


Resolving Architectural Mismatches of COTS Through Architectural Reconciliation

View metadata, citation and similar papers at core.ac.uk

brought to you by  **CORE**

provided by University of Groningen Digital Archive

Paris Avgeriou and Nicolas Guelfi

Software Engineering Competence Center (SE2C), University of Luxembourg,
6, rue Richard Coudenhove-Kalergi L-1359 Luxembourg-Kirchberg, Luxembourg
{paris.avgeriou, nicolas.guelfi}@uni.lu

Abstract. The integration of COTS components into a system under development entails architectural mismatches. These have been tackled, so far, at the component level, through component adaptation techniques, but they also must be tackled at an architectural level of abstraction. In this paper we propose an approach for resolving architectural mismatches, with the aid of architectural reconciliation. The approach consists of designing and subsequently reconciling two architectural models, one that is forward-engineered from the requirements and another that is reverse-engineered from the COTS-based implementation. The final reconciled model is optimally adapted both to the requirements and to the actual COTS-based implementation. The contribution of this paper lies in the application of architectural reconciliation in the context of COTS-based software development. Architectural modeling is based upon the UML 2.0 standard, while the reconciliation is performed by transforming the two models, with the help of architectural design decisions.

1 Introduction

The inevitable problem with reusing COTS components is that they simply don't correspond perfectly to the requirements specification and consequently to the envisioned architecture of the system [1]. Even when COTS-based systems are designed by taking into consideration pre-existing components from the market that roughly correspond to the requirements, eventually there will still be disparities when the COTS are integrated. One of the major causes of this problem is **architectural mismatches**: differences between a COTS component and the software system, where it will be integrated, which occur when the former makes the wrong assumptions about the latter [1, 8]. For example, a commercial component can falsely assume that it is in charge of controlling the sequence of interactions between itself and other components, or that other components should comply with specific protocols of interactions. To make matters worse, such assumptions are implicit and are usually in conflict with each other. The consequences are that system-wide properties are diverged from the requirements, both functional and quality ones. Especially quality requirements such as performance, reliability, and flexibility that depend profoundly on the architecture [4, 5, 24] may be to a large extent distressed by the use of COTS components.

The research community has attempted to tackle the problem of architectural mismatches, focusing on the component level, by means of **component adaptation** techniques, which attempt to incorporate unintended changes in a component for use in a particular application [3]. These techniques are distinguished into **white-box** (e.g. inheritance) and **black-box** (e.g. wrapping), depending on whether the component itself is adapted or whether its interface is adapted [4]. In the case of COTS components, black-box techniques are usually applied since the component's source code is usually prohibited from being inspected or modified [1]. There are several techniques proposed so far [3, 12, 15, 16, 29], and they can be applied according to the context of use and the possible benefits and liabilities they entail [12].

However architectural mismatches cannot only be resolved at the component level since they do not concern an isolated component but they affect a greater part of the system, which collectively includes a number of components and connectors [8, 25]. Architectural mismatches caused by a single component may influence not only the components that communicate with it but may also be propagated further on to other components. Therefore such mismatches may require not only the adaptation of the COTS component but also the modification, addition or removal of other architectural elements. In order to perform these changes we need to examine a greater part of the system's architecture, identify those elements that are affected and subsequently decide on how exactly the architecture should be modified. We thus need to tackle the problem of architectural mismatches from an architectural perspective [8].

This paper proposes an approach to resolve architectural mismatches, caused by integrating COTS, using the technique of **architectural reconciliation**. In specific, it suggests the design and subsequently the reconciliation of two architectural models: one that is forward engineered from the requirements specification and a second that is reverse-engineered from the COTS-based system implementation. The former expresses the architectural decisions in an *ideal* system, which conforms to the requirements. The latter not only grasps the implementation constraints, but also explicitly specifies the architectural impact of COTS that were incorporated in the implementation, making their design assumptions explicit, with respect to the rest of the application. These two models are reconciled into a third model that will combine the two perspectives in the best possible tradeoff, by taking under consideration the design assumptions of the COTS components, but also addressing the requirements, to the best possible extent. The reconciliation is performed by transforming the two models, based on architectural design decisions, depending on which side, requirements or implementation should be more supported. The reconciled model can eventually be used to re-engineer the COTS-based system and also update the requirements. Architectural modeling is based upon the UML 2.0 standard.

