure 3: Using "ATLoperations" view

3.4 Commenting a transformation

Since it could be useful adding some extra information to each transformation file that has been created and edited, a user can add personal comments by means of a special property page. The content is then attached in a persistent way to the file itself.

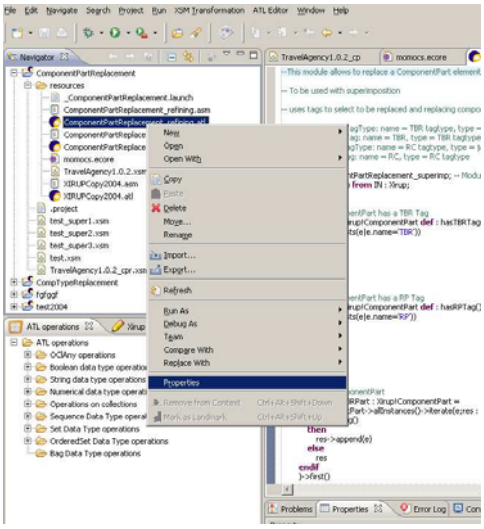


Figure 4: Accessing property pages

A property page is thus displayed and the user can comment the transformation to better identify its contents.

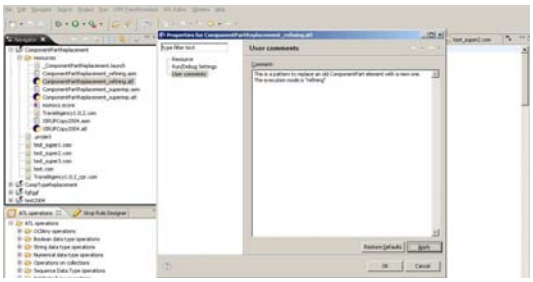


Figure 5: Commenting a transformation

4 A XIRUP Designer Use Example for called and matched rules

For example, a called rule on the *ComponentPart* element can be initialized as follows:

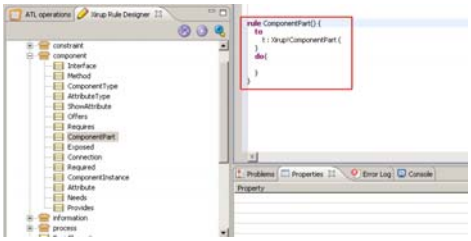


Figure 6: Choosing a *called rule* for *ComponentPart* elements

This is a statement to define imperative instruction to be associated to a certain kind of element (in this case to a *ComponentPart*). If a strong knowledge of the metamodel and all it's features is not mastered by the user, the rule can be generated with all attributes and references that are distinctive of that metamodel element, subject of the rule.

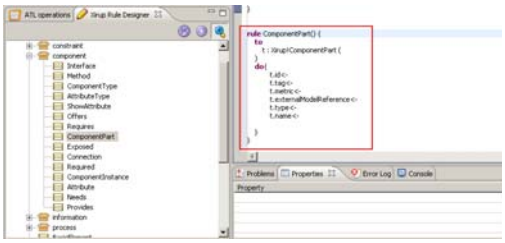


Figure 7: Zooming in the detail level for a *Called rule*

Regarding matched rules, the process is similar, but here a user can decide to select either one element (it matches itself according to current requirements) or two elements (a source element is matched to a different target element).

Suppose, for example, that *ComponentType* elements have to be checked in order to update the list of their subcomponents.

The user opens the Xirup Desiger view and selects the *ComponentType* item in the proper package (in this case *componet*); then he/she can click on the *matched rule* view button which triggers the rule initialization (Figure below).

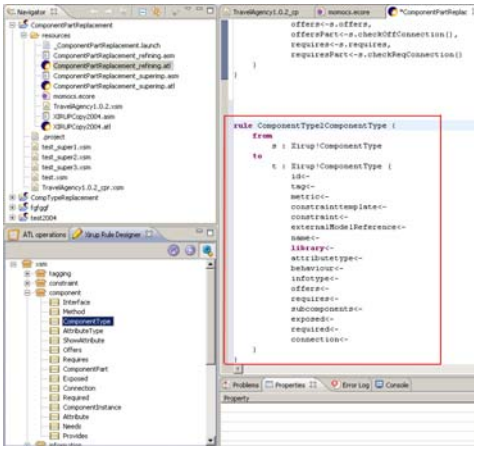


Figure 8: A *matched rule* for *ComponentType* elements

At this point, the rule is ready to be completed and customized according to the scenario. In this case, the structural feature *subcomponents* is being upgraded to address the replacement of an old *ComponentPart* with a new one, while the rest of other attributes and relationships don't need to be changed (Figure below)

