CaCOphONy:
Metamodel-Driven Software Architecture Reconstruction

Jean-Marie Favre
Adele Team, Laboratoire LSR-IMAG
University of Grenoble, France
http://www-adele.imag.fr/~jmfavre

Abstract

Far too often, architecture descriptions of existing software systems are out of sync with the implementation. If they are, they must be reconstructed, but this is a very challenging task. The first problem to be solved is to define what "software architecture" means in the company. The answer can greatly vary, especially among the many stakeholders. In order to solve this problem, this paper presents CacOphoNy, a generic metamodel-driven process for reconstructing software architecture. This paper provides a methodological guide and shows how metamodels can be used (1) to define architectural viewpoints, (2) to link these viewpoints to existing metaware and (2) to drive architecture reconstruction processes. The concepts presented in this paper were identified over the last decade in the context of Dassault Systèmes, one of the largest software companies in Europe, with more than 1200 developers. CacOphoNy is however a very generic process pattern, and as such it can be applied in many other contexts. This process pattern is in line with the MDA and ADM approaches from the OMG. It also complies with the IEEE Standard 1471 for software architecture. A megamodel integrating these standards is presented.

1. Introduction

> *" I define architecture as a word we use when we want to talk about design but want to puff it up to make it sound important". Martin Flower [36].*

What is software architecture? Defining "Software Architecture" (SA) led to extensive debates in the academic community during the 90's. There was no clear outcome because each research group attempted to put emphasis on the perspective it dealt with. Much of the effort went to the study of the "components & connectors" view and to the promotion of so-called Architecture Description Languages (ADLs). Unfortunately, these languages have failed to find their path to industry. *Software industry is still code-centric*. Descriptions of software architecture remain inexistent or informal. When SA descriptions exist, they soon become out of sync with the code. So, whatever software architecture is, it is missing in practice. That's why evolving very large scale software is so difficult.

To cope with this problem, the Reverse Engineering (RE) community has focused on architecture reconstruction, trying to extract architectural views from source code (e.g. [25][41][36]). For instance early work such as the C Information Abstraction (CIA) from AT&T attempted to extract dependencies between source files. Then a large body of work was dedicated to the design and evaluation of clustering algorithms to identify meaningful structures in complex dependency graphs (see [25] for an extensive survey of component recovery techniques). While useful in some situations, this body of work also considered software architecture from a single perspective.

It then appeared that the notion of software architecture was not so simple to define. In fact, in many research conferences and workshops there were some animated discussions about a proper definition for software architecture. This sometimes turned to a cacophony, everybody coming with its own definition of SA. At the time of writing this paper, 165 definitions of software architecture have been collected by the SEI (see www.sei.cmu.edu/architecture/definitions).

Beyond specialized formalisms, specific techniques and global definitions, more pragmatic approaches attempted to study the reality of SA, as actually practised daily in large software companies (e.g.[19][22][36]). This kind of work led to the recognition of the following facts: (1) the "components & connectors" covered by ADLs represented only one particular aspect of SA, (2) the same was true for the structural dependencies studied by the RE community, and (3) giving a precise definition of software architecture was not considered as a topmost priority in industry.

In 2000, the IEEE Standard 1471 ("SAStd" for short), eventually established the fact that software architecture could not be considered outside of its context [23]. This standard is technology-free. It brings no concrete solutions. It "just" provides a set of concepts with some "recommended practices". This standard is often consider as very weak or too abstract by academics eager of well-defined formalisms or specific algorithms. It gives nevertheless an overall view on SA in context. The SAStd, which was subject to intensive reviews by over 150 international reviewers before its publication, put a strong emphasis on the relationships between stakeholders, their

Working Conference on Reverse Engineering (WCRE'2004)
Delft, The Netherlands, November 8th-12, 2004

concerns and relevant architectural viewpoints. Simply put, this standard recognized that there is no such thing such as a standard definition for Software Architecture.

"So, what is Software Architecture? We are not sure, but we know one when we see one." IEEE 1471 FAQ [23]

Figure 1 "So, what is Software Architecture?"

The core contribution of the SAStd has been to show that the notion of software architecture heavily depends on stakeholders and their concerns. As pointed out by Fowler, architecture is best seen as a social construct.

"It is hard to tell people how to describe their architecture. 'Tell us what is important.' Architecture is about the important stuff. Whatever that is. " Ralph Johnson [17]

"Whether something is part of the architecture is entirely based on whether the developers think it is important. Architecture is a social construct because it doesn't just depend on the software, but on what part of the software is considered important by group consensus ". Martin Fowler [17]

Figure 2 "Architecture is about important stuff"

These points of view are close to the one expressed by Holt in the paper "Software Architecture as a Shared Mental Model" [22]. Architecture is about communicating and sharing important stuff. We believe that the notion of "group consensus" identified by Fowler is important. But though a "group consensus" might exist in a small or medium company, this is certainly more difficult to achieve in very large software companies with hundreds of software engineers [7][44]. In such a company, various groups of stakeholders will have a different definitions of SA, depending on their jobs and concerns. This does *not* mean that the chaos is there. On the contrary, this just mean that people consider software architecture from a different point of view. Though each pair of groups could work in symphony, looking at the whole naturally lead to an impression of cacophony. But who cares? Each group can have its own definition of SA, as long as it can communicate smoothly with other groups when required.

The first steps in reconstructing the software architecture of a very large software are therefore (1) to identify which group of stakeholders is the target of the reconstruction effort, (2) to formalize what software architecture really mean for this particular group, and (3) to establish which viewpoints are relevant for their particular concerns.

This fact was recognized by O'Brien and his colleagues from the SEI [36]. Thanks to their extensive experience in architecture reconstruction, these authors established a catalogue of *practice scenarios*, one of these being the "View-Set Scenario", or VSS for short (Figure 3).