The rest of the paper is organized as follows: section 2 provides the details of the proposed approach for resolving architectural mismatches through architectural reconciliation. Section 3 illustrates the implementation of the approach through a case study while Section 4 presents some related research work with respect to architectural reconciliation. Finally Section 5 wraps up with conclusions and future work.

2 Architectural Reconciliation

2.1 The Reconciliation Process

The process of reconciliation is graphically illustrated in Fig. 1., and is comprised of six consecutive phases.

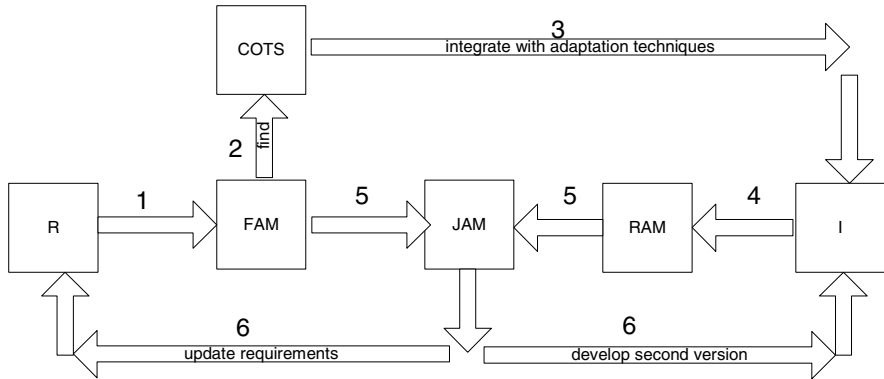


Fig. 1. Process of Architectural Reconciliation

The first three phases follow a simplistic forward engineering style. The process commences by using the requirements specification (R) to design the *ideal* architecture of the system, which we name the **Forward Architectural Model** or **FAM**. This model should, if possible, take into account pre-existing COTS from the market that correspond more or less to the requirements. This forward-engineering design of the architecture can be performed by following any architecture-driven software development process. We thus do not impose or even suggest a specific process to be followed, since we consider that our approach is independent of specific processes. In sequence, commercial components are located in the market, that is, if they haven't already been found. Eventually the implementation (I) is developed according to the FAM, by building new components from scratch and by including the COTS found. At the best-case scenario, the COTS components will be adapted at a component level according to one of the aforementioned component adaptation techniques.

The fourth phase is to reverse-architect the COTS-based implementation in order to *recover* its architecture, which we name the **Reverse Architectural Model** or **RAM**. It is obvious that reverse-architecting is a special case of reverse-engineering, which concerns only architectural design. Here, similarly as before, we do not prescribe a specific reverse-architecting approach, though there are a few such techniques and tools proposed, such as those in [11, 20, 22, 23, 25, 27, 28].

The fifth and most crucial phase is to bridge the RAM and the FAM into the **Joint Architectural Model** or **JAM**, which must compromise between the COTS-based implementation and the set of ideal requirements. This is achieved by performing a transformation, which accepts the RAM and the FAM as inputs and produces the JAM as the output. A necessary tradeoff must of course be made since it is highly im-

possible to perfectly satisfy the requirements, especially the non-functional or quality requirements. The transformation enforces a set of design decisions that resolve the incompatibilities between the RAM and the FAM. In specific, the architect must go through the following steps:

- **Identify the architectural mismatches between the RAM and the FAM.** The architect must start by looking for the four different kinds of false assumptions that integration of COTS components may entail, as explained in [8]. These assumptions may lead to architectural mismatches, or more simply differences between the FAM and the RAM, that must be explicitly specified. The architectural mismatches can be detected by comparing the RAM and the FAM, either informally (e.g. UML diagrams) or more formally (e.g. formal models with precise semantics).
- **Resolve the architectural mismatches.** By resolving the architectural mismatches, the architect needs to decide between one of the following:
 - Keep the part of the FAM and delete the part of the RAM that causes the mismatch, if enforcing the requirements is more significant.
 - Keep the part of the RAM and delete the part of the FAM that causes the mismatch, if requirements can be compromised in favor of the COTS components.
 - Come up with a tradeoff solution that mixes both parts. In this case some of the elements from both models may be deleted, others may be retained and possibly modified, while more elements may be added. Component adaptation techniques can be again enforced here, if it is necessary to adapt the behavior of COTS components.
- **Complete the JAM.** The resolution of the architectural mismatches will probably have consequences to other architectural elements that were not themselves part of the problem. Therefore, the architect needs to take some last decisions with respect to keeping, deleting or modifying architectural elements that were affected by the reconciliation actions.

