Reverse Engineering a Large Component-based Software Product

Jean-Marie Favre*, Frédéric Duclos+, Jacky Estublier*, Remy Sanlaville+, Jean-Jacques Auffret+

*Laboratoire LSR-IMAG
220 , Rue de la chimie
Domaine Universitaire, BP53X
38041, Grenoble Cedex 9, France

+Dassault Systèmes
9, quai Marcel Dassault
92150 Suresnes
France

Abstract

Most research done to date on software maintenance has been focused mainly on the evolution of legacy systems based on obsolete technologies. However, the use of more recent yet evolving technologies, like component-based techniques, also raise various issues about software comprehension and evolution. In particular, current industrial-strength component models like COM are based on many technical aspects that make them difficult to understand and use. The evolution of large component-based software products is thus an emerging issue. This paper takes as a case study the component model developed and used by Dassault Systèmes, one of the largest software companies in Europe, for the development of its product lines, namely CATIA, DELMIA, and ENOVIA. This paper shows how the use of a meta model can help in understanding and reasoning about components, and how this meta model constitutes a good basis for building a reverse engineering environment. Currently, two kinds of tools have been integrated in this environment: OMVT which is Dassault Systèmes specific, and GSEE which is a generic tool independent from the meta-model used.

1. Introduction

Large software products have always been difficult to understand and evolve [10]. In the late 80's this has leaded to the emergence of closely related techniques like reengineering, reverse-engineering and restructuring, collectively called RE3 technologies [26].

Traditionally, most research work in RE3 focused on the evolution of legacy software products based on obsolete technology. Many tools have been proposed to deal with old-fashioned programming languages such as Cobol, Fortran or C for instance. There is still a belief that the usage of RE3 techniques are restricted to legacy systems.

However, reverse engineering is defined as "*the process of analyzing a subject system to: (1) identify the system's components and their interrelationships and (2) create representations of the system in another form or at a higher level of abstraction*" [6]. This definition makes no mention about the level of maturity of the technology involved, nor the definitions of restructuring and reengineering do [1,6].

RE3 techniques have to follow the evolution of industrial software engineering practice. The wave model [21] is very valuable in this context, since it provides different historical views on software engineering evolution. For instance, Figure 1 shows the stream of interest in software structuring paradigms. One way to read this figure is to notice that there have been a series of shifts from ad-hoc programming, to object-oriented (OO) programming, the latter being the most popular paradigm used today in industry.

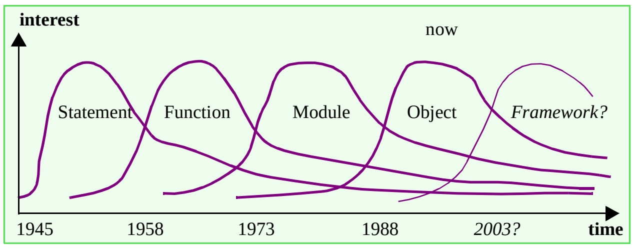


Figure 1. The stream of interest in structuring paradigms [21]

These waves of interest in forward engineering technology have an strong impact on RE3 evolution. For instance, in the mid 80's, the first restructuring tools focused on the shift from ad-hoc programming to structured programming by removing goto statements from unstructured programs [2]. The shift from structured programming to modular programming also led to clustering and (re)modularization tools, tools to recover software architecture, etc. [1,3,15,16,19,27].

The increasing interest in object-oriented technology in the last decade, results today in the existence of large OO software products. However, every technology will show its limits when applied at large [4]. In particular, large software companies, pioneers in the OO-at-large approach, understood that the OO paradigm is no silver bullet [7].

The existence of large OO software products naturally give rise to significant research effort focusing on the intersection of OO and RE3 (e.g. the Spool [36] and Famoos projects [7]). Note that this last step in the evolution of RE3 discipline marks a discontinuity: RE3 techniques are no longer restricted to the maintenance of legacy systems, they can be applied for the evolution of state-of-the-art software products. Many researchers now believe that RE3 techniques must be smoothly integrated within the forward engineering process, leading for instance to the concept of round-trip engineering (a series of short forward and reverse engineering cycles). As Fowler pointed out [13], experienced OO programmers know that an object-oriented framework can not be right the first time - it must evolve as experience is gained with its use. So, software refactoring [13] (the term used in the OO world for restructuring), should therefore be seen as a continuous restructuring effort integrated in the development and evolution of any OO software product.

Then, what will be the next step? We believe that component-based (CB) software development may be the one, at least this is what the experience reported in this paper suggests. Nowadays, there is a widely accepted belief that large software products should be built as the assembly of *software components*. Tough promising, this idea was not put into practice at a large scale until the emergence, in the last few years, of industrial-strength component models like Microsoft' COM [5,29], OMG' Corba and CCM [30,33], Sun' JavaBean [9,32] or Sun' Enterprise Java Bean [34]. The availability of such powerful and innovative CB techniques may constitute the basis of the next significant wave of interest in industry.

Dassault Systèmes (DS), the world leader in CAD/CAM, is a pioneer in this domain. This large software company has developed a proprietary component model which has been successfully used for years in the development of CATIA [28]. However, like other companies such as MicroSoft or Sun, DS faces difficulties in teaching his component model. Understanding large CB software is not an easy task. The existence of these issues should not be surprising since the CB approach is still in its infancy and is usually not formalized.

This paper results from the collaboration between an academic institution, the LSR laboratory, and one of the largest software company in europe Dassault Systèmes (DS), in an attempt to deal with problems related to the evolution of large component-based software products. In particular, this paper shows how a reverse engineering approach can substantially improve the understanding of a CB software product, taking CATIA as a case study.