Both the SAStd [23] and the catalogue of practice scenarios [36] provide conceptual frameworks identifying recurring issues in software industry. They do not bring particular methodologies for architecture reconstruction. This paper tries to fill this gap. *The VSS is seen as a primary specification of the problem addressed here*.

Name:
The view-set scenario covers the identification of architectural views that sufficiently describe a software system.
Problem:
The problem is to determine which architecture views sufficiently describe the system and cover stakeholder needs.
Desired Solution:
The desired solution consists of a method to determine the relevant architecture view set for a particular system.

Figure 3 Excerpt of the "View Set Scenario", [36] p.4

This paper gives an overall view on Ca*c*Oph*o*Ny, a metamodel-driven methodology for software architecture reconstruction. Ca*c*Oph*o*Ny is a generic process that integrates (1) Reverse Engineering concerns, (2) the IEEE 1471 std [23], and (3) Model Driven Engineering [26][33] [35][37].

The reconstruction process presented in the second half of this paper was presented at various workshops [9][14][15], including the Dagsthul seminar on software architecture. The Symphony approach also comes from this context, and in fact [3] can be seen as a companion paper. Many ideas are shared, in particular the idea to define a generic process for architecture reconstruction. The global structure of the processes proposed are very similar and the reader is thus invited to refer to [3] to get more information about the process and related techniques. Here we take a abstract perspective on software architecture reconstruction but we go much further by integrating SA and MDE.

Why MDE? Though one should not really care about informal definitions of software architecture, building operational tools requires the provision of very precise definitions of significant architectural concepts. And this, whatever these concepts are. Metamodels are keys for that.

Symphony does not recognize the importance of metamodels, and as a result the work presented in [3] remains in the niche of software architecture reconstruction. By contrast, MDE is much broader in scope. Integrating MDE and SA is much more promising because MDE has the potential to cover the whole spectrum of software engineering processes, and this in an integrated manner. The thesis of this paper is that software architecture reconstruction can just be seen as a particular MDE processes.

Model-Driven Engineering is just about *productive metamodels*, an approach in which metamodels are not just words, boxes and arrows, but productive means [35][37]. This paper shows that metamodels can be used

- (1) to give a precise and operational definition of architectural viewpoints, so that tools can be build;
- (2) to link abstract concepts with existing metaware (see below), so that existing assets can be reused,
- (3) to drive whole reconstruction processes, whatever the definition of software architecture used.

Emphasis should be put on the fact just like Symphony, Ca*c*Oph*o*Ny is a methodology and generic process pattern.

CaCOphONy is linked to a particular set of techniques, and following [3], this paper is technology-free. Just like the MDA approach from the OMG and the IEEE 1471 standard, the focus in this paper is on the articulation of essential concepts. Almost all previous works differ from CaCOphONy and t in that they address a determined goal, concrete techniques, and a certain fixed sets of views to be reconstructed, whereas CaCOphONy and Symphony provide a general reconstruction method [3].

The remainder of this paper is structured as following. The background for this work is presented in section 2. Then section 3 gives the keys to integrate the SA and MDA standards. Reverse engineering concerns are then integrated in section 4 along with the notion of metaware. Then an overview of the CaCOphONy process is given in section 5. Finally section 6 concludes the paper.

2. Background

The best thing to learn about software architecture reality is be to leave the lab, to go in large software companies, and there to observe what is going on. The Symphony approach [3] is based on the experiences of various authors about SA developed in large companies such as Nokia [41] and Siemens [19]. We followed a similar approach over the last decade and we also went to large software construction fields. In particular we spent 7 years observing the evolution of a very large scale software developed by Dassault Systèmes (DS) [7], one of the largest software companies in Europe. The figure below, extracted from [42], gives a better idea of what large software means in this paper.

Company: Dassault Systèmes,
World leader in CAD/CAM
+1200 software developers
Main software: CATIA, DELMIA and ENOVIA
20 years of partnership with IBM which distributes the software
Main customers: Boeing, Daimler Chrysler, Sony…
+ 19 000 clients over 60 countries
+ 500 000 seats
Many software development partners
Software Product
+ 8 MLOC in C++, Fortran, Java, Visual Basic…
7 platforms supported including Unix and Windows,
20 years of continuous software evolution
Software Architecture
A proprietary component technology called the OM
+ 70 000 classes
+ 8 000 OM components and 5 000 OM interfaces
+ 3000 dynamic libraries
+ 140 products in the portfolio
+ 800 frameworks

Figure 4 Key figures for a Large Software Product

During our collaboration with DS we learned 3 main things about very large scale software architecture:

- (1) The notion of software architecture is really more complex than the academic vision tends to explain.
- (2) The nature of software architecture really depends on the culture of the company and on its know-how.
- (3) This know-how is in part intangible knowledge in the head of key people in the company, but it is also materialized by a large range of tools and repositories used to support the software process [12].

This last point is fundamental in the context of this paper. In DS, the *Tool Support Team* is in charge to provide tools and process support to developers of software applications. Though the notion of software architecture and architectural viewpoints are not materialized, we found plenty of tools and repositories developed internally [7]. These tools were built to deal with tests, bugs, frameworks, documentation items, modules, features, products, solutions, etc. The exact meaning of these terms belongs to DS architectural know-how and is not relevant here. The goal of this paper is by no means to explain the notion of architecture as practised within DS. Key ideas has been already published and can be found for instance in [6][8][42][43]. A complete description of DS architectural know-how would certainly require a whole book, just as the description of software architecture practices within Siemens led to [19].

The goal of this paper is instead to provide a *generic* approach that help in (1) defining what software architecture really means in the context of a given company, and (2) effectively supports architecture reconstruction tasks. The reconstruction process performed in the context of Dassault Systèmes is an example of this approach [8].

3. Keys for MDE and SA integration

The distinctions Model/Metamodel and View/Viewpoints are keys to understand how MDE and SA can be integrated.

3.1. MDE : Models vs. Metamodels

The distinction between *Models* and *Metamodels* is central to Model Driven Engineering (MDE) [2], and in particular in the context of the MDA standard [26][37][39].

