What do we do with re-use in MDA?

Nathalie Moreno and Antonio Vallecillo

Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
{vergara,av}@lcc.uma.es

Abstract. MDA seems to be one of the most promising approaches for designing and developing software applications. It provides the right kinds of abstractions and mechanisms for improving the way applications are built nowadays: in MDA, software development becomes model transformation. MDA also seems to suggest a top-down development process, whereby PIMs are progressively transformed into PSMs until a final system implementation (PSM) is reached. However, there are situations in which a bottom-up approach is also required, e.g., when *re-use* is required. Here, re-use means for instance using pre-developed COTS components to build applications, or dealing with legacy systems. Moreover, many times we are not interested in the creation of new systems but in the maintenance or evolution of existing ones. How to deal with these issues within the context of MDA? How much benefit will MDA bring to those problems? In this paper we try to introduce the main problems involved in dealing with re-use in MDA, identify the major issues, and propose some ways to address them, particularly in the context of Component-based Software Development.

1 Introduction

The Model Driven Architecture (MDA) [16, 19] is an OMG initiative that provides an approach to system development based on models. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification of systems. It provides an approach for specifying a system independently of the platform that will support it (Platform Independent Model, PIM); specifying platforms (Platform Models, PM); choosing one or more particular platforms for the system; and transforming the PIM into one (or more) Platform Specific Models (or PSM)—one for each particular platform.

Platform independence is the quality of a model to be independent of the features of a platform of any particular type. This aims at separating the business logic and rules from the technology and middleware platform(s) on which the system is implemented, protecting part of the organization investment in software development from changes in the fast-pace evolving software technologies.

Model transformation is the process of converting one model to another model of the same system. In MDA, software development becomes an iterative model transformation process: each step transforms one (or more) PIM of the system

at one level into one (or more) PSM at the next level, until a final system implementation is reached. (Here, an implementation is just another PSM, which provides all the information needed to construct a system and to put it into operation.)

This process seems to imply a top-down development process, by which models at different levels of abstraction of the system are progressively transformed (merged and/or refined) until the implementation code is finally generated. However, there are situations in which a bottom-up approach is also required. For instance, how to use and integrate pre-developed COTS components into the application? How to deal with pieces of legacy code, or with legacy applications? Furthermore, many times we are not interested in the creation of new systems but in the maintenance or evolution of existing ones. How to deal with these *re-use* issues within the context of MDA? How much benefit will MDA bring to those problems? For MDA to become mainstream, the current re-use issue has to be properly addressed.

In this paper we try to introduce the problems involved in dealing with re-use within the MDA, identify the major issues, and propose some ways to address them in the particular context of Component-based Software Development (CBSD). More precisely, the structure of this document is as follows. After this introduction, Section 2 describes the major problems that we think that need to be addressed in order to deal with re-use within MDA. Then, Section 3 presents a solution based on a set of assumptions. The feasibility of such assumptions is discussed in Section 4, which analyzes how far we currently are to overcome the problems that each assumption faces. Finally, Section 5 draws some conclusions and outlines some future lines of research.

2 The problems

There are many different problems that need to be addressed in order to deal with re-use in the context of MDA. There are general problems of system modeling and of MDA, and specific problems of COTS components and legacy systems.

2.1 Problems related to the modeling of systems and MDA

The first kind of problems are general to all system modeling approaches, and in particular to MDA: What kind of information should the model of a software system contain? How do we express such information? (Not to speak about the methodology or approach to derive the model from the user's requirements.)

First, there seems to be no consensus about the information that comprises the model of a system, a component, or a service. In this paper we will suppose that this information contains three main parts: the *structure*, the *behavior*, and the *choreography* [20]. The first one describes the major classes or components types representing services in the system, their attributes, the signature of their operations, and the relationships between them. Usually, UML class or component diagrams capture such architectural information. The *behavior* specifies the

precise behavior of every object or component, usually in terms of state machines, action semantics, or by the specification of the pre- and post-conditions of their operations (see [14] for a comprehensive discussion of the different approaches for behavior modeling). Finally, the *choreography* defines the valid sequences of messages and interactions that the different objects and components of the system may exchange. Notations like sequence and interaction diagrams, languages like BPEL4WS, or formal notations like Petri Nets or the π-calculus may describe such kind of information.

Most system architects and modelers currently use UML (class or component diagrams) for describing the structural parts of the system model. However, there is no consensus on the notation to use for modeling behavior and choreography. This is something that somehow needs to be resolved.