The rest of this paper is organized as follows. Section 2 briefly presents the main features of the DS component model in an informal way. Section 3 describes how a meta model can be used to formalize the concept of component. Section 4 shows how this meta model can be converted into useful reverse engineering tools. Sections 5 provides a discussion and briefly presents related work. Section 6 concludes the paper.

2. The DS Component Model

In the mid 90s, when DS initiated the development of CATIA V5 [28], it was rapidly discovered that OO technology has serious limitations and in particular that C++ did not satisfy all of the requirements. The two most important aspects were the following:

• **Concurrent engineering**. C++ entities are too closely related: even a minor change may produce a dramatic number of recompilations. For large products and high concurrent engineering constraints, this is a major issue.

• **Extension capabilities**. The CATIA major customers and development partners have a need to be able to extend DS components with their own code, even without the availability of the source code [8].

To solve these (and other) issues, DS developed, on top of C++, a component model borrowing ideas from COM, Corba and Java. Here follows a very short and informal description of the "Object Modeler" (OM). Despite its name the OM is best viewed as a component model. This section first presents the main OM concepts, and then provides some information about its realization.

2.1. Conceptual level

OM *components* are pieces of code that can be manipulated through the use of *interfaces*. Interfaces can be seen as abstract proxies for real objects that receive client requests and forward them to the component implementing the interface. The interface concept helps in addressing the concurrent engineering issue, since it isolates interface clients from modification of the component implementation.

To be more precise, a component is made of set of elementary pieces of code, called *implementations* (an implementation is realized by a C++ class). One of these implementations is called the *base* (of the component). Other implementations, called *extensions*, can be attached later to the base in order to extend the component. A fundamental feature is that extensions refer to the base, but the base ignores that it is being extended. This allows a new extension to be added at a later time, without any need to recompile the base nor any of the other extensions.

The OM also provides several other mechanisms not described in this paper. For instance it supports the concept of delegation or conditional implementation.

2.2. Realization level

All concepts provided by the OM, are implemented in terms of C++ entities. For instance, interfaces and implementations are both represented by C++ classes. In fact, the realization level is much more complex since the mapping is not one to one: the realization of a single OM entity can produce many C++ entities. Moreover, for a given conceptual entity there are many realization choices: to improve performance and address other non-functional requirements, DS has designed and tested a wide range of realization techniques. All these techniques allow to build efficient components, but at the same time developing and maintaining these components is quite a complex task.

To keep the control on the resulting software, OM concepts are translated into C++ code using patterns and naming conventions. This approach is very similar to those taken by other component models (e.g. [32]). In the case of OM, additional information is also inserted into the source code through the use of macros. This alleviates the burden of writing repetitive pieces of code. Some pieces of code are also automatically generated.

Extra information is also provided in separate text files called *dictionaries*, containing tuples "component - interface - dll". These files permit, at run time, to locate and load only the necessary DLLs required during an execution and therefore to increase performances.

2.3. Related issues

The OM has been successfully used to build very large software products (hundreds of applications made of thousands of components, developed by hundreds of software engineers). Several issues have been raised:

• **Need for a conceptual view**. Software engineers describe components using low-level mechanisms at the realization level (naming conventions, macro, etc.). OM conceptual entities are mixed with huge amount of C++ code.

• **Need for a centralized description**. Information about a single OM entity is often spread among many different files, including source code and dictionaries.

• **Need of formalization**. The OM component model is informally defined by means of a huge documentation. While very valuable, this documentation is often imprecise and many realization constraints are poorly documented. Moreover, since the realization techniques tend to evolve over time to ensure continuous improvement, the most accurate information is available from experienced software engineers.

• **Need of specialized tools**. Software engineers develop and maintain components using traditional C++ tools. While sufficient to complete most of the tasks, those tools are inadequate for instance to understand the behavior of the software at the conceptual level. DS also developed different tools to cope with specific problems but they are limited in scope.

Indeed, the OM model, just like other component models (COM, CCM, etc.), is difficult to teach and to understand. Experienced software engineers learn how to build components, but they often find it difficult to know what went wrong when the software they have developed does not show the expected behavior.

What is missing is a clear picture of the overall component structure at a conceptual level. The realization level is available, but it contains too many technical details. Reverse engineering provides thus a logical approach to these problems, since its goal is to “*create representations of the system in another form or at a higher level of abstraction*” [6]. However, while most reverse engineering techniques deal with traditional and well-defined concepts, the problem here is to deal with the reverse engineering of component-based software systems, which is a rather new issue in the RE3 domain. Before trying to develop a reverse engineering tool, the first step is to give a rigorous definition of what a component is. This is what is done in the next section.

3. Building a meta model

Defining the meta model for the OM was the first step of our approach[1]. The key idea is to describe each concept of the OM model as an object-oriented item described using the UML notation [31]. The production of the meta model has been a long process since the model is quite complex and is still slightly evolving. Describing the full meta model is out of the scope of the paper; we rather emphasize the method and the main properties of this meta model.

One of the main interests of using a meta model is that it makes it possible to define different views on it. This paper concentrates on a small but central part of the meta model: how the components are built from bases and extensions. Here the OM is described only at the conceptual level, without giving any detail on the realization level. Furthermore, a few simplifications have been made to keep things simple.

1. In this paper the term meta-model is used to be consistent with the approach used to describe the UML language itself [31], to describe CCM with UML [33], etc. The term meta model is also used in the Famoos project [7] in a similar way.

3.1. Describing components as black boxes

While the OM model is quite sophisticated, from an external point of view, the OM is only based on two main concepts: components and interfaces. Clients of a component don't have to know how this component is built. This idea is described in the UML class diagram presented in Figure 2 on the top of the next page. Components and interfaces are linked together by a single association: a component can implement many interfaces (this is indicated through the * symbol near to the name of the role allInterfaces). Conversely, an interface may be implemented by any number of components.

