Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 127 (2005) 165–181
www.elsevier.com/locate/entcs

Evolution Through Architectural Reconciliation

Paris Avgeriou[1] Nicolas Guelfi[2] Gilles Perrouin[3]

Software Engineering Competence Center
Faculty of Science, Technology and Communication
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
Luxembourg-Kirchberg, Luxembourg

Abstract

One of the possible scenarios in a system evolution cycle, is to translate an emergent set of new requirements into software architecture design and subsequently to update the system implementation. In this paper, we argue that this form of forward engineering, even though addresses the new system requirements, tends to overlook the implementation constraints. An architect must also reverse-engineer the system, in order to make these constraints explicit. Thus, we propose an approach where we reconcile two architectural models, one that is forward-engineered from the requirements and another that is reverse-engineered from the implementation. The final reconciled model is optimally adapted to the emergent set of requirements and to the actual system implementation. The contribution of this paper is twofold: the application of architectural reconciliation in the context of software evolution and an approach to formalize both the specification and transformation of the architectural models. The architectural modeling is based upon the UML 2.0 standard, while the formalization approach is based on set theory and first-order logic.

Keywords: software architecture, architectural design, model transformation, architectural recovery, architectural reconciliation.

1 Introduction

It is nowadays well-established that evolution of software should not be taken lightly, as it may correspond up to 90% of the total lifecycle costs. As a re-

[1] Email: paris.avgeriou@uni.lu
[2] Email: nicolas.guelfi@uni.lu
[3] Email: gilles.perrouin@uni.lu

doi:10.1016/j.entcs.2004.08.042

sult, software evolution has emerged as a new and promising field of software engineering, which tackles the problems of software change and software maintenance [16,38]. Even though research on software evolution, has been taking place for more than thirty years, it is only recently that concrete results have started to appear and the research community began to grasp the significance of the field [15]. Over the past years, some of the most important advances include: the eight laws of evolution [14], a phenomenology of software evolution [4,12,16], theories and practices on formalizing software evolution [15,22], tools that analyze and depict evolution of systems [38,29,33].

Research in software evolution strives to answer either the 'how' or the 'what' and 'why' [15,16,27]. The 'how' concerns the methods, practices and tools for evolving a system, in particular for synchronizing three distinct entities: 1) a model of the real world that is constantly changing, e.g. a domain model for an e-business system, 2) a specification of the system, e.g. the system's software architecture, and 3) the system implementation. The 'what' and 'why' observe the phenomenon of software evolution, trying to identify its causes and its internal workings.

This paper deals with the 'how' of software evolution, and in specific with keeping the system implementation in synchronization with the real-world model through a system specification. The real-world model might potentially range from a full domain model or business model of the entire business system to a minimal requirements specification for the software system under development. However, for the purposes of this paper we are only interested in the software requirements specification and consider that requirements are indeed well-defined. As far as the system specification is concerned, it is a key factor in evolution, since it formalizes the description of the system to be built and bridges the abstract real-world domain model and the system implementation [15,27]. In our approach we adopt software architecture as a form of system specification, since it has been proposed as an ideal abstraction to support system evolution [2,10,11,32].

A typical scenario of architecture-based evolution is to design the system's architecture in order to address the new set of requirements, and then realize this architecture into the system implementation [2]. The problem with this forward engineering approach is that the architectural design takes into account only the set of requirements and not the existing system implementation. As a consequence, implementation constraints are overlooked in the architecture, which in turn cannot be properly implemented into code. Apparently the system implementation contains implicitly a large numbers of design decisions that are 'hidden' in the code and can potentially contradict the new set of design decisions that emerge in the architecture. Even if we do look at the system and try to identify implementation constraints, we will prob-

ably fail to identify everything unless we reverse-engineer the system. This paper proposes an approach to tackle this problem by applying architectural reconciliation. In particular, the architectural model that is used to develop the next system release, is derived from reconciling two architectural models, one that is forward engineered from the requirements specification and a second that is reverse-engineered from the system implementation. In this sense, the reconciled architecture will not only address the new requirements, but it will also take under consideration the implementation constraints. The documentation of software architectures is based on the UML 2.0 standard and formal methods. The rest of the paper is organized as follows: Section 2 provides the details of the proposed approach for evolution through architectural reconciliation, including the informal and the formal technique used for architectural description. Section 3 illustrates the application 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 some conclusions.

2 Architectural Reconciliation

2.1 The Process of Reconciliation

At the beginning of an evolution cycle we have an existing system implementation as well as a new set of requirements. The process of their reconciliation is comprised of three steps and is graphically illustrated in Figure 1.