2.2 Problems related to COTS and legacy systems

The second set of problems is related to the COTS components or legacy systems that we need to integrate in our system. The kind of information that is available from them will allow us to check whether they match our requirements or not, as described by the system model. More precisely, this information should be able to allow us to:

(a) model the component or legacy system (e.g., by describing its structure, behavior, and choreography);
(b) check whether it matches the system requirements (this is also known as the *gap analysis* problem [8]);
(c) evaluate the changes and adaptation effort required to make it match the system requirements (i.e., evaluate the *distance* between the models of the "required" and the "actual" services, see e.g., [15]); and
(d) ideally, provide the specification of an adaptor that resolves these possible mismatches and differences (see e.g., [5, 6]).

Figure 1 shows these processes in a graphical way.

The problem is that both COTS components and legacy applications are usually back-box pieces of software for which there is no documentation or modeling information at all. Even worse, if a model of a component or legacy system exists, it may correspond to the original design but not to the actual piece of software. The current separation between the model of the system and its final implementation usually leads to situations in which changes and evolutions of the code do not reflect in the documentation—same as it happens in a building that gets refurbished but nobody cares to update the floor and electricity maps.

Some people propose the use of reverse engineering to obtain the information we require about legacy systems (basically, obtain their models from their code, whenever the code is available). Thus, a reverse transformation would convert the code of the legacy application into a fairly high-level model with a defined interface that can be used to perform all the previous tasks.

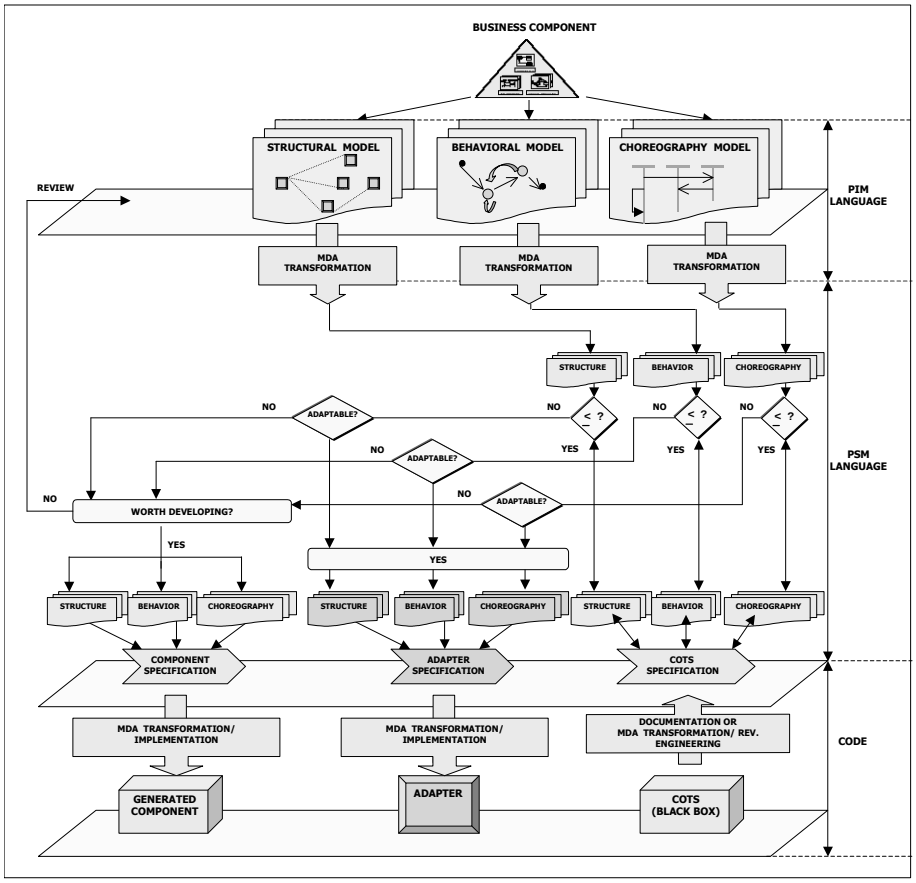

Fig. 1. Integrating COTS into the MDA chain

But the problem is that reverse engineering can only provide a model at the lowest possible level of abstraction. In fact, you can't reverse engineer an architecture of any value out of something that did not have an architecture to begin with. And even if the original system was created with a sound architecture, very often the original architecture tends to get eroded during the development process. So, what you usually get after reverse engineering is essentially just an execution model of the actual software in graphical form. At that point, most of the high level design decisions have been wiped out.