The final phase in this process is to re-engineer the system according to the JAM, and update the requirements document to reflect the changes that occurred during the reconciliation. How exactly the JAM is implemented into code is again out of the scope of this paper. We emphasize that our goal in this process was not to invent yet another forward or reverse-architecting process, but to focus on the reconciliation of architectural models.

2.2 The Architectural Description

An architectural description is comprised of multiple views [6, 13, 14, 17], for example the component-connector view, the logical view, the implementation view, the data view and the deployment view. In order to reduce the complexity of bridging two complex multiple-view architectural models, we have focused on the *component-and-connector* view [6] for two reasons: it is considered to contain the most significant architectural information, and it is the most appropriate view to describe COTS components. This view deals with the system run-time by showing the *components*, which

are units of run-time computation or data-storage, and the *connectors*, which are the interaction mechanisms between components.

As far as the language for describing the architecture, we have selected the widely accepted Unified Modeling Language. We have been working on the emergent UML 2.0 standard, to describe the component and connector view, and especially chose modeling elements from the Composite Structures and Components packages, namely: components, connectors, interfaces, ports, and classes that belong to the internal structures of components. In UML 2.0 components are associated with provided and required interfaces and may own ports that formalize their interactions points. A special case of connectors, that are called *assembly connectors* connect the required interface of one component to the provided interface of a second. For more information, in [2] we have elaborated on the UML 2.0 elements for describing the component and connector view.

3 A Case Study

The system that was used as a case study for the approach, is a popular open-source Learning Management System, named Ganesha [7], which supports e-learning in higher education and training institutes. This system was chosen for two reasons: a) being an open-source project, its code can be inspected and thus re-engineered without the copyright issues of commercial systems; b) its simple PHP-based and medium-sized code makes it manageable and suitable for this kind of experiment. We have experimented with integrating various COTS components in this system, in order to check the validity of the method. For illustrative purposes, this section focuses on the integration of a particular commercial chat component. Ganesha already had a simple chat component, which allowed for basic chat functionality, but we attempted to replace it with a COTS component, which offered more advanced functionality.

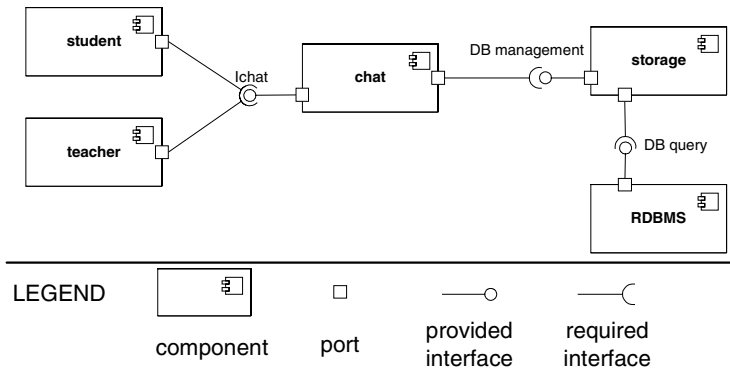


Fig. 2. Part of the Forward Architectural Model concerning the Chat Component

Fig. 2 depicts the *chat* component as well as the rest of the components, which it interacts with, in the Forward Architectural Model, designed to conform to the re-

quirements. The *chat* component provides its functionalities through the *Ichat* interface, which is used by the *student* and *teacher* components that implement the application logic for students and teachers. The *Ichat* interface mandates that the *student* and *teacher* components call the *chat* component, by passing a unique identifier as a parameter, that proves they are authorized to use it. The *chat* component needs to query and update the database in order to store the currently-connected users, and maintain a log file of conversations. It accesses the database by using the interface *database management*, offered by the *storage* component, which in sequence handles direct *database queries* to the *RDBMS*.