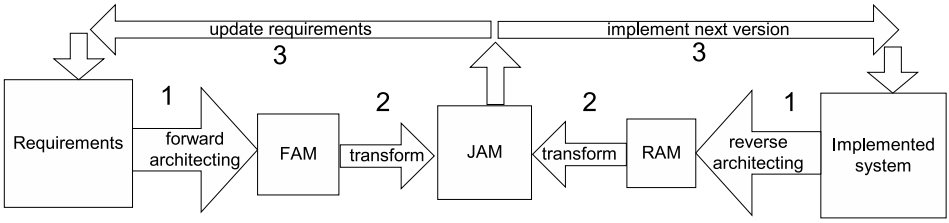

Fig. 1. Architectural Reconciliation for One Evolution Cycle

At the first phase we inspect the implementation code, and reverse-architect the implemented system in order to recover its architecture, which we name the **Reverse Architectural Model** or **RAM**. We do not prescribe a specific reverse-architecting approach, though there are a few such techniques and tools proposed, such as those in [9,23,29,33,37,38]. The RAM is mainly focused on identifying implementation constraints that may play a significant role dur-

ing the system evolution. These constraints are usually expressed in the form of natural language and accompany the UML diagrams or the formal models. At the same phase, we use the new Requirements specification to design the *ideal* architecture of the system, which we name the **Forward Architectural Model** or **FAM**. This forward-engineering design of the architecture can be performed by following any architecture-driven software development process. It is important to stress that the FAM derives clearly from the requirements side and strives to ideally satisfy the new requirements. The implementation constraints are not given much attention here, since they will be dealt with during the reconciliation in the next phase.

The second and most crucial phase is to bridge the RAM and the FAM into the **Joint Architectural Model** or **JAM**, which must satisfy both the new set of requirements and the implementation constraints. This is achieved by performing a transformation, that accepts the RAM and the FAM as inputs and produces the JAM as the output. In specific, the architect must go through the following steps:

(i) **Identify the implementation constraints that contradict the FAM.** These implementation constraints derive from the design decisions that the architect took during the previous evolution cycle and contradict the new design decisions taken in the FAM. The constraints may appear in any form, though we simply propose the use of natural language in combination with UML or formal models. This is because the software architecture community is still trying to tackle the problem of representing precisely local or global architectural constraints [20].

(ii) **Resolve the problems caused by the implementation constraints.** In order to resolve each such problem, the architect needs to decide between one, or a combination of the following actions:
   (a) Keep the part of the FAM and delete the part of the RAM that causes the problem.
   (b) Keep the part of the RAM and delete the part of the FAM that causes the problem.
   (c) Come up with a compromising solution that mixes both parts. In this case some of the elements from both models may be deleted, others may be retained, while more elements may be added.

(iii) **Complete the JAM.** The resolution of the problems 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 FAM and RAM elements that were affected by the reconciliation actions.

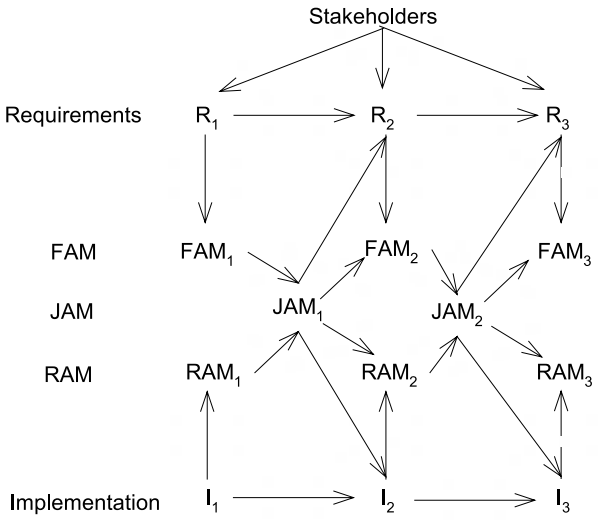

Fig. 2. Architectural Reconciliation over Several Evolution Cycles

The final phase in this process concerns with building the new version of the system according to the JAM, and also updating the requirements so as they reflect the reconciliation results. Figure 2 depicts the proposed approach over several evolution cycles, and also shows the rest of the details of the evolution process. It is of paramount importance to emphasize that there is always feedback from the JAM to the next evolution cycle. Firstly, as aforementioned, the JAM is not only used to develop the next implementation version of the system but it is also used to update the requirements according to the results of the reconciliation. Secondly, the JAM, as the only valid architectural model of the system, is used as the starting point for the FAM and RAM of the next iteration: FAM_i and RAM_i are designed starting from JAM_{i-1} and by considering the new set of requirements R_i and the implementation I_i respectively.

The next two subsections propose two different ways for describing the architectural models: informally in UML 2.0, and formally using set theory and first-order logic.

2.2 Description of the Architectural Models in UML

An architectural description is comprised of multiple views [6,10,11,13]. In order to reduce the complexity of reconciliating two complex multiple-view

architectural models, we have focused on the view that is considered to contain the most significant architectural information: the *component-and-connector* view [6]. 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. However the same theory of reconciliation can equally well apply to the rest of the views.

The documentation of software architectures has been performed with the aid of **Architecture Description Languages (ADLs)**, which aim at formally representing software architectures [21]. Unfortunately these languages have never been broadly used in the industry