3.2. Describing component items separately

As it was said before, actually components are made of elementary pieces of software produced separately by software engineers. The concrete representation of these items in terms of C++ entities or other low level mechanisms like macros is not relevant from a conceptual point of view. So, instead of giving the many technical details required to describe those items, Figure 3 introduces four abstract languages. The first three represent abstraction of information contained in the source code, while the last one is the abstraction of "dictionaries".

Each abstract language summarizes all the information required by the OM at the conceptual level. Note that, within the UML diagrams, arrows indicate unidirectional associations. For instance, an interface refers to its super interface but not to its sub-interfaces. Similarly, an extension refers to the bases it extends, but not the other way around. Cardinality information also brings useful precision. For instance, from the Figure 3 we can learn that both interfaces and bases support single inheritance (roles named super).

Thanks to the Object Constraint Language (OCL) [25] provided with UML, it is also possible to: (1) define *derived information*, (2) describe *additional constraints*. As we will see in the Section 4, this is very important in practice. Consider for instance, the following OCL expression.

```
1  context i : Interface
2    inv : i.allSuper = i.super.allSuper->including(i.super)
3    inv : i.allSuper->excludes(i)
```

Line 1 and 2 defines for each interface the *allSuper* role (not depicted in the figure), as being the set of all super interfaces for a given interface. This recursive definition provides an example of *derived information*. Line 3 uses this derived information to describe an *additional constraint*: the inheritance hierarchy between interfaces contains no cycle.

3.3. Linking component items together

Even if software engineers describe component items separately (that is required for concurrent engineering), one of the important aspects of the OM is how components are built from these items. Figure 4 shows a class diagram gathering the 4 languages described previously (these associations are drawn in black in the figure) and add derived information (in grey and prefixed by a "/" symbol).

Putting together component items must be done with an extreme care, not all combinations will work. Describing assembly constraint is therefore of fundamental importance. Indeed, this process leads to a great number of constraints that each assembly must satisfy to be considered as consistent. In the context of this paper, only two of these constraints will be described in Section 3.4 to illustrate the approach, but we first need to introduce the necessary derived information upon which the constraint are based. This is what is done in Figure 5.

The OCL expressions explain how components are made from implementations and define inheritance on components. Line 2 indicates that component inheritance (*super*) is in fact directly derived from base inheritance (*base.super*). Lines 3 indicate that the extensions of a component (*extensions*) are all extensions attached to its base. Line 4 defines the direct implementation of a component (*implementations*). Line 5 recursively defines the set of all implementations (*allImplementations*) of a component considering component inheritance. Line 6 defines the direct interfaces of a component. Finally line 7 defines the set of all interfaces (*allInterfaces*) that can be reached from the component following either the interface inheritance relationship or the component inheritance relationship.

3.4. Discovering potential inconsistencies

Gathering the four abstract languages (Figure 3) into a single diagram (Figure 4) helps to discover possible inconsistencies between the information they describe. Indeed, the global view provided by a meta model is one of the main benefits of the approach.

For instance, in our context, one should wonder what is the relationship between the derived role *declaredInterfaces* and the explicit role *allInterfaces*. After asking for more precision from OM designers, we learned that software engineers must explicitly declare all interfaces in the component language (i.e. in the dictionaries). So the next invariant is expected to hold.

```
1  -- context c : Component
2  --   inv : c.declaredInterfaces = c.allInterfaces
```

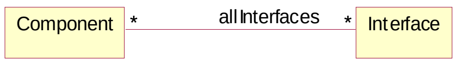


Figure 2. Describing components as black boxes (external point of view).

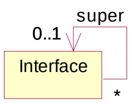
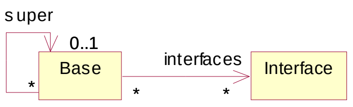
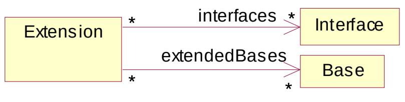
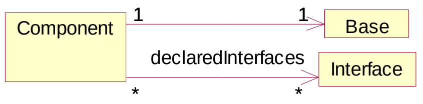
	class diagram	abstract syntax
(1) interface language		**interface** *<interfacename>* [**inherits** *<interfacename>*]
(2) base language		**base** *<basename>* [**inherits** *<basename>*] [**implements** <interfacename>*]
(3) extension language		**extension** *<extensionname>* [**extends** <basename>*] [**implements** <interfacename>*]
(4) component language		**component** *<basename>* [**implements** <interfacename>*]

Figure 3. Describing component items separately by means of four abstract languages