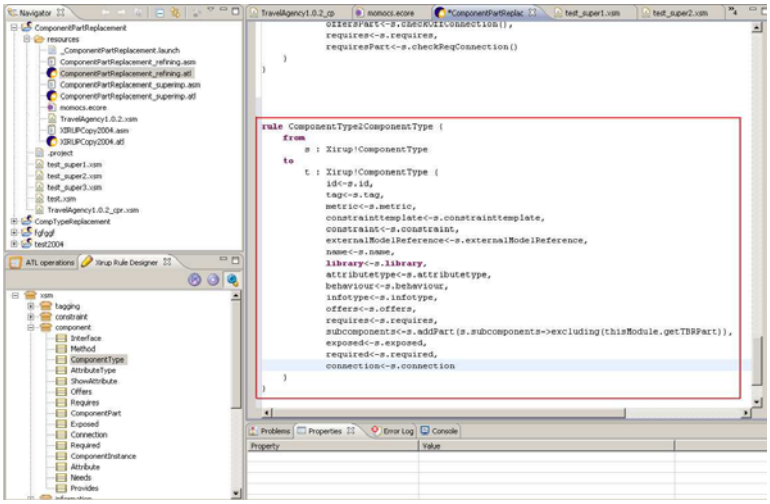


Figure 9: Customizing the *matched rule*

An online help has been provided along with the set of XSM Transformation Tool plug-ins, describing its functionalities and concepts behind.

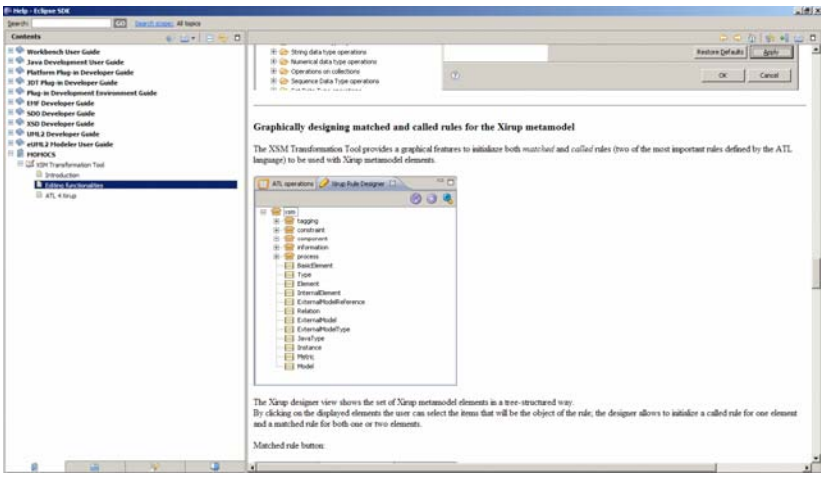


Figure 10: Online Help

5 XIRUP2XIRUP Transformations

At present, more than one ATL technique exists in order to transform a source model into a target model and, additionally, two ATL compilers can be used to perform a transformation (see ATL user manual for references): this means different possibilities with different pros and cons to pursue the same task.

That said, the XSM Transformation Tool must give to users the chance to choose among these several possibilities and select the best solution that suit to a particular scenario. Basically, when creating and defining a transformation, a user can proceed in two ways:

1. declare each rule so that the ATL engine allocates only those model elements that are explicitly mapped by a rule (default mode)
2. refine a model so that the ATL lets the developer map only model elements he/she is interested in and automatically allocates all other elements that have not been explicitly managed by user rules (refining mode)

According to scenarios, (1) or (2) can result more suitable each other.
It should be clear that with (1), the developer has a full control over the whole transformation process, thus he/she must explicitly manage every model element in order to keep the source model consistent with the target model: not doing that would mean generating only those elements that have been specified in user rules, with an evident loss of information.

This leads to the concept of "Copy transformation": a copy transformation explicitly manages, thus generates, all model elements and permits the user to focus only on those that have to be changed for modernization purposes (this is strictly correlated to the so called technique superimposition [ATL_SUPER]).

On the other side, (2) implements a similar approach as (1) but in an automatic way since the ATL engine does the "copy" in place of the user. Note that, due to current execution semantics of the refining mode, some specific precautions still have to be taken by developers. It may be required, for example, when designing an ATL module in refining mode, to specify additional explicit rules in order to make sure that all source model elements are transformed into their corresponding target model elements.