"*A **model** is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.*" *[2]*
*“A **metamodel** is a model that defines the language for expressing a model.” MOF Standard, [40]*

Figure 5 Models vs. Metamodels

The concept of model and metamodels are thoroughly described in the series “From Ancient Egypt to Model Driven Engineering” [18]. This series presents incrementally a “MDE megamodel” describing the essential relations between MDE concepts. A basic theory for MDE based on the set and language theories is also defined there. A *very* simplified view of this megamodel is reproduced below in the form of a UML class diagram

(other versions are under construction in Prolog, Z, and hieroglyphs). A model "*conforms to*" a metamodel [18].

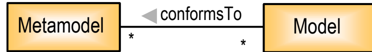

Figure 6 Simplified view of the "MDE megamodel"

Though the MDA standard from the OMG promotes the systematic use of a particular set of technologies such as MOF [40], XMI, CWM or SPEM [40], the MDE approach is much more general [35]. This approach supports various "technological spaces" (TS) [28]. The conformsTo relation, noted χ in greek or in hieroglyphs, takes different forms in the various TSs [12][18][28]. The following table shows a correspondence across terminologies.

TS / terminology	Model level (M1)	Metamodel level (M2)
MDE	"model"	"metamodel"
Languages	"program"	"grammar"
DBMS	"data"	"schema"
Architecture	"view"	"viewpoint"
Object	"instance"	"class"
Reverse Engin.	"software facts"	"schema"
XML	"document"	"DTD, schema"

Table 1: The ConformsTo relation

As an illustration, Figure 7 shows an example of a software model (on the left) that conforms to the metamodel (on the right). The model represents a particular piece of an hypothetical banking application. It is at the M1 level in the metamodeling pyramid [18][40]. The metamodel describes the concepts involved in the C programming language. It is at the M2 level. The reader is invited to compare the content of this figure with the above definitions (Figure 5).

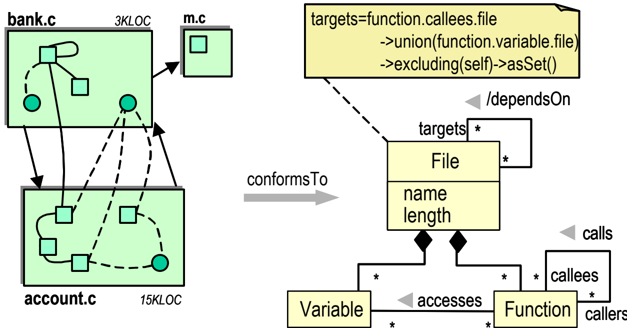

Figure 7 A model conforming to a metamodel

The size of the example, the syntax and the technology used to express the model and the metamodel are irrelevant details in the context of this paper. The focus is on the essential concepts and their articulations, not on a particular technology. The metamodel represented here conforms itself to UML and OCL. But it could also takes the form of a grammar, a DTD, a database schema, an XML schema, etc. [12]. Each TS presents its own advantages. What is important is (1) the availability of bridges between TSs [28] and (2) the correspondence between the megamodel and the particular TSs [18]. Discussing these relationships is beyond the scope of this paper, and anyway this does not change anything from a conceptual point of view.

3.2. SA: Views vs. Viewpoints

The distinction Model/Metamodel is the basis of the MDA Standard; the distinction *View/Viewpoints* comes from the IEEE 1471 Standard [23].

*"A **viewpoint** on a system is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system." [39]*
*"... A **view** of a system is a representation of that system from the perspective of a chosen viewpoint." [39]*

Figure 8 View vs. Viewpoints

The SAStd makes explicit the relation between *stakeholders*, their *concerns* and architectural viewpoints. The core of this standard is a conceptual framework represented in the form of a class diagram [23]. A subset of this standardized "megamodel" is reproduced below.

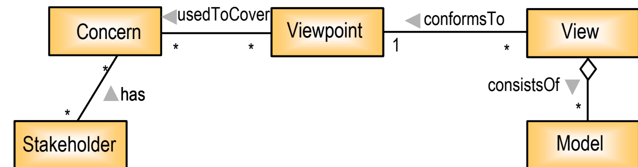

Figure 9 Simplified view of the "SA megamodel" [23]

Views and viewpoints must not be confused. Viewpoints are at the metamodel level (M2). They are independent from application domains and concentrate instead on software engineering concepts. By contrast, views are at the model level (M1) and describe particular software applications. For instance "Module" or "Component" are concepts at the viewpoint level (M2), while "Account" and "Withdraw" are concepts at the view level (M1).

Since viewpoints are reusable architectural bodies of knowledge, it makes sense to build catalogue of viewpoints. This idea is far from new. The 4+1 view(*point!*) is about 10 years old now [29]. Another set of viewpoints is given in [19]. In the context of DS, we identified yet another set [42][43]. In [46], Smolander and his colleagues list various other catalogues. According to these authors "*the viewpoints cannot be standardized but they must be selected or defined according to the environment and the situation*" [46]. In fact, this issue corresponds to the View Set Scenario mentioned above.

This naturally raises the question of languages to define viewpoints. The SAStd does not provide any kind of solutions. Symphony leaves this question open [3].

3.3. Integrating MDA and SA

The MDA and SA standards comes from two distinct organizations (OMG and IEEE), but fortunately these

standards are not incompatible. They share at least the concept of model as shown on the right of Figure 6 and of Figure 9). The OMG even made an effort to make the last version of the MDA Guide [39] compatible with the SA Std.

However something is missing. According to the SAStd a *View consistsOf* a collection of *Models* (Figure 9). This standard however does not make any reference to the concept of *Metamodels*. So we propose to merge the MDE and SA megamodels as following (Figure 10).