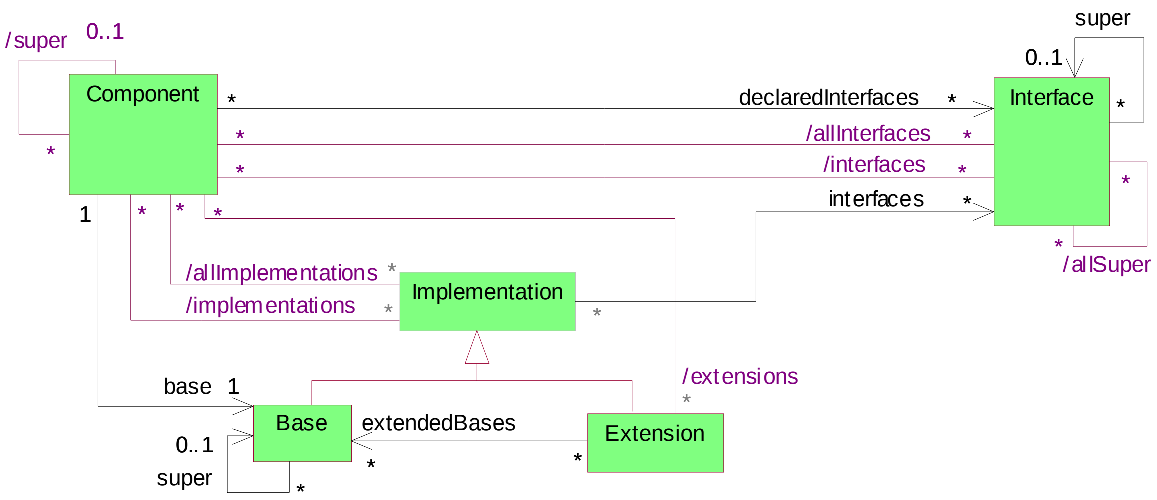


Figure 4. Linking component items together

```
1   context c : Component
2     inv : c.super = c.base.super.component
3     inv : c.extensions = Extension.allInstances ->select(e |  e.extendedBases->includes(c.base))
4     inv : c.implementations = c.extensions->including(c.base)
5     inv : c.allImplementations =c.implementations->union(c.super.allImplementation)
6     inv : c.interfaces = c.implementations.interfaces->asSet
7     inv : c.allInterfaces = c.allImplementations.interfaces->asSet->union(c.allImplementations.interfaces.allSuper->asSet)
```

Figure 5. Derived information about the linking of component items

In practice ensuring this kind of constraint proved to be difficult, since the whole graph of entities is developed concurrently by hundreds of software engineers working in different sites, without a conceptual or global view. So, *the meta model has to deal with inconsistencies, rather than ensure strict consistency.* Therefore we comment out this constraint, so this is not an invariant of the meta model. This approach permits to represent "invalid" data. Next section will show how to locate these constraint violations in practice.

While the constraint above can be discovered through the examination of the structure of the meta-model many other constraints require a better knowledge of the component model. For instance, one important requirement in the OM, is that the behavior associated by a component to an interface must be unique; this means that, within a given component, an interface must always be associated to a single implementation.

```
--context c : Component inv :
-- c.allImplementations
--    ->forall(imp1,imp2:Implementation | imp1<>imp2 implie
--        (imp1.allInterfaces->intersection(imp2.allInterfaces))
--            ->isEmpty
```

This expression translates the fact that two implementations of a component must not implement the same interface. The next section will give some examples showing how to locate and identify entities leading to such a constraint violation called *multi-adhesion*.

4. Building reverse engineering tools

Building a meta model not only improves the understanding of the component model. It also provides a very good basis to build a reverse engineering platform on which a large set of tools can be built, ranging from simple visualization tools, to complex analysis or restructuring tools. This includes for instance, tools that detect constraint violation. Developing all these tools from scratch is certainly not cost effective. Fortunately, a common platform can be derived from the meta model.

4.1. A reverse engineering platform

Figure 6 shows a simplified view of the overall architecture of the reverse engineering platform we have built. This traditional architecture for a reverse engineering environment [1,6] is made of the following parts.

• **Extractors**. The first step is to extract information from concrete software artefacts. In our case, source code and dictionaries are parsed and analyzed.

• **Repository**. The repository plays a central role in the environment. One important feature of our approach is that the structure of this repository is directly derived from the meta model.

• **Tools**. The tools generate different views on the repository. While some tools generate specific views, generic tools take as input a specification of the view to be generated. As we will see, the meta model can be directly used to express the the information to be displayed.