We then integrated the new commercial chat component into Ganesha, which we had located in the component market. This specific component was provided as a fully functional evaluation version, implemented as a Java servlet, which can be parameterized through a text configuration file. The integration of the COTS component into the system, yielded the reverse architectural model, as shown on Fig. 3. The new *chat2* component provides a slightly different interface, called *Ichat2*, since there is a new way of calling the servlet and passing parameters. For the same reason the *student* and *teacher* components are also slightly modified (*student2* and *teacher2*) in order for them to require this new interface. Also the new chat component offers an interface for *WML access*, so that mobile clients can connect and access the chat functionality. Other than that, the COTS component makes two false assumptions that lead to architectural mismatches:

- The component assumes that it can have direct access to the database and thus requires an interface from the *RDBMS* to connect and perform queries. In this sense, it overrides Ganesha's database access mechanism through the *storage* component.
- The component assumes that it should not take care of access control, but can allow any potential web client to call the servlet and participate to the chat. This assumption is again wrong in the context of a Learning Management System, which mandates a strict access control to students and teachers registered for a particular course.

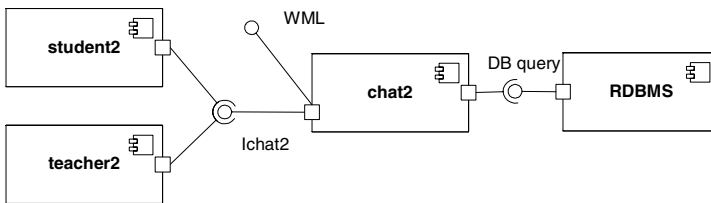


Fig. 3. Part of the Reverse Architectural Model concerning the Chat Component

In the first step of the reconciliation process, the architectural mismatches, which are caused by the above false assumptions, are identified:

- database access should be performed indirectly, as the *chat* component does through the *database management* interface in the FAM; however it is performed directly by the *chat2* component through the *database query* interface in the RAM.

- access control is managed by the *Ichat* interface of the *chat* component, but it is not managed by the *Ichat2* interface of the *chat2* component.

In the second step, that is the resolution of the mismatches, it is obvious that the *chat* component in the FAM and the *chat2* component in the RAM cause both mismatches. We cannot keep either component as it is, so the design decision is to use the wrapping adaptation technique [3], in order to adapt the *chat2* component to the functionality of the *chat* component. In specific, the wrapping technique involved a new component, the *wrapped chat*, which encapsulates the *chat2* component and delegates requests from other components to it and vice versa. The two assumptions were resolved as follows:

- The assumption about the direct database access is resolved by having the *wrapped chat* forwarding SQL queries that were previously meant to go directly to the *RDBMS*, to the *storage* component through its *DB management* interface.

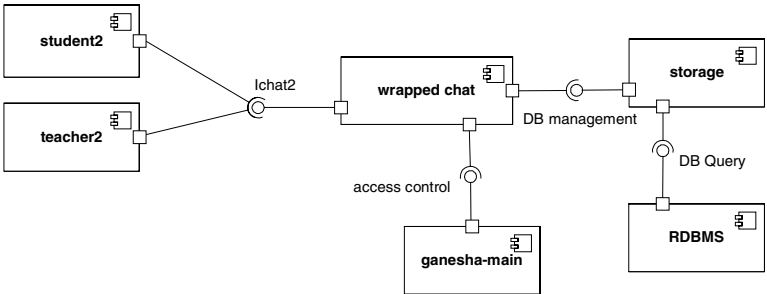


Fig. 4. Part of the Joint Architectural Model concerning the Chat Component

Table 1. Correspondence of architectural elements in the three models

FAM	RAM	JAM
student	student2	student2
teacher	teacher2	teacher2
Chat	chat2	wrapped chat
Ichat	Ichat2	Ichat2
Storage	-	storage
DB management	-	DB management
RDBMS	RDBMS	RDBMS
DB query	DB query	DB Query
-	WML	-
-	-	ganesha-main
-	-	access control

- The assumption concerning the lack of access control is resolved by having the *wrapped chat* check if each client that requests to connect to the *chat2* component

is authorized through the *access control* interface that the *ganesha-main* component provides. If the client indeed has access rights, the chat invocation is forwarded to the *chat2* component.