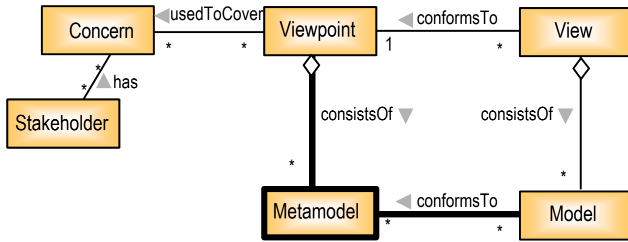

Figure 10 Merging MDE and SA megamodels

The key proposal in this paper is *the systematic use of Metamodels to represent Viewpoints* (see the association in centre of the previous figure). This approach is indeed very powerful because it merges the SA and MDE worlds.

In fact, the idea to use metamodels to describe view point is not really new. Just like others authors (e.g. [19][20]), we used metamodels to describe viewpoints [43][8]. But conversely to [19] and [20] which used contemplative metamodels, we used precise and productive metamodels with OCL constraints. Moreover this was in the context of software architecture reconstruction and the metamodels were used to build or drive concrete reverse engineering tools [8][10][11][6].

4. Integrating Reverse Engineering concerns

The previous section shows the relation between viewpoints and metamodels, but from where these elements come from? How these elements are connected to concrete stuff and existing assets in the company? Do large software companies define viewpoints and architectural metamodels in the first place? In practice, large software companies discover what is software architecture on the run [43]. Year after year successful companies acquire significant know-how in software architecture and develop much "metaware" to support this know-how [12][43]. The remainder of this paper shows that software architecture reconstruction can be based on reverse engineering existing metaware.

4.1. Appliware vs. Metaware

The distinction *Appliware/Metaware* has been introduced to model the evolution of very large software [12]. Simply put, appliware is software at the model level (M1), while metaware is software at the meta level (M2).

Appliware *is the set of software applications developed by a given company.* ***Metaware*** *is software to develop appliware and to control its evolution. In short Software = Appliware + Metaware*

Figure 11 Software = Appliware + Metaware

The term "modelware" is sometimes used in the MDE community to conveys the idea that appliware has to be developed using clean and high-level modelling language such as UML models. Reality is quite different. This paper is about industrial concerns, not about idealistic intellectual constructions. Industry may remain code-centric for a long time. Anyway large industrial software products are made of raw software items using legacy technologies. *Models are the neat conceptual parts of appliware, while metamodels are the neat conceptual part of metaware*. But metaware and appliware can be represented also in concrete software artefacts such as binaries or programs.

In practice, our experience shows that models and metamodels are everywhere. They are just deeply buried into legacy software items. That's why reverse engineering is important. As an illustration, let's come back to Figure 7. The architectural model depicted on the left could have been extracted from some application programs. This is *appliware reverse engineering* (M1). By contrast, the architectural meta-model represented on the right of Figure 7 could have been extracted either from a book about the C language, or from a metaware tool such as CIA. Both artifacts are at the M2 level [12][18]. When the CIA metaware tool was built in the 80's, the metamodel had not been made explicit, but it makes no doubt that the information contained in the metamodel of Figure 7 was already there. It had been buried both into the code of CIA and into the schema of the corresponding repository. Recovering such an architectural metamodel is an example of *metaware reverse engineering* (M2) [12][18].

This notion might seem quite odd. But there is nothing new there. This is just explicit modelling of existing practices. For instance Lammel and his colleagues have worked for long on *grammar reverse engineering* [30], that is recovering grammars from low level grammar-dependent software items. Compilers, pretty printers, interpreters are collectively called *grammarware in* [27]. Grammarware is to grammars what metaware is to metamodels. Metaware is just the generalization of grammarware.

	Appliware (M1)	Metaware (M2)
Programming in-the-Small	programs, binaries...	compilers, pretty printers, interpreters, grammarware tools, IDEs...
Programming in-the-Large	build files, log files, release information, product descriptions, application portfolios,...	component technologies, configuration manager, architectural tools, build tools, bug tracking systems, product managers, impact analysis tools...

Table 2: Metaware-in-the-small/metaware-in-the-large

As shown in the table above, beyond compilers, pretty printers and other *metaware-in-the-small items*, large companies had developed over the last decades large amount of *metaware-in-the-large items*. This list on metaware items depends on the software company and on the underlying software processes. Though metaware is usually developed in the dark in the company, this piece of software is extremely valuable. We had the chance to discover metaware in DS, because this company was so big, that the metaware was explicit. It was developed by an explicit Tool Support Team with no connection with appliware development [7]. But metaware also exist in smaller companies, though it might be more difficult to see. Metaware codifies the know-how of the company in an operational way, in particular the SA know-how. For instance, AT&T developed internally a myriad of metaware tools such Make or CIA, and this to handle the appliware produced by this company. DS just did the same [7].

4.2. Metaware in context

One might wonder how metaware relates to the MDA and SA standards. The answer is given by the following megamodel which extends the MDA and SA standards with the notion of *MetaUsecase*, *MetawareItem* and *AppliwareItem* (see the bottom part of the figure in bold).

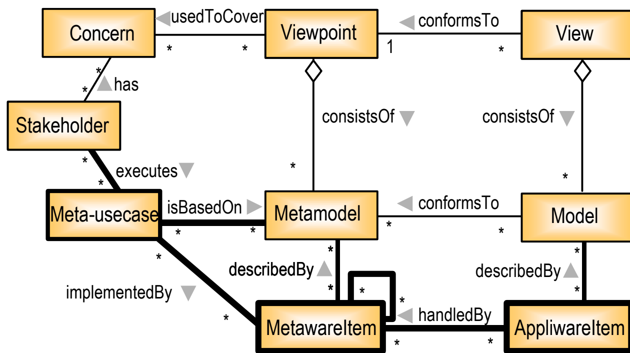

Figure 12 A megamodel for architecture recovery

These concepts and relations are studied in the remainder of this paper, but to illustrate each concept let's consider again Figure 7. The source code and the executable of the banking system are *AppliwareItems* (M1) that are *describedBy* the *Model* (M1) depicted on the left of Figure 7 . This source code *AppliwareItem* (M1) is *handledBy* tools such the C compiler or CIA, which are themselves *MetawareItems* (M2). The CIA tool is partially *describedBy* the *Metamodel* (M2) depicted on the right of Figure 7. Let's assume that a *Stakeholder* (M2) such as a release manager want to eliminate deadcode from the banking application. The stakeholder could *execute* the "Identify Dead Code" *MetaUsecase* (M2) *implementedBy* the CIA *MetawareItem* (M2). If CIA was not supporting this use case, then it would be wise to modify it, leading to *metaware reengineering*.