Our proposal is then to model just the interfaces to legacy systems and leave them as code—not to reverse engineer their contents. In this way we can deal with them as if they were COTS components, whose internals are unaccessible.

With regard to adaptation, an old rule of thumb claims that if more than 80% of the functionality of a component needs to be modified in order to be

Table 1. Examples of software elements and available notations for expressing their structural, behavioral, and choreography models.

Software element	Structure	Behavior	Choreography
Web Service	WSDL	RDF	BPEL4WS [9]
CORBA object	CORBA IDL	SDL [11]	Message Sequence Charts [10]
CORBA object	CORBA IDL	Larch-CORBA [12]	CORBA-Roles [7] or Petri-nets [2]
Java Class	Java	JML [13]	UML seq. diagrams
.NET assembly	C#	contracts [1]	BPEL4WS [9]

integrated into our system, it is faster (and cheaper) to develop it from scratch. In other words, if a legacy component can be wrapped and then successfully deployed with a "minor" repair/upgrade effort, then this is a reasonable approach. If any more than a "minor" effort is needed to make it match our high-level system requirements (as stated by the system PIM), then it is a strong candidate for forward engineering—of course using the existing legacy component as conceptual input.

Summarizing, the main problems related to the re-use of COTS components and legacy systems that we perceive are: the definition of the information (set of models) that needs to be provided/obtained for a piece of software in order to understand its functionality, and how to re-use it; the evaluation of the effort required to adapt it to match the new system's requirements; and the (semi)automatic generation of adapters that iron out the mismatches.

3 Assumptions for addressing these problems

This Section discusses how to address some of the issues mentioned above, making certain assumptions.

(1) First, we will suppose that we count with a model of the COTS component or legacy system that we need to re-use, with the information about its structure, behavior, and choreography. Table 1 shows some examples of software elements and the notations in which the information can be expressed. These models will constitute our target PSM.
(2) We will also suppose that the PIM of the application we are developing describes the system as a set of interacting parts, each one with the information about its structure, behavior, and choreography. (This information can be either individually modeled, or obtained for each element from the global PIM—by using projections, for example.)
(3) Third, we will assume that there are MDA transformations defined between the metamodels of the notations used in the PIM for describing the system structure, behavior and choreography, and those used in the PSM. For instance, MDA transformations from Message Sequence Charts to BPEL4WS.
(4) Fourth, we will suppose that associated to each notation for describing structure, behavior and choreography at the PSM level, there are a set of matchmaking operators that will implement the substitutability tests. These tests

are required to check whether the required business component (as specified in the PIM, and then "translated" into the PSM) can be safely *substituted* by the existing component or piece of legacy software. For simplicity, we will use the name notation ($\leq$) for referring to all these operators.
For example, at the structural level given two interfaces A and B, we shall say that $A \leq B$ if A can replace B, i.e., if A is a subtype of B using the common subtyping relations for interface signatures [21]. At the behavioural level, this operator can be defined to deal with the behavioral semantics of components, following the usual subtyping relations for pre-post conditions [22], for instance. Operator $\leq$ can also be defined for choreography models expressed using process algebras [6, 7, 17].

(5) An finally, we need to count on the existence of (semi) automated derivation of software adaptors (e.g., wrappers) that resolve the potential mismatches found by the substitutability tests.

Using all these assumptions, our approach is graphically depicted in Figure 1. As we can see, our starting point is the PIM of a business service or component. This PIM comes from the PIM of the global system, that we suppose composed of individual business services or components interacting together to achieve the system functionality. The PIM of each business service comprises (at least) three models with its structure, behavior, and choreography.

At the right hand side of the bottom of the Figure 1 we have the piece of software that we want to re-use, let it be a COTS component or a legacy application (for example, think of an external Web Service that offers the financial services we are interested in, or a COTS component that provides part of the functionality that our business requires). From its available information and/or code we need to extract its high-level models, that will the constitute the PSM of the software element. This PSM will be constructed using the information available from the COTS component or legacy application, and probably complemented with some information obtained using reverse engineering. The *Platform* in this case will be the one in which we express the information available about the element. Let us call P to that platform, and let M_s, M_b and M_c the models of the structure, behavior and choreography of the software element to be re-used, respectively.

Once we count with a PIM of the business service (our requirements) and the PSM of the available software in a platform P, we need to compare them, and check whether the PSM can serve as an implementation of the PIM in that platform. In order to implement such a comparison, both models need to be expressed in the same platform. Therefore, we will transform the three models of the PIM into three models in P, using MDA transformations. Let they be M'_s, M'_b and M'_c, respectively.

