Software Adaptation in the Context of MDA

Nathalie Moreno, José Raúl Romero and Antonio Vallecillo

Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
{vergara,jrromero,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. 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 to adapt these systems when we are using an MDA approach for building our final application? In this paper we try to introduce the main problems involved in dealing with *software adaptation* in MDA, identify the major issues, and propose some ways to address them, particularly in the context of Component-based Software Development.

1 Introduction

Component-based software engineering is an emergent discipline that promises to reduce development costs by creating a marketplace of pre-produced components, that can be effectively used for building applications. Since components may use different technologies and platforms, the possibility of reuse existing software is a difficult problem to be addressed, so existing components may work properly within new applications. Thus, *software adaptation* is required to guarantee that different components will be able to interact in the right way both at the syntactical and at the protocol and semantical levels. In this sense, the Model Driven Architecture (MDA) [12] has recently appeared as an interesting approach to address software adaptation and interoperability.

MDA allows us to: describe a system independently of the platform that will support it (Platform Independent Model, PIM); specify platforms (Platform Models, PM); select one or more particular platforms for the system; and transform the PIM into one (or more) Platform Specific Models (PSM) — one for each particular platform. 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.

MDA seems to imply a top-down development process. 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 third-party 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?

In this paper we introduce some problems involved in dealing with software adaption within the context of the MDA approach. More specifically, the main problems concerning to the reuse of COTS and legacy systems that we perceive are: (*a*) 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; (*b*) the evaluation of the effort required to adapt it to match the new system's requirements; and (*c*) the (semi)automatic generation of adapters that iron out the mismatches. After identifying some major issues, we propose ways to address them in the particular context of Component-based Software Development (CBSD).

The structure of this paper is as follows. After this introduction, Section 2 describes the major issues to be considered when reusing software pieces from different technologies. Then, Section 3 discusses how to address some of these issues concerning to software adaption within the context of the MDA. Finally, Section 4 draws some conclusions and open lines of research.

2 Adaptation for re-use

Most of the existing approaches deal with software adaptation in a platform and environment dependent way. In consequence, adaptors obtained by such techniques do not seem to be as reusable as it is desirable. From our point of view, this matter might be faced from a higher level of abstraction, e.g., at the model level. In particular, MDA provides an approach for specifying a system independently of the platform that will support it. However, several issues need to be firstly answered, such as: What kind of information should the model of a software system contain? How do we express such information?

Issues related to system modeling. 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* [14]. The former 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 [10] 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.

Issues related to components and legacy applications. Sometimes, components and legacy applications also need to be integrated in systems. Thus, the kind of information that is available from them will allow us to check whether they match the system requirements or not. 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 [7]);
(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 [11]); and
(d) ideally, provide the specification of an adaptor that resolves these possible mismatches and differences [4, 5]).

The problem is that both COTS components and legacy applications are usually black-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.

Some authors 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.

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.

3 Modeling adaptors with MDA

Our proposal discusses how to address some of the problems mentioned in the introduction concerning to software adaption within the context of the MDA, making certain assumptions.

(1) We count with a model of the component or legacy system that we need to re-use (e.g., structure, behavior and choreography).
(2) The PIM of the application 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) 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.
(4) Associated to each notation for describing structure, behavior and choreography at the PSM level, there are a set of matchmaking operators ($\leq$) that will implement the substitutability tests. These tests are required to check whether the required business component can be safely *substituted* by the existing piece of software.
(5) We count on the existence of (semi)automated derivation of software adaptors (e.g., wrappers) that resolve the potential mismatches found by the substitutability tests.

As shown in Figure 1, our starting point is the PIM of a business service or component. As previously mentioned, 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 (e.g., an external Web Service that offers the financial services we are interested in). From its available information and/or code we need to extract its high-level models, that will constitute the PSM of the software element (and perhaps enriched with some information inferred 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,