4.3. Metaware reengineering vs. forward engineering

The SAStd concentrates on defining the various stakeholder' *Concerns* and associated viewpoints (top of Figure 12). One can then imagine a forward engineering process leading to an full architectural environment made of *MetawareItems* (bottom of Figure 12). This would be *metaware forward engineering*.

Unfortunately this view is quite naive, and in our opinion unrealistic. The notion of software architecture is a moving target [12], and the defining the relevant architectural concepts is an incremental process [3]. Moreover, existing (and sometimes legacy) metaware items already in the company must not be ignored. While Symphony recognizes the incremental nature of architecture reconstruction processes, it starts from scratch whereas Ca*c*Oph*o*Ny put the emphasis of reusing existing metaware. Developing a architectural environment for a large company, is best viewed as a *continuous metaware reengineering process*.

5. Overview of the Ca*c*Oph*o*Ny generic process

In this paper we extend the View Set Scenario with the problem of building a metaware environment. The overall structure of the Ca*c*Oph*o*Ny process is depicted in the figure below. They are six major steps, though the process is by nature iterative and incremental.

	Step	Sub activities
I	**Metaware domain and asset analysis** (section 5.1)	Metaware inventory Metamodels recovery Metamodels integration Metamodel clustering Metamodel packaging
II	**Metaware requirement analysis** (section 5.2)	Meta-level actors identification Meta-level use cases identification Metaware assessment Metaware improvement analysis Meta-level use cases description
III	**Metaware specification** (section 5.3)	Meta-model filtering and extension Presentation specification Metaware specification packaging
IV	**Metaware implementation**	Extractors development and reuse Viewers development and reuse Extractors and Viewers integration
V	**Metaware execution**	Deployment Execution Monitoring
VI	**Metaware evolution**	Evaluation Feedback Change analysis

Table 3: The Ca*c*Oph*o*Ny metamodel-driven process

Going into details is impossible due to a lack of space, so we concentrate on giving an idea of the whole process. Step I to III are also be shortly described below. The reader is invited to refer to Symphony [3] to get a more detail view of the reconstruction process and to see which techniques can be used at each step. Symphony and Ca*c*Oph*o*Ny are quite similar in their processes. The focus in this paper is on the use of metamodels to drive the reconstruction process.

5.1. Metaware domain and asset analysis (I)

Let us assume that you've just arrived in a given company. You don't know anything about the application domains addressed by the company. Don't worry, you should not really care about it. Just like in regular forward engineering processes, *domain analysis* is the first step. But we refer here to the Software Architecture domain (M2), not to application domains (M1). *Asset analysis* is fundamental because the company has its own SA know-how (M2).

5.1.1. Metaware inventory (I.a). Information about architectural concepts can be extracted by means of interviews [46]. However, this could lead to fuzzy and even contradictory information, due to a lack of a well established architectural ontology in the company. Anyway they are also plenty of other sources of information [8][7], including for instance slides about how to develop components, quality insurance documents, quality tools, software repositories, bug tracking systems, application portfolios, etc. As explained in as explained in [8] and [10], studying these artefact enable to establish a first inventory of the SA concepts used in the company. The output of this step is a raw mapping between (1) these concepts and (2) the metaware artefacts used to deal with them (Figure 13).

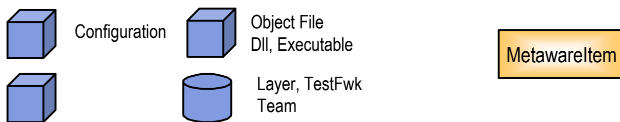

Figure 13 Inventory of metaware items

5.1.2. Metamodel recovery (I.b). In a very large company such as DS, they are plenty of metaware items. After 7 years collaboration we were only aware of some parts of the metaware. The problem is therefore to select from the inventory the most promising items in order to analyse them first. The objective of this analysis is to provide a conceptual meta-model [12] for each selected metaware item. For instance, you might find that in the bug report database there is an association that maps each bug found with the "test framework" that enabled to find the bug. Some database fields might indicate that "test frameworks" are decomposed into so called "test modules", and so on. With this kind of information a meta-model can be build incrementally to describe each metaware item (Figure 14).

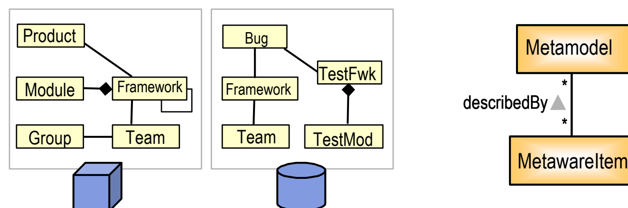

Figure 14 Recovering metamodels of metaware items

5.1.3. Metamodels integration (I.c) The meta-models recovered in the previous phase must then be integrated in order to produce a global metamodel. In practice, this is not so easy. Schemas and metamodel integration (e.g. [12]) is an active research field. Metaware items were probably produced incrementally by different teams. So there are good chance that the tools are not on slightly divergent SA concepts. The integrated meta-model should provide a solution for the conceptual integration, typically thanks to conceptual bridges. Implementation issues should be recorded for further analysis (step IV). The output of this step is a global integrated meta-model. At this point is "just" a conceptual metamodel [12] (Figure 14).