Once they are expressed in the same platform and in compatible languages, we can make use of the appropriate reemplazability operators and tools defined for those languages to check that the software element fulfils our requirements, i.e., $M_s \leq M'_s$, $M_b \leq M'_b$, and $M_c \leq M'_c$. If so, it is just a matter to use the PSM software element as a valid transformation from the PIM to that platform.

But in case the software element cannot fulfil our requirements (i.e. its PSM cannot safely replace the PSM obtained by transforming the PIM), we need to evaluate whether we can adapt it, and if so, how much is the effort involved in that adaptation. Some recent works are showing interesting results in this area [5, 6, 15]. The idea is, given the specifications of two software elements, obtain the specification of an adaptor that resolves its differences. If such an adaptor is feasible (and affordable!) we can use some MDA transformations to get its implementation from the three models of its PSM. Otherwise, it is better to forward-engineering the component, using MDA standard techniques from the original business component's PIM (left hand side of Figure 1).

Alternatively, the original PIM of the system might have to be revisited in case there is a strong requirement of using the software element, which does not allow us to develop it from scratch (e.g. in the cases of a financial service offered by an external provider such as VISA or AMEX, or of a Web Service that implements a typical service from Amazon or Adobe). In those cases, we must accommodate the software design and architecture of our system to the existing products, maybe using spiral development methods such as those described in [18].

4 Dealing with the assumptions

We have presented an approach to deal with COTS components and legacy code within the context of MDA, based on a set of assumptions. In this Section we will discuss how far we currently are from achieving these assumptions, and the work that needs to be carried out for making them become true. The assumptions were introduced in Section 3.

The first one had to do with obtaining the PSM of the piece of software to be re-used, and in particular the models of its structure, behavior and choreography. Examples of such models were presented in Table 1. Some of this information is not difficult to obtain, specially at the structure level: the signature of the interfaces of the software elements are commonly available (e.g. WSDL descriptions of Web Services, IDLs of CORBA and COM components, etc.). However, the situation at the other two levels is not so bright, and only for Web Services we think that it will resolve in a near future—this information is definitely required if re-use is to be achieved, and we perceive a clear support from software developers and vendors to re-use Web Services. Proposals for describing the choreography and behavioral semantics of Web Services are starting to be developed, and we expect to see them widely agreed soon. For the rest of the COTS components there are some small advances (see, e.g., the work by Bertrand Meyer on extracting contract information from .NET components [1]) but most of the required information will probably never be supplied [3], unless a real software marketplace for them does ever materialize.

Regarding legacy systems, the use of reverse engineering may be of great help, although it also presents strong limitations, as we previously discussed in Section 2. Basically, the result after reverse engineering is essentially just an

execution model of the actual software in graphical form, without most of the high level design decisions and architecture.

The second assumption was about counting with a PIM of the individual business services that form part of the system, with information about their structure, behavior, and choreography. Again, there seems to be no major problems with the structure, but we see how the software engineering community currently struggles to deal with the modeling of behavior and choreography of business components and services (a quick look at the discussion happening at the MDA, Business Processes, and WS-Choreography mailing lists is very illustrative).

Although there is no agreed notation for modeling behavior (or even consensus on a common behavioral model), we expect UML 2.0 to bring some consensus here: even when UML 2.0 proposed behavioral and choreography models are far from being perfect, we expect the "U" of UML to do its job. However, this also strongly depends on the availability of tools to support the forthcoming UML 2.0 standard.

The third assumption relied on the availability of MDA transformations between the metamodels of the notations used in the PIM for describing the system structure, behavior and choreography, and those used in the PSM. We expect MOF/QVT to be of great help here. In fact, there are some proposals already available that provide transformations between different languages, such as UML (Class diagrams) to Java (interfaces), EDOC to BPEL4WS, etc. [4]. They are still at a fairly low level, but they are very promising when considered from the MOF/QVT perspective.

We also supposed, as fourth assumption, the existence of formal operations ($\leq$) and tools for checking the substitutability of two specifications. The situation is easy at the structure level, since this implies just common subtyping of interfaces. However, there is much work to be done at the behavior or choreography levels, for which only a limited set of operators and tools exist (basically, the works by Gary Leavens on Larch [13, 12], and the works by Carlos Canal et al. for choreography [6, 7]).