Completing the JAM in the third step involved the following decisions:

- The *storage* component of the FAM is required by the *wrapped chat* so it is retained in the JAM.
- The *ganesha-main* component comes neither from the FAM or the RAM, but it is a central component of Ganesha that provides an *access control* interface, and thus it is added to the JAM.
- Since the *Ichat2* interface is provided by the *wrapped chat* component, the *student2* and *teacher2* components were retained from the RAM.
- The *WML access* interface of the COTS component is not needed in the FAM, which expresses the requirements, and was thus removed in the joint architectural model.

The reconciliation process resulted in the JAM, which is illustrated in Fig. 4, while the correspondence between the elements of all three models is shown in Table 1.

4 Related Work

The approach described in this paper has been based on research work with respect to bridging the gap between the system implementation and its requirements. Perry and Wolf in [21] first introduced the architectural problems of *erosion* and *drift*, which express the phenomenon of having the implementation architecture driven away from the ideal architecture, either on purpose or due to indifference. In [25, 26], Tran et al. introduced an architecture ‘repair’ technique for fixing this gap, by discovering and further eliminating the differences between the ideal architecture and the implementation architecture. They distinguish between *forward repair* where the implementation architecture is altered to match the conceptual, and *reverse repair* for the opposite. Architectural repair is then performed by combining both forward and reverse repair. They have also defined a number of repair techniques for removing unexpected dependencies from the architectural models [25]. They do not propose an approach for performing the design of the conceptual architecture but they do suggest tools such as those in [22, 23] for reverse-architecting.

Roughly, the same problem has been dealt with in [19], where Medvidovic et al. propose the introduction of two intermediate steps: a) designing the ‘discovered’ architecture from the requirements and b) designing the ‘recovered’ architecture from the implementation. These two architectural models are then much easier bridged into the actual Architecture of the system. The ‘discovery’ of the architecture is performed using the CBSP method [9] that transforms the requirements into a handful of simple architectural elements that represent something between requirements and architecture. The ‘recovery’ of the architecture is performed using a blend of techniques that reverse-engineer the code and package the derived classes into architectural elements. The final bridging is performed manually by applying architectural styles to one of the two models and then mapping the second model to the outcome, or by first integrating the two models and then applying architectural styles.

Our own approach has been influenced by both the aforementioned approaches. However we propose specific actions on how to perform the reconciliation, by transforming the two models based on design decisions. We also do not use repair techniques for removing dependencies in the models, but decisions for modifying, removing or retaining the elements of both models. Finally we extend these approaches by working on providing formalisms for the definition of the architectural models and subsequently their transformations, as will be explained in the next section.

5 Conclusions and Future Work

In this paper we have argued that COTS-based software development entails architectural mismatches that must be dealt with, not only at a component level through component adaptation techniques, but also at the architectural level. By doing so, we can examine a number of components and their connectors in a group, and thus make modifications to a considerable part of the system's architecture. We have thus proposed to design two architectural models, the first based on the requirements and the second based on the existing implementation, and then reconciling these two models through a tradeoff decision process. The added value of our approach concerns the adoption of architectural reconciliation in the context of COTS-based software development in order to resolve architectural mismatches at an architectural level.

We are currently working on formalizing the specification of the architectural models as well as their transformation, based on our previous work on model transformation [2, 10]. Our approach is established on defining the reconciliation as a mathematical relationship between a subset of our UML 2.0 architectural models (FAM, RAM and JAM). We specify this relationship by employing a logical formula that in turn uses a pre-defined formal metamodel defined for the architectural models. This formalization of the architectural models and their transformations will provide further added value to our work by allowing an explicit and simple specification of the reconciliation and offering support for semi-automatic reconciliation.