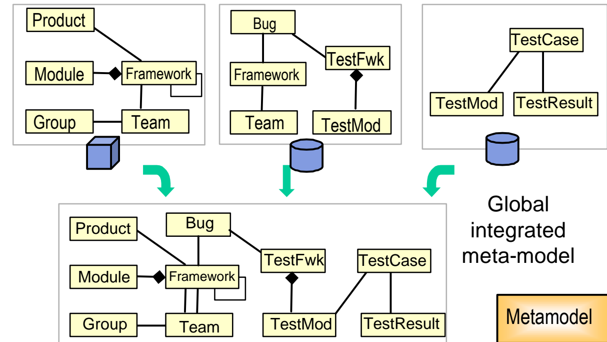

Figure 15 Metamodels integration

5.1.4. Metamodel clustering (I.d) In large software companies, the development of metaware items is usually driven by coincidental facts [7][12]. One of the goal of meta-model integration is to break the somewhat arbitrary boundaries of individual meta-models. This could lead to a large flat metamodel with tens or hundreds of elements, or even more depending at which level of abstractions the analysis was done. To cope with this problem the global meta-model should be structured by identifying cohesive sub-domains. We did that in the context of DS [43][44], but at the time of writing this paper we don't have clear-cut rules for that. Anyway some of the identified sub-domains could be candidates to form reusable SA viewpoints.

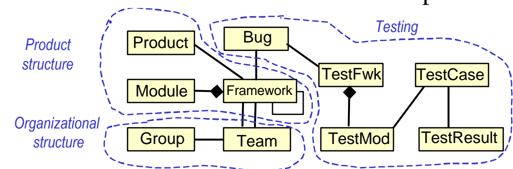

Figure 16 Metamodel clusters

5.2. Metaware requirement analysis (II)

What is missing? People. The problem to be addressed here is to define *actors* (i.e. *who* is using the architectural environment) and *usecases* (i.e. *what are they doing* with that metaware). Note that we refer here to the meta-level actors and meta-level usecases (M2) (see below). Interestingly, we found that traditional M1-level techniques

could be applied as well the M2 level. For instance, the Unified Process [24] suggests three steps for requirements analysis: (1) actor identification, (2) usecase identification and (3) usecase detailed description. These steps are applied below at the metamodel level, on metaware (M2) rather than appliware (M1).

5.2.1. Meta-level actor identification (II.1) Simply put, meta-level actors are those who are referred as *stakeholders* in the IEEE 1471 standard [23], *ProcessRole* in the OMG' SPEM standard [38], and *worker* in the Unified Process [24]. Whatever the name used, identifying meta-level actors is not easy. But this step is now well covered in general [38], and, in particular in the SA literature. The reader can refer to [45] for a discussion issues related to SA. See [44] for some examples of meta-level actors identified within DS.

Figure 17 Meta-level actors (stakeholders)

Note that the meta-level actors should not be confused with the model-level actors, i.e. the users of appliware. *Client* and *Teller* are M1-level actors, *Architect* and *TestManager* are M2-level actors. The first ones use appliware, the second ones use metaware (to produce appliware). See [12] for a more complete discussion on this.

5.2.2. Meta-level use case identification (II.2) Following the UML standard and the Unified Process, the identification of actors naturally leads to identify use cases. Providing short yet meaningful names for use case is key [38]. These terms should be part of the metaware culture of the company. The following figure provides some examples of meta-usecases (M2). By contrast, *WithdrawMoney* and *TransferMoney* are at the M1 level.

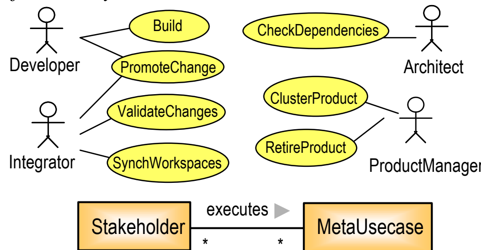

Figure 18 Meta-usecase diagram

5.2.3. Metaware assessment and improvement analysis (II.3) Actors and use cases must be gathered in one of more annotated uses cases diagrams. Meetings should be organized in the company to determine where the focus of attention should be put and what is missing. For instance, during a meeting you might learn that the *RetireProduct* is indeed one of the key issue in the company and that people from the business department do not have enough visibility on product dependencies. After further explanation, one or various use cases might be added to the use case diagram. Our experience suggests these diagrams greatly help in setting priority and focusing on essential parts, but more research is required on this topic. Note that, though we are not aware of any standard using the systematic use of meta-level actors along with metamodels definitions, some standards from the OMG seem to go in that direction (e.g. the standard for deployment include a use case diagram).

5.3. Metaware specification (III)

After the validation of requirements, the next step is naturally the specification of the metaware to be build.

5.3.1. Meta-model filtering and extension (III.1) The first step is to perform *meta-model filtering*, that is to identify with great precision which subset of the global meta-model is required to execute a given meta-usecase.

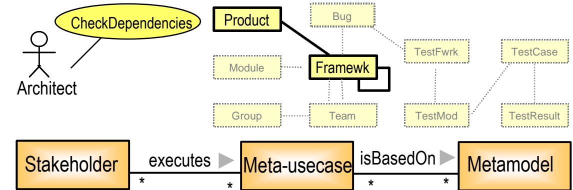

Figure 19 Filtering metamodel with meta-usecase

When information is missing to realize a usecase, the metamodel have to be extended. The specification of derived elements (prefixed by / in UML) can be done in OCL (Figure 7). This technique has been used successfully in the context of DS. See [8] for an example of architectural view defined with OCL. and [10][11] for an example of tool directly interpreting by such a kind of formalism. Note that Symphony uses the terms *source viewpoints* and *target viewpoints*, but does not provide particular techniques to express the mapping between these viewpoints. Expressing such mappings and making them executable is an active research area in MDE. The QVT standard is for that.