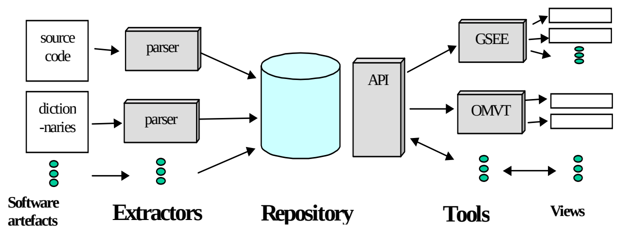


Figure 6. The reverse engineering platform

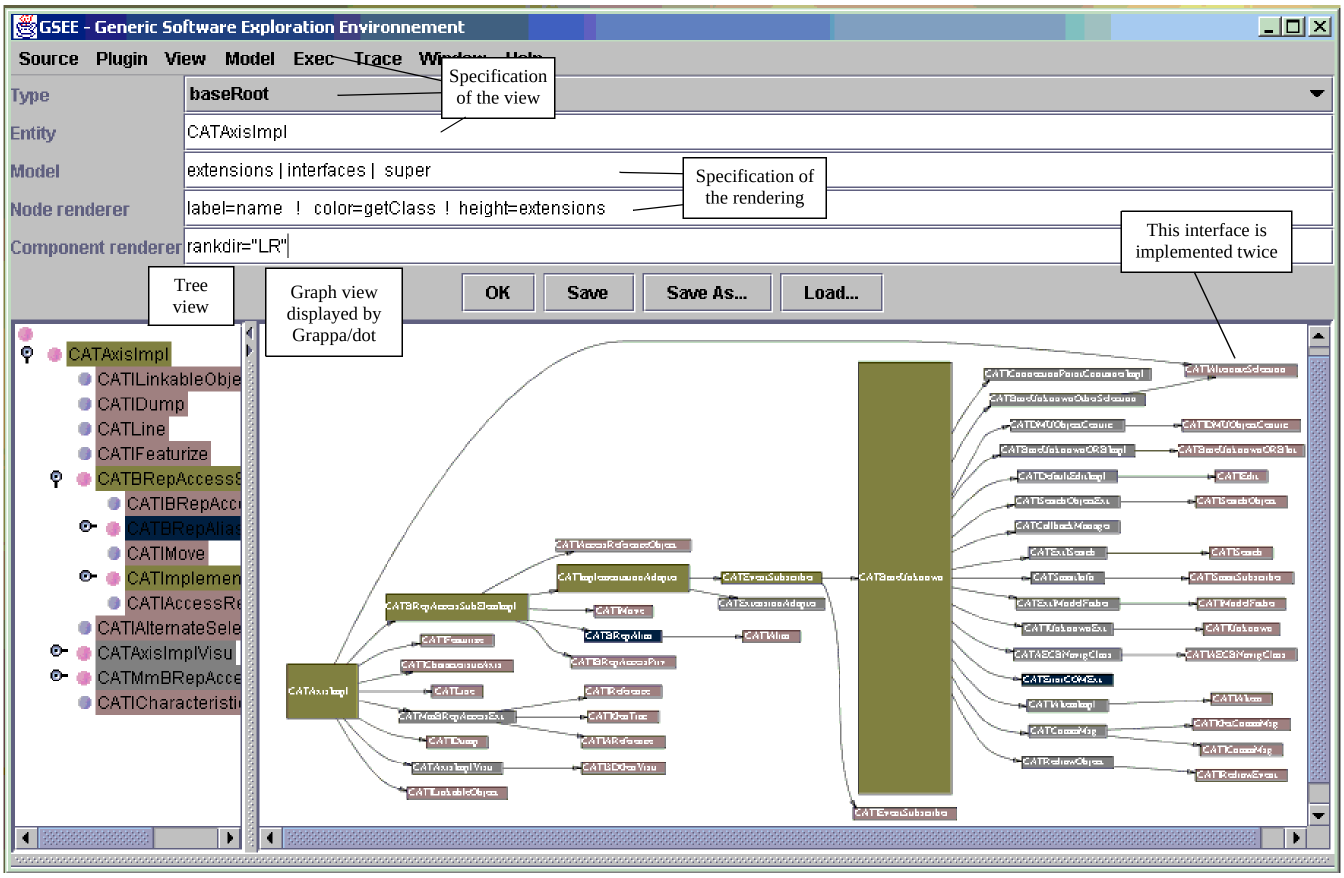
As an illustration, the next section shows how the meta model was used to build views displaying components using different visualization techniques. It then shows how inconsistencies can be found and located through the use of specific views. The experiment was done on a version of the CATIA software consisting in 4038 components made of 8155 implementations and implementing 2504 interfaces. These figures correspond to a high level of abstraction. The realization level is far more complex. In this particular case, there were 49821 C++ classes involved in the concrete representation of these components.

4.2. Example of visualization tools

Displaying components was the first application of our reverse engineering platform. This was a very interesting experiment because components are built in a blind way (through the use of macros and other low level mechanisms spread out over many files), software engineers had never actually "seen" these components.

4.2.1. Visualizing components with a generic tool. Figure 7 on the next page shows the internal view of a component displayed by GSEE, a Generic Software Exploration Environment [12]. The component is represented as a tree on the left part of the window and as a graph on the right.

GSEE is itself a generic environment. Its implementation does not contain any single line of code related to the OM. The five lines at the top of the window constitute the whole specification of the view in textual form. All roles defined in the meta model can be used. Additional information can also be derived thanks to a query language close in functionality to OCL. The main advantage of the GSEE tool is that it makes it possible to display any piece of information present in the repository at almost no cost: new views can be created interactively, just by changing the specification lines and pressing the OK button.


GSEE - Generic Software Exploration Environnement
Source Plugin View Model Exec Trace Window Help
Specification of the view
Specification of the rendering
This interface is implemented twice
Tree view
Graph view displayed by Grappa/dot
OK
Save
Save As...
Load...

Type: baseRoot
Entity: CATAxisImpl
Model: extensions | interfaces | super
Node renderer: label=name ! color=getClass ! height=extensions
Component renderer: rankdir="LR"

Figure 7. A component displayed by the Generic Software Exploration Environment (GSEE)

4.2.2. Visualizing components with a specific tool. In parallel with the definition of the meta model, we defined a graphical component language. This language is supported by a specific tool: the Object Modeler Visualization Tool (OMVT). The main benefit of this tool is that it implements specific layouts suited to the needs of different stakeholders [23]. For instance the black-box view shown in Figure 8 represents the component depicted in Figure 7 but from the client point of view. The component can be "opened", as shown in Figure 9 to discover its internal view. This last view is typically used by implementers to understand the effect of inheritance and extension features.

4.3. Examples of constraint-checking tools

As said before, rather than enforcing strict consistency, the meta model has do deal with inconsistencies. Reverse engineering tools can be built to check particular constraints.

4.3.1. Specifying constraint-checking tools. OCL can be used to specify such tools. Instead of describing a constraint as an invariant, a better approach is to define enough information to be able to identify in greater detail the occurrences of constraint violation (CV) as well as the faulty entities. For instance, as described in Section 3.4 the information contained in source code and in dictionaries should be equal. A better approach is to compute the difference between those associations.

```
context Component
  inv : onlyDicoInterfaces = declaredInterfaces - allInterfaces
  inv : onlySourceInterfaces = allInterfaces - declaredInterfaces
  inv : bothDicoSourceInterfaces=
            allInterfaces->intersection(declaredInterfaces)
```

Similarly, the following OCL expression identifies the implementations responsible for a multi adhesion.

```
context ComponentImplementationRelation
    inv : createsMultiAdhesionError =
      component.allImplementations->excluding(self)
        ->exists( imp2 : Implementation |
            self.allInterfaces
            ->intersection(imp2.allInterfaces)->notEmpty))
```

Such OCL expressions can easily be converted into code and integrated into specific tool like OMVT or transformed in to a query interpretable by GSEE.