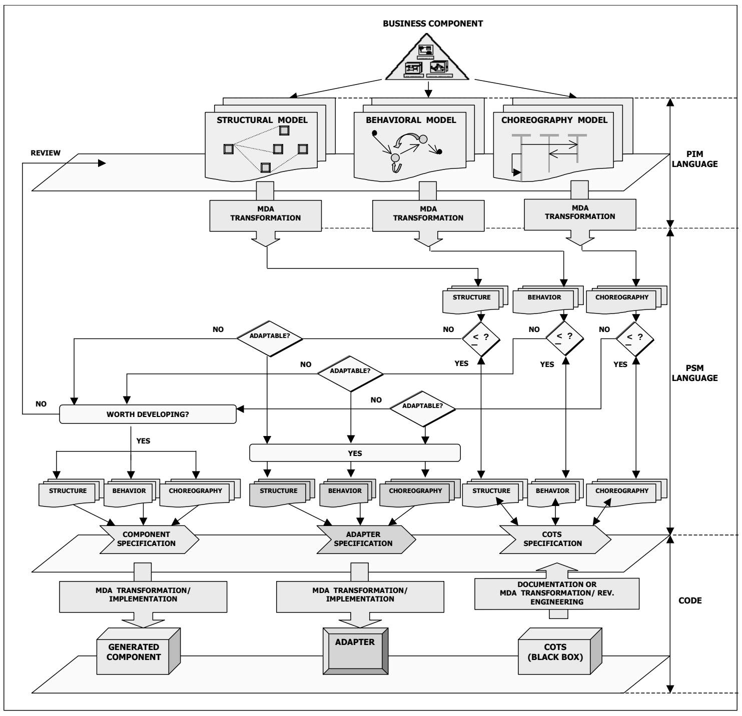

Fig. 1. Integrating COTS into the MDA chain

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 [4, 11]. 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 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 [13].

4 Concluding Remarks

The general problem of re-use is much more complex, though. Although we have over-simplified it, in this position paper we have discussed the major issues associated to re-use within the context of MDA. However, how to deal with the extra-functional requirements (e.g. robustness, usability, etc.)? Many of these requirements are even more important than functionality when it comes to reuse or upgrade an existing system. More specifically, we have presented an approach to deal with COTS components and legacy code, based on a set of assumptions. At this point, how far we currently are from achieving these assumptions ? What work need to be carried out for making them become true?

Some of the required 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). However, the situation at the other two levels is not so bright, and only for Web Services might definitely be resolved in a near future. For the rest of the components there are some small advances (see, e.g., the work by Meyer [1] on extracting contract information from .NET components) but most of the required information will probably never be supplied [2], unless a real software marketplace for them does ever materialize.

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. However, this also strongly depends on the availability of tools to support the forthcoming UML 2.0 standard.

Regarding to MDA transformations, there are some proposals already available that provide correspondences between different languages, such as UML (Class diagrams) to Java (interfaces), EDOC to BPEL4WS, etc. [3]. They are still at a fairly low level, but they are very promising when considered from the MOF/QVT perspective.

We also supposed 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 [9], and the works by Carlos Canal et al. for choreography [6]).

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 [11], deciding about the potential existence of a wrapper that resolves the mismatches, generating the wrappers at the different levels, etc.

Acknowledgements. The authors would like to thank the anonymous referees for their helpful comments and remarks. This work has been supported by Spanish Research Project TIC2002- 04309-C02-02.

References

1. K. Arnout and B. Meyer. Finding implicit contracts in .NET components. In *Formal Methods for Components and Objects (First International Symposium, FMCO 2002)*, no. 2852 in LNCS, pp. 285–318, 2003. Springer-Verlag.
2. 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 QAOOSE*, pp. 25–30, Germany, 2003.
3. 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, 2004.
4. 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*, 2004.
5. A. Brogi, C. Canal, E. Pimentel and A. Vallecillo. Formalizing web services choreographies. In *Proc. of the 1st Intl. Workshop on Web Services and Formal Methods (WS-FM'04)*, vol. 86 of *ENTCS*, pp. 1–20, Italy, 2004. Elsevier.
6. 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, 2003.
7. J. Cheesman and J. Daniels. *UML Components. A simple process for specifying component-based software.* Addison-Wesley, 2000.
8. ITU-T. *SDL: Specification and Description Language.* Intl. Telecommunications Union, Switzerland, 1994. ITU-T Rec. Z.100.
9. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Kluwer Academic Publishers, 1999.
10. A. McNeile and N. Simons. Methods of behaviour modelling, 2004. `http://www.metamaxim.com/download/documents/Methods.pdf`.
11. R. Mili, J. Desharnais, M. Frappier, and A. Mili. Semantic distance between specifications. *Theoretical Comput. Sci.*, 247:257–276, 2000.
12. J. Miller and J. Mukerji. *MDA Guide.* Object Management Group, 2003. OMG document `ab/2003-05-01`.
13. B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117, 2001.
14. A. Vallecillo, J. Hernández, and J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, no. 1964 in LNCS, pp. 256–269. Springer-Verlag, 2000.