The brief description above, aims at letting the reader understand that, according to the scenario, type of elements to be managed and user preferences, (1) and (2) can be preferred to each other. If (2) could be easily managed by developers but it requires some precautions, (1) should require a Xirup copy transformation to be defined, but it warranties a complete control over the transformation process.

Giving the amount of Xirup metamodel elements, the work done for the XSM Transformation Tool also includes that copy transformation, preventing end-users from defining it by themselves.

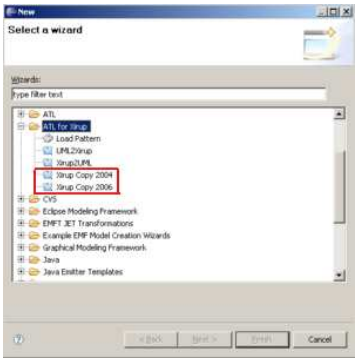


Figure 11: Accessing XirupCopy transformations (1)

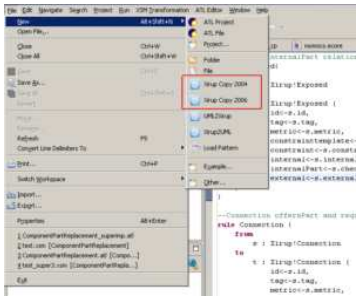


Figure 12: Accessing XirupCopy transformations (2)

XirupCopy2004 transformation consists of more than forty rules intended to copy all the information from source to target models: at this point, the user can directly modify that transformation by creating, deleting and changing rules in order to generate a (partly or fully) modernized output model without worrying about the rest, namely model elements that must be kept unchanged.

If the result is satisfactory, the modified XirupCopy transformation (actually no more a "XirupCopy") can be stored to the KB Repository: whenever the user will need a new XirupCopy transformation, a new wizard can be used. In addition to a direct modification, a new transformation can be created and "superimposed" to the XirupCopy transformation.
Superimposition means that "while ATL transformation modules and queries are normally run by themselves, that is one transformation module or query at a time, it is also possible to superimpose several transformation modules on top of each other. The end result is a transformation module that contains the union of all transformation rules and all helpers, where it is possible for a transformation module to override rules and helpers from the transformation modules underneath" (from [ATL_SUPER]).

After having imported the resources, creating and modifying transformations according to current needs, the user can run the transformation basing upon the launch configuration stub generated automatically by the wizard

XirupCopy 2006 wizard provides the same functionalities as XirupCopy 2004 one, but the embedded copy transformation is written to be executed with ATL compiler 2006.Additionally, the ATL2006 compiler introduces new functionalities (such as rule extension mechanism) but it still doesn't support some features, for example the refining mode itself. As a consequence, two Xirup copy transformations have been defined: one for ATL2004 compiler and one for ATL2006 compiler.

These two transformations are embedded within the XSM Transformation Tool and can be always retrieved by means of proper wizards. They can be accessed by selecting *File->New- >XirupCopy 2004/2006* or *File->New->Other->ATL4Xirup category* or the *XSM Transformation menu*.

6 Transformation patterns

The purpose of transformation patterns is providing an approach to define transformations in a way that makes their reuse easier.

Figure 13: *ComponentPart* replacement pattern graphical explanation

This is achieved through the tagging mechanism supplied by the Xirup metamodel: by using tags, model elements can be identified and updated according to the rules specified in the transformation.

A pattern is thus an ATL transformation describing a particular modernization task and exploiting special tags (how to use these tags is included in each pattern description) to identify the model elements to be taken into account.

For example, a *ComponentPart* replacement pattern recognizes a "To be Reaplaced" ComponentPart element if it contains a tag named "TBR".

At the same manner, it can identifiey a replacing ComponentPart element by means of a "RP" named tag

This way, non-ATL experts don't have to write rules by hands: they have just to tag interested elements and run the pattern.

A wizard allows to access patterns and use them; a combined graphical and textual explanation makes the user actually understand how to use a selected pattern and when.