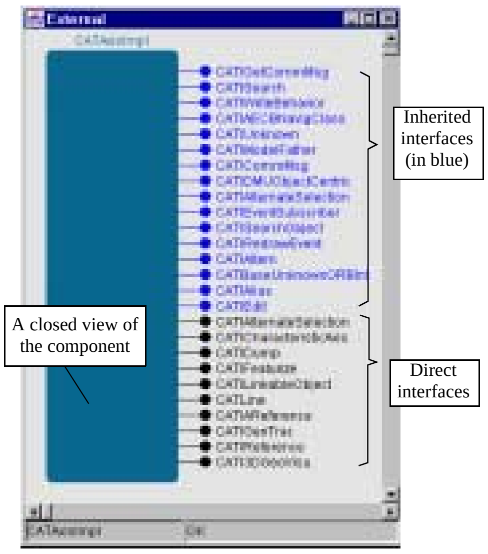

External
CATAxisImpl
Inherited interfaces (in blue)
A closed view of the component
Direct interfaces
OK

Figure 8. External view of the component

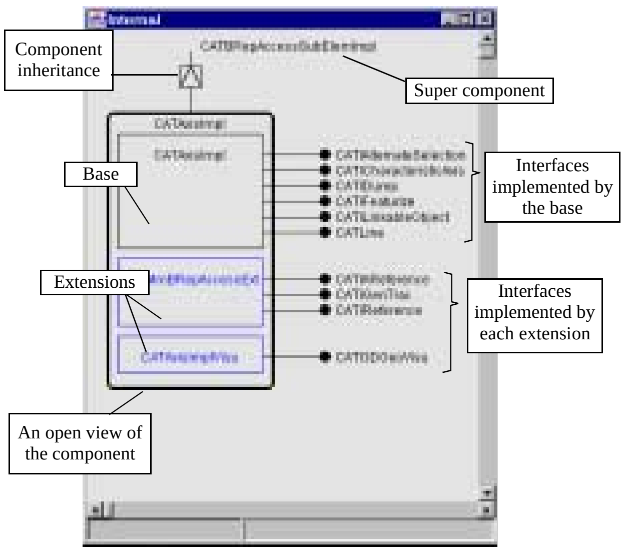
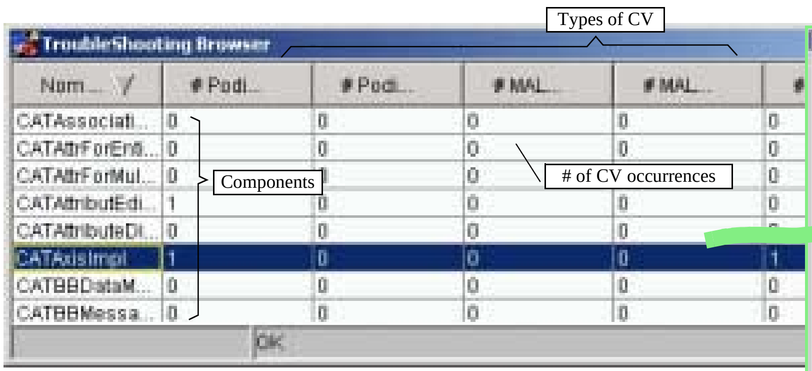

Internal
Component inheritance
Super component
Base
Interfaces implemented by the base
Extensions
Interfaces implemented by each extension
An open view of the component

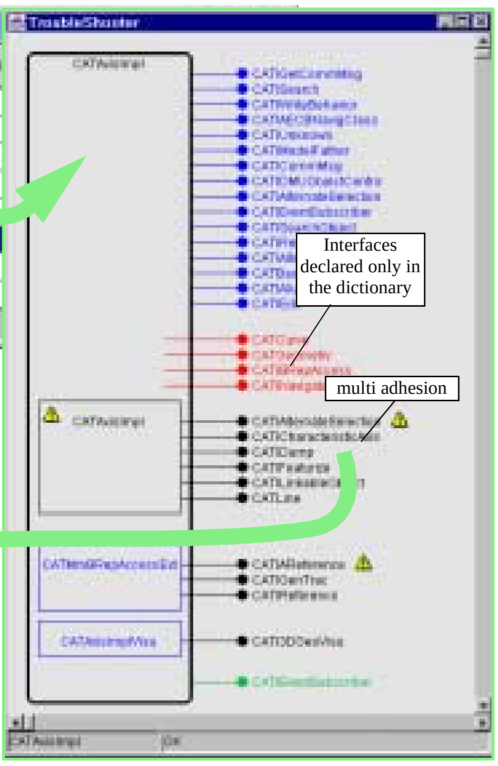
Figure 9. Internal view of the component

4.3.2. Constraint-violation exploration tools. To help software engineers in localizing CVs within large software products and understanding their cause, a set of tools was built. The following example illustrates a typical session with the OM "trouble shooter". At first, a browser is used to get an overall picture of CV occurrences within the software. Figure 10 shows, for each component, the number of CV occurrences it contains. On selecting a component, a trouble shooter view is opened Figure 11. This view gives the structure of the component emphasizing CV through the use of colors, or warning symbols. A simple click on such a symbol opens a diagnostic view focusing on the faulty entities. Figure 12 focuses on the occurrence of a multi adhesion. Global views are also available (e.g. Figure 7).