5.3.2. Presentation specification (III.2) Metamodels can also be used to specify the visual appearance of metaware tools [13]. For instance the next figure uses UML and OCL to specify the visualization technique used to display the architectural model shown on the left of Figure 7. A simple visualization metamodel is on the right of Figure 20. This metamodel is in fact a simplified version of the visualization metamodel that can be extracted from the Dot tool from AT&T. This technique of mapping is directly in line with the MDE approach [10].

5.4. Last steps of Ca*c*Oph*o*Ny (IV, V, VI)

The last steps of the Ca*c*Oph*o*Ny process are metaware implementation, metaware evaluation and metaware

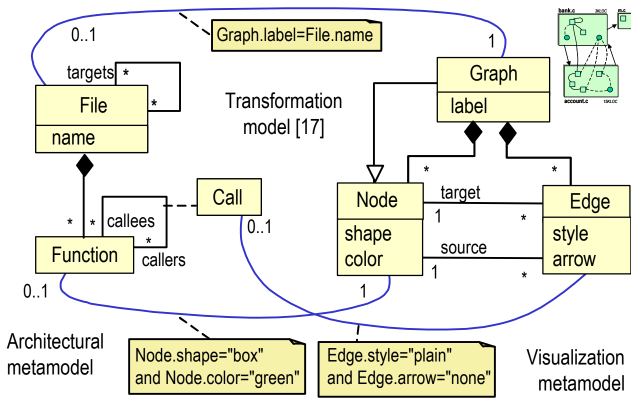

Figure 20 Metamodel-driven visualization

evolution. They are obviously determinant for the success of the whole approach since they deliver concrete metaware; but as said before this paper concentrates on the View Set Scenario. The lack of space makes it impossible to cover the implementation and evolution issues.

However, it should be clear that an implementation can be derived from the specification produced in the previous section. Metaware is just software, and there are plenty of techniques and methods to implement software. In the context of Dassault Systèmes, an architectural environment was successfully implemented [8][42].

Obviously, not everything can be automated, but still, the specification is then a description of the output to get. In fact, even if the implementation is done in an ad-hoc way, the requirements analysis and the specification phase still provide a good help. A more complete description of the last steps can be found in the Symphony paper [3].

One can build either metamodel-specific tools (e.g. OMVT [6][42]), or generic tools. This includes GuPRO [5], Rigi [34], or PBS [21]. We attempted to use Rigi in the context of DS. But we found its visualization technique too limited, and its support to metamodels (called domains) too weak. This therefore developed the Generic *S*oftware *E*xploration *E*nvironment (G^{SEE}). G^{SEE} is a metamodel-driven prototype that directly interprets a metamodel specification [10][11]. G^{SEE} connects to existing metaware items, explores their metamodels, and then models are extracted on demand. Thanks to G^{SEE} we have shown that it feasible in some circumstances to support the co-exploration of the models and meta-models. In this case, the meta-model is recovered on the fly [10][13].

6. Conclusion

So what is software architecture? This question leads to cacophony in academy, cacophony in industry, and we still don’t know what is software architecture. But now we don’t care; because we know that when we will see one, we will be able to describe it very rigorously using metamodels.

The process presented in this paper has been successfully applied in the context of Dassault Systèmes, though the process was not identified as such in previous publications [6][7][8][42][43].

The systematic use of metamodels makes the Ca*c*Oph*o*Ny process very generic, yet precise. It does not depend on a particular technology, company or SA definition. In fact, Symphony and Ca*c*Oph*o*Ny are complementary approaches, and we bet that the benefit of using metamodels will be recognized in the near future.

A major contribution of this paper has been to merge Model Driven Engineering with the IEEE 1471 standard and the notion of metaware reverse engineering.

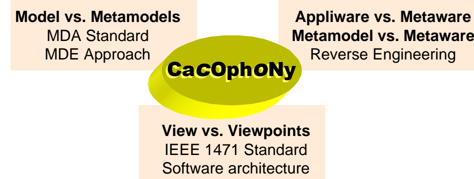

Figure 21 Ca*c*Oph*o*Ny: an integrative approach

This fusion is important because, after a boom on ADLs in the 90’s, advances in the software architecture have been quite limited. On the contrary, MDE is a very active field. MDE has the potential to bring not only new perspectives on software architecture but also a uniform approach along with new or existing technologies. Most approaches and techniques can be recast to fit in the MDE framework.

Beyond the merge of the MDA and the IEEE standard, various concepts have been identified in this paper, though *none* of these concepts have *not* been invented. They just model the reality we discovered in software industry [12]. For instance we discovered the importance of making the distinction between Metaware and Appliware during our collaboration with DS. In 7 years, we saw a lot of metaware items, but we never saw a single line of appliware [7].

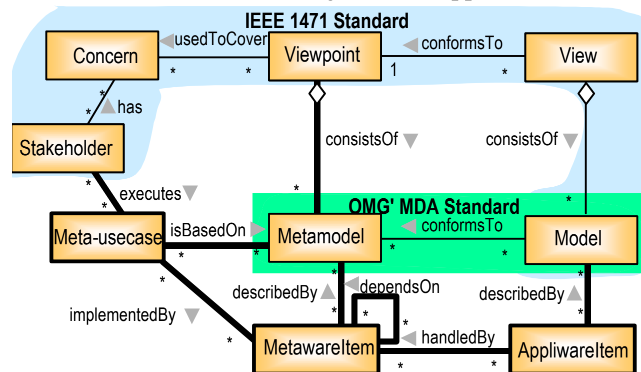

Figure 22 Contributions of this paper

Obviously further research is required on metaware (re)engineering. In this paper we just scratched the surface, but meta-model driven reverse engineering seems to be a very promising approach for the future. After all, what is specific to software architecture in this paper?