Finally, there is also plenty of work to do with regard to the (semi) automated derivation of software adaptors (e.g., wrappers) that resolve the potential mismatches found by the substitutability tests. There are some initial results only, but most of the problems seem to be unsolved yet: defining distances between specifications [15], deciding about the potential existence of a wrapper that resolves the mismatches, generating the wrappers at the different levels, etc.

5 Concluding Remarks

In this position paper we have discussed the issues associated to re-use within the context of MDA. We have also presented a proposal, based on a set of assumptions. The problem is that these assumptions are not feasible yet. However, their identification has helped us detect some areas of research that may help solve the

problems associated to re-use, or at least alleviate it in some particular contexts, e.g., CBSD or Web Engineering.

The general problem of re-use is much more complex, though. We have oversimplified it by just concentrating on three aspects of the systems: structure, behavior and choreography. These models allow the specification and implementation of most of the "operationalizable" properties of systems: basically, its functionality and some QoS and security requirements. However, how to deal with the extra-functional requirements (e.g. robustness, stability, usability, demonstrability, maintainability,etc.)? Many of these requirements are more important than functionality when it comes to reuse or upgrade an existing system. As usual, we will leave them for future research.

Acknowledgements. This work has been partially supported by Spanish Project TIC2002- 04309-C02-02.

References

1. K. Arnout and B. Meyer. Finding implicit contracts in .NET components. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (First International Symposium, FMCO 2002)*, number 2852 in Lecture Notes in Computer Science, pages 285–318, Leiden, The Netherlands, 2003. Springer-Verlag, Heildelberg. http://www.inf.ethz.ch/ meyer/publications/extraction/extraction.pdf.

2. R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In *Proceedings of ECOOP'99*, number 1628 in Lecture Notes in Computer Science, pages 474–494, Lisbon, Portugal, 14–18 June 1999. Springer-Verlag, Heildelberg.

3. M. F. Bertoa, J. M. Troya, and A. Vallecillo. A survey on the quality information provided by software component vendors. In *Proc. of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2003)*, pages 25–30, Darmstadt, Germany, 21 July 2003.

4. J. Bézivin, S. Hammoudi, D. Lopes, and F. Jouault. An experiment in mapping web services to implementation platforms. Reserach Report 04.01, University of Nantes, Mar. 2004.

5. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering (in press)*, 2004.

6. A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web services choreographies. In *Proc. of the 1st International Workshop on Web Services and Formal Methods (WS-FM'04)*, volume 86 of *Electronic Notes in Theoretical Computer Science*, pages 1–20, Pisa, Italy, Sept. 2004. Elsevier.

7. C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding roles to CORBA objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, Mar. 2003.

8. J. Cheesman and J. Daniels. *UML Components. A simple process for specifying component-based software*. Addison-Wesley, Boston, 2000.

9. IBM. *Business Process Execution Language for Web services (BPEL4WS) 1.1*. IBM and Microsoft, May 2003. Available at http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

10. ITU-T. *Message Sequence Charts*. International Telecommunications Union, Geneva, Switzerland, 1994. ITU-T Recommendation Z.120.

11. ITU-T. *SDL: Specification and Description Language*. International Telecommunications Union, Geneva, Switzerland, 1994. ITU-T Recommendation Z.100.

12. G. T. Leavens. Larch-corba. http://www.cs.iastate.edu/ leavens/main.htmlLarchCORBA.

13. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999. JML web page: http://www.cs.iastate.edu/ leavens/JML.html.

14. A. McNeile and N. Simons. Methods of behaviour modelling, Apr. 2004. http://www.metamaxim.com/download/documents/Methods.pdf.

15. R. Mili, J. Desharnais, M. Frappier, and A. Mili. Semantic distance between specifications. *Theoretical Comput. Sci.*, 247:257–276, Sept. 2000.

16. J. Miller and J. Mukerji. *MDA Guide*. Object Management Group, Jan. 2003. OMG document ab/2003-05-01.

17. O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

18. B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117, Mar. 2001.

19. OMG. *Model Driven Architecture. A Technical Perspective*. Object Management Group, Jan. 2001. OMG document ab/2001-01-01.

20. A. Vallecillo, J. Hernández, and J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, number 1964 in Lecture Notes in Computer Science, chapter 21, pages 256–269. Springer-Verlag, Heidelberg, 2000.

21. A. M. Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Trans. on Software Engineering and Methodology*, 4(2):146–170, Apr. 1995.

22. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. on Software Engineering and Methodology*, 6(4):333–369, Oct. 1997.