Nom...	# Podi...	# Podi...	# MAL...	# MAL...	#
CATAssociati...	0	0	0	0	0
CATAttrForEnti...	0	0	0	0	0
CATAttrForMul...	0	0	0	0	0
CATAttributEdi...	1	0	0	0	0
CATAttributeDi...	0	0	0	0	0
CATAxisImpl	1	0	0	0	1
CATBBDataM...	0	0	0	0	0
CATBBMessa...	0	0	0	0	0

Figure 10. The trooubleshooter browser



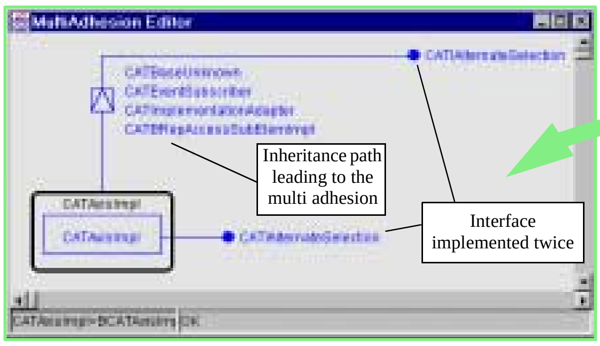

MultiAdhesion Editor
Inheritance path leading to the multi adhesion
Interface implemented twice
CATAxisImpl
OK

Figure 12. Zoom on a multi adhesion

Figure 11. Zoom on a component

5. Discussion and related work

Our approach is based on two major steps: (1) building a meta model describing the component model, and (2) building a reverse engineering platform to explore and analyse software built using this model.

5.1. About the meta model

It is important to stress that this paper concentrates only on a small but central part of the OM meta model. The complete meta model is much more complex; it also describes the realization level (C++ classes, C++ inheritance, DLLs, etc.), larger grained entities like frameworks, products, etc. Describing precisely the constraints at these various levels proved to be difficult, mainly because the model evolved over time with the underlying technology and realization techniques.

We also found that working only at the meta model level is insufficient, because it gives no information about actual instances. The first reverse engineering and exploration tools we have implemented provided us invaluable insights on the usage of the component model. For instance, we learned that some apparently important mechanisms are in fact almost not used at all. The reverse engineering tools also provided a great help in validating the meta model, through discussions with DS software architects and designers.

5.2. About reverse engineering tools

The platform is implemented in java, C++ parsers are developed by DS, and the repository is based on Object Store [35], a commercial Object Oriented Data Base. Various tools have been built around the platform. The OMVT tool, also implemented in java, represents a significant development effort, but it is clearly worth since it has been designed specifically to fit the needs of DS software engineers [23].

We also experimented with available RE3 generic tools, in particular with Rigi [19]. Our goal was to evaluate the current state of generic exploration tools and their capability to explore large component-based software products [20]. This experience show us (1) that it is easy to integrate new tool into our environment, (2) that getting first results with Rigi can take only few hours. However, this tool also shows a number of limitations in our context [20]. We thus decided to develop GSEE, the Generic Software Exploration Environment [12]. This environment has been used not only in the context of DS, but also to explore other software artifacts. Indeed, GSEE can be seen as a generalization of the approach presented in this paper. Roughly speaking, this environment is parametrized by the meta model and enables software engineers to build “any” view on virtually arbitrary set of data, by just specifying the view in terms of the meta model [12].

Scalability and performance were considered as important issues during the design and the implementation of all the tools we have built. It is interesting to notice that the use of java do not raises performance issues. Actually, extraction from source code is the bottleneck of the reverse engineering process: it takes several hours to parse the whole software developed at DS (4 millions LOC in C++). This step is done once a week, and is integrated in the whole development process of the company.

5.3. Related work

Describing industrial component models in a rigorous way is gaining an increasing attention in the academic community. For instance, the COM component model has been described using the Z notation [24]. We preferred to use UML [22] and OCL [25] since these languages are increasingly popular in industry. A similar approach has been taken recently in the definition of the Corba Component Model (CCM) [33]. In this case, the meta model is mostly seen as a documentation vehicle.

Actually, the use of meta models has been widely recognized in software engineering, but most work aims at defining new models, or describing existing and stable models with well known properties (i.e. a programming language). This contrasts with our problem, since the OM component model is evolving and a very large amount of instances are already available. This last property naturally leads to RE3 techniques. In particular meta models have been used at the intersection of OO and RE3 (e.g. Famoos [7] is based on Famix, Spool [36] is based on UML). However, these projects model OO concepts, not components. In this paper we have gone one step further: we consider that OO programming languages correspond to the realization level, and components to the conceptual level. Finally, note that the meta model we have built do not enforce strict consistency, but instead deal with inconsistencies.

In parallel with component-based approach, a very large body of work have been done in the academic community to define Architecture Description Languages (ADLs) [14][18]. These languages introduce the concepts of connector and configuration in addition to the concept of component. Unfortunately the ADL approach have failed so far to find its way to industry [17] in part because no support is provided to deal with existing software products. The lack of large industrial software products based on these concepts explain why most of research done in architecture recovery are usually based on traditional concepts like modules and dependency relationships (e.g. [15,16,19,27]).

6. Conclusion and future work

This paper represents a study of the intersection between reverse engineering and component-based software engineering. We believe that this topic will be of increasing importance as component technology will spread in industry. DS is pioneering in this domain.

Tough this paper presents the platform as a reverse engineering platform, one of our goal is indeed to build a complete architectural environment to support the *evolution* of large software products [17,23]. This environment will also include forward engineering capabilities, and other RE3 techniques like impact analysis, restructuration, etc. All existing techniques need to be revisited to be applicable at the architectural level. We found that the use of the meta model is a very good basis to develop this kind of tool.