7. References

[1] J. Bézivin, N. Ploquin, “Tooling the MDA framework: a new software maintenance and evolution scheme proposal”, Journal of Object Oriented Programming, 2001

[2] J. Bézivin, O. Gerbé, "Towards a Precise Definition of the OMG/MDA Framework", ASE 2001

[3] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, C. Riva, “Symphony: View-Driven Software Architecture Reconstruction”, WICSA’ 2004

[4] S. Demeyer, S. Ducasse, O. Nierstrasz, “Object-Oriented Reengineering Patterns”, Morgan Kaufman Publishers, 2003

[5] J. Ebert et al, GUPRO, http://www.gupro.de

[6] J. Estublier, J.M. Favre, R. Sanlaville, "An Industrial Experience with Dassault Systèmes' Component Model", Book chapter in Building Reliable Component-Based Systems, Archtech House publishers, 2002

[7] J.M. Favre, J. Estublier, R. Sanlaville, “Tool Adoption Issues in Very Large Software Company”, ACSE 2003

[8] J.M. Favre and al., "Reverse Engineering a Large Component-based Software Product", CSMR'2001

[9] J.M. Favre, R. Sanlaville, “Continuous Discovery of Software Architecture in a Large Evolving Company”, Workshop on Software Architecture Reconstruction, WCRE’2002, slides available at www.imag.fr/~jmfavre

[10] J.M. Favre, "A New Approach to Software Exploration: Back-packing with G*SEE*", CSMR'2002

[11] J.M. Favre, "G*SEE*: a Generic Software Exploration Environment", IWPC'2001

[12] J.M. Favre, “Meta-Model and Model Co-evolution within the 3D Software Space”, ELISA’2003

[13] J.M. Favre, “Meta-Model and Model Co-exploration with G*SEE*”, VISSOFT’2003

[14] J.M. Favre, “Meta-Model Engineering for Architecture Reconstruction”, Presentation at the Dagsthul Seminar on Software Architecture Recovery and Modelling, slides available at www.imag.fr/~jmfavre, March 2003

[15] J.M. Favre, “Meta-Model (Driven) Reverse Engineering”, Presentation at the Dagsthul Seminar on Language Engineering for Model Driven Development, slides available at www.imag.fr/~jmfavre, March 2004

[16] M. Fischer, M. Pinzger, H. Gall, “Populating a release history database from version control and bug tracking systems”, ICSM’ 2003

[17] M. Fowler, “Who Needs an Architect?”, IEEE Software, July 2003

[18] “From Ancient Egypt to Model Driven Engineering”, series available at www-adele.imag.fr/mda

[19] C. Hofmeister, R. Nord and D. Soni. Applied Software Architecture. Addison-Wesley Publisher, 2000.

[20] R. Hillard, “Viewpoint Modelling”, 1st ICSE Workshop on Describing Software Architecture with UML, 2001

[21] R. Holt et al, PBS, http:// www.turing.toronto.edu

[22] R. Holt, “Software Architecture as a Shared Mental Model”, IWPC’ 2003

[23] IEEE. “IEEE Recommended Practice for Architectural Description of Software-Intensive Systems”. IEEE Std 1471-2000, www.pithecanthropus.com/~awg/public_html/

[24] I. Jacobson, G. Booch, J. Rumbaugh, “The Unified Software Development Process“, Addison Wesley, 1999

[25] R. Koschke, “Atomic Architectural Component Recovery for Program Understanding and Evolution“, PhD, University of Stuttgart, 1999

[26] A. Kleppe, S. Warmer, W. Bast, "MDA Explained. The Model Driven Architecture: Practice and Promise", Addison-Wesley, April 2003

[27] P. Klint, R. Lämmel, C. Verhoef, “Towards an engineering discipline for Grammarware”, www.cs.vu.nl/grammarware

[28] I. Kurtev, J. Bézivin, M. Aksit, "Technological Spaces: an Initial Appraisal", CoopIS, DOA'2002, Industrial track, 2002

[29] P. B. Kruchten. The 4+1 view model of architecture. IEEE Software, 12(6): 42--50, November 1995.

[30] R. Lämmel, C. Verhoef, “Semi-automatic grammar recovery”, Software Practice and Experience, 2001

[31] A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, M. Maroti, “On Metamodel composition”

[32] J. Michaud, M.A. Storey, H. Muller, “Integrating Information Sources for Visualizing Java Programs”, ICSM’2001

[33] S.J.Mellor, K.Scott, A.Uhl, D.Weise, "MDA Distilled: Principles of Model-Driven Architecture", Addison Wesley, March 2004

[34] H.A. Muller et al, RIGI, http://www.rigi.csc.uvic.ca/

[35] CNRS, Model Driven Engineering, CNRS Action Spécifique Project, http://www-adele.imag.fr/mda/as

[36] L. O’Brien, C. Stoermer, C. Verhoef, “Software Architecture Reconstruction: Practice Needs and Current Approaches”, SEI Technical Report CMU/SEI-2002-TR-024, 2002

[37] OMG, “MDA: the OMG Model Driven Architecture”, www.omg.org/mda/

[38] OMG, “Software Process Engineering Metamodel Specification”, Version 1.0, formal/02-11-14, Nov. 2002

[39] OMG, “MDA Guide Version 1.0.1”, omg/2003-06-01, 2003

[40] OMG, "Meta Object Facility (MOF) Specification" Version 1.4, April 2002

[41] C. Riva, “Architecture Reconstruction in Practice”, WICSA’ 2002

[42] R. Sanlaville, Y. Ledru, J. Estublier, J.M. Favre, “An Architectural Environment for the Evolution of Complex Software”, submitted to Software Practice & Experience

[43] R. Sanlaville, “Software Architecture: An Industrial Case Study within Dassault Systèmes”, PhD dissertation in french, University of Grenoble, 2002

[44] R. Sanlaville, J.M.Favre, Y. Ledru, "Helping Various Stakeholders to Understand a Very Large Software Product" ECBSE, 2001

[45] K. Smolander, T. Päivärinta, “Describing and Communicating Software Architecture in Practice: Observations on Stakeholders and Rationale”, CAiSE 2002

[46] K. Smolander and *al.*, “Required and Optional Viewpoints What Is Included in Software Architecture?”, TBRC Lappeenranta, ISBN 951-764-575-9, 2001