References

1. Albert, C., Brownsword, L. "Evolutionary Process for Integrating COTS-Based Systems (EPIC)". SEI Technical Report CMU/SEI-2002-TR-005. Software Engineering Institute, Carnegie Mellon University, 2002.
2. Avgeriou, P., Guelfi, N. Perrouin, G., Evolution Through Architectural Reconciliation, workshop on Software Evolution Through Transformations (SETra) 2004, Rome, Italy, Electronic Notes in Theoretical Computer Science, Elsevier, 2004.
3. Bosch, J., Superimposition: A component adaptation technique, *Information and Software Technology*, No. 41, pp. 257-73, April 1999.
4. Bosch, J., Design and Use of Software Architectures. Addison-Wesley, 2000.
5. Clements, P., Kazman, R., Klein, M., Evaluating Software Architecture, Addison-Wesley, 2002.
6. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2002.
7. Ganesha web site. <http://www.anemalab.org/ganesha/>.

8. Garlan, D., Allen, R. and Ockerbloom, J., "Architectural Mismatch: or Why It's Hard to Build Systems Out of Existing Parts," Proceedings of the International Conference on Software Engineering, Seattle, 1995.
9. Grunbacher, P., Egyed, A. and Medvidovic, N., Reconciling Software Requirements and Architectures with Intermediate Models, Journal of Software and Systems Modeling (SoSyM), to appear.
10. Guelfi, N., Ries, B., Sterges, P., *MEDAL: A CASE Tool Extension for Model-driven Software Engineering*, SwSTE'03 IEEE International Conference on Software - Science, Technology & Engineering, Hertzeliyah, Israel, 2003
11. Guo, G. Y., Atlee, J. M. and Kazman, R., A Software Architecture Reconstruction Method. *WICSA-I*, San Antonio, Feb. 1999.
12. Heineman, G., A model for designing adaptable software components, Twenty-second International Conference on Computer Software and Applications Conference (COMPSAC), pp. 121-127, Vienna, Austria, August, 1998.
13. Hofmeister, C., Nord, R. and Soni, D., Applied Software Architecture, Addison-Wesley, 1999.
14. IEEE, Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE std. 1471-2000, 2000.
15. Keller, R. and Hölze, U., Binary component adaptation, Technical report TRCS97-20, University of California, Santa Barbara, December 1997.
16. Kiczales, G., Lamping, J., Lopes, C., Maeda, C., Mendhekar, A., Murphy, G., Open implementation design guidelines, Proceedings of the 19th international conference on Software engineering, p.481-490, May 17-23, 1997, Boston, Massachusetts, United States
17. Kruchten, P., "The 4+1 view model of architecture", IEEE Software, November 1995.
18. Medvidovic, N., Taylor, R.N., "A classification and comparison framework for software architecture description languages". *IEEE Transactions on Software Engineering*, vol.26, (no.1), p.70-93, Jan. 2000.
19. Medvidovic, N., Egyed, A., Gruenbacher, P., Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery, Proceedings of the Second International Requirements to Architecture Workshop (STRAW 03), Portland, Oregon, May 3-11, 2003.
20. Mikic-Rakic, M., Mehta, N. R. and Medvidovic, N., Architectural Style Requirements for Self-Healing Systems. 1st Workshop on Self-Healing Systems, Charleston, Nov. 2002.
21. Perry, D.E. and Wolf, A.L. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.
22. Portable Bookshelf website, <http://www.swag.uwaterloo.ca/pbs/>
23. SHriMP web site, <http://shrimp.cs.uvic.ca/>
24. Szyperski, C., "Component Software – Beyond Object-Oriented Programming", ACM Press, 1999.
25. Tran, J. and Holt., R., Forward and Reverse Architecture Repair. Proc. of CASCON '99, Toronto, pages 15–24, November 1999.
26. Tran, J., Godfrey, M., Lee, E. and Holt, R., Architecture repair of open source software, *Proc. of 2000 Intl. Workshop on Program Comprehension (IWPC-00)*, Limerick, Ireland.
27. Tzerpos, V. and Holt, R. C., A Hybrid Process for Recovering Software Architecture. In CASCON'96, Toronto, Nov. 1996.
28. Tu, Q. and Godfrey, M., An Integrated Approach for Studying Software Architectural Evolution, *Proc. of 2002 Intl. Workshop on Program Comprehension (IWPC-02)*, Paris, June 2002.
29. Welch, I. and Stroud, R., Adaptation of connectors in software architectures, Third International Workshop on Component-Oriented Programming, Brussels, Belgium, July 1998.