Based on the understanding we have gained in this work, we are defining a new component model, along with the associated formalisms and tools. One way to validate this component model is to use it to develop our own platform and tools. Our current research seeks to show, on the one hand, how to apply component-based technology to build RE3 environment like GSEE [12], and on the other hand, how RE3 can be applied to component-based technology.

7. Acknowledgment

We thank Jorge Villalobos who participated in the design and development of the OMVT tool; Rosana Angles, Michelle Flamand who participated in its development and Daniel Galton who developed the parsers, Humberto Cervantes for its readings. We gratefully acknowledge talks with Jean-Francois Doué, Jacques Bacry, Philippe Zimny, Christian Ronce, Laurent Lalère from Dassault Système.

8. References

[1] R.S. Arnold; “Software Reengineering”, ISBN 0818632720, IEEE Computer Society Press, 1993

[2] R.S. Arnold; “Tutorial on Software Restructuring”, ISBN 0818606800, IEEE Computer Society Press, 1993

[3] L. Bass, P. Clements, R. Kazman; “Software Architecture in Practice”, ISBN 0201199300, Addison-Wesley, 1998.

[4] F.P. Brooks; “No Silver Bullet. Essence and Accidents of Software Engineering”, in IEEE Computer, April 1987.

[5] D. Box; “Essential COM”, ISBN 0201634465, Addison-Wesley, Jan. 1998.

[6] E.J. Chikofsky, J.H. Cross; “Reverse Engineering and Design Recovery : A Taxonomy”, in IEEE Software, January 1990.

[7] S. Ducasse, S. Demeyer, editors; "The FAMOOS Object-Oriented Reegineering Handbook", Univ. of Bern, Oct. 1999.

[8] F. Duclos, J. Estublier, R. Sanlaville; “Open architectures for Software Adaptation”, (in french) 13th International Conference on Software and Systems Engineering and their Applications (ICSSEA’2000), Dec. 2000.

[9] R. Englander; “Developing Java Beans”, O'Reilly & Associates. Jun. 1997.

[10] J.M. Favre; "Understanding-In-The-Large", 5th International Workshop on Program Comprehension (IWPC'97), 1997.

[11] J.M. Favre; “A rigorous approach to the maintenance of large portable software”, European Conference on Software Maintenance and Reengineering (CSMR’97), March 1997

[12] J.M. Favre, “GSEE: a Generic Software Exploration Environment”, submitted to the International Workshop on Program Comprehension (IWPC’2001), May 2001. http://www-adele.imag.fr/~jmfavre/GSEE

[13] M. Fowler, “Refactoring. Improving the Design of Existing Code”, ISBN 0201485672, Addison-Wesley, Nov. 1999

[14] D. Garlan; “Software Architecture: a Roadmap”, in A. Finkelstein, editor, The Future of Software Engineering, 22nd Int. Conference on Software Engineering, Jun. 2000.

[15] R. Holt et al, PBS: Portable Bookshelf Tools, http://www.turing.toronto.edu

[16] R. Kazman, S.J. Carrière; "Playing Detective: Reconstructing Software Architecture From Available Evidence", Tech. Rep. CMU-SEI-TR-010, Software Engineering Institute, 1997.

[17] Y. Ledru, R. Sanlaville, J. Estublier; “Defining an Architecture Description Language for Dassault Systèmes”,. 4th Int. Software Architecture Workshop, Jun.2000.

[18] N. Medvidovic, R.N. Taylor; “A Framework for Classifying and Comparing Architecture Description Languages”. 6th European Software Engineering Conference (ESEC’97),. LNCS 1013, Springer-Verlag, Sep. 1997.

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

[20] S.T. Nguyen, J.M. Favre, Y. Ledru, J. Estublier; "Exploring Large Software Products", (in french), 13th International Conference on Software and Systems Engineering and their Applications (ICSSEA’2000), Dec. 2000.

[21] L.B.S. Raccoon, “Fifty Years of Progress in Software Engineering”, Software Engineering Notes, Vol. 22, No 1, ACM SigSoft, Jan. 1997.

[22] J. Rumbaugh, I. Jacobson, G. Booch; “The Unified Modeling Language Reference Manual”, ISBN 020130998X, 1999.

[23] R. Sanlaville, J.M. Favre, Y. Ledru, "Helping Various Stakeholders to Understand a Very Large Software Product" submitted to IWPC’2001.

[24] K.J. Sullivan, J. Socha, M. Marchukov; "Using Formal Methods to Reason about Architectural Standards", International Conference on Software Engineering (ICSE’97), 1997

[25] J.Warmer, A. Kleppe; “The Object Constraint Language”, ISBN 0201379406, Addison-Wesley, 1999.

[26] E. Yourdon; “Re-3 : Re-engineering, Restructuring, Reverse Engineering” in American Programmer, Vol.2, No. 4, 1989.

[27] CIA/++,CIAO http:// http://www.research.att.com/~ciao/

[28] Dassault Systèmes CATIA Software.http://www.catia.com/

[29] COM Specification. Available at http://www.microsoft.com/com/resources/comdocs.asp

[30] Corba. Object Management Group. http://www.omg.org

[31] Unified Modeling Language Specification V1.3., Jun. 1999

[32] JavaBeans Specification. http://java.sun.com/products/javabeans/docs/spec.html

[33] “CCM: Corba Component Model”, OMG, August 1999

[34] Entreprise Java Bean, Sun, http://java.sun.com/products/ejb

[35] http://www.objectdesign.com/products/objectstore.html

[36] Spool Project, http://www.iro.umontreal.ca/labs/gelo/spool/