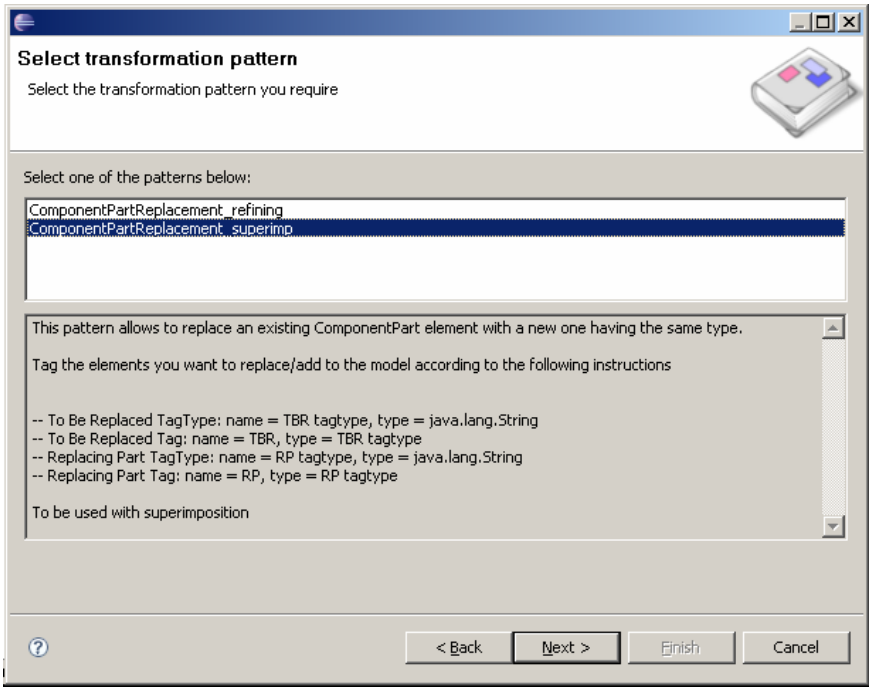
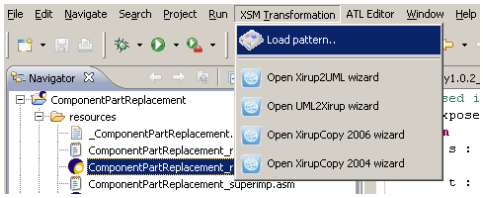


Figure 14: Choosing a pattern

The pattern wizard can be accessed through the proper *XSM Transformation* menu (or toolbar action button) or by selecting *File->New- >Load Pattern*:



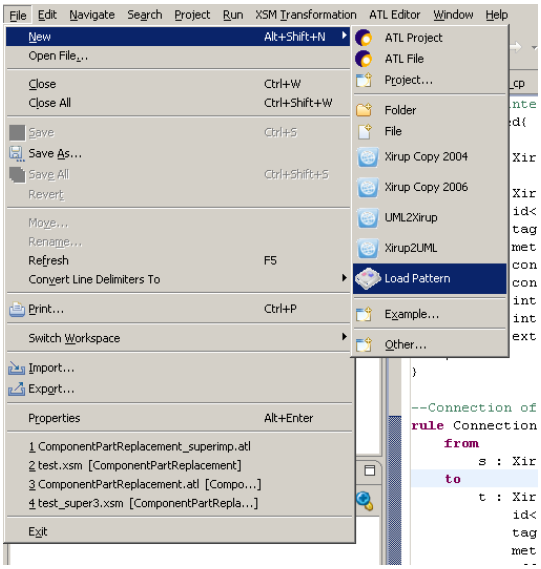


Figure 15: Accessing "Patterns" Wizard

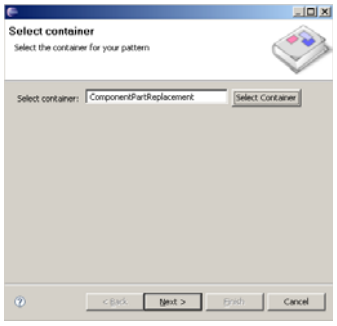


Figure 16: Selecting the container where to store the pattern

7 Conclusion

The aim of this research work is providing an approach to exploit, improve and especially glue ATL together with the Xirup modernization and its reference metamodel through a tool specifically thought for ATL transformation for modernisazation problems based on the XIRUP [MO08] metamodel, actually promoted by the MOMOCS project.

8 Acknowledgement

This work was partly supported by the European Commission (IST-2006-034466-MOMOCS).

9 References

[MO08] MOMOCS Project Homepage http://www.momocs.org
[XIRUP] MOMOCS consortium. D31 XIRUP Methodology Specifications, MOMOCS Project, June 2007
[NEPTUNE] Alessandra Bagnato et al. MOMOCS: MDE for the modernization of complex systems, Neptune Conference, 2008
[TXT]TXT Group Homepage http://www.txtgroup.com/
[ATL] ATL (ATLAS Transformation Language) http://www.eclipse.org/m2m/atl/
[ATL_DOC] ATL User Manual:
http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf
[ATL_SUPER] ATL superimposition http://wiki.eclipse.org/ATL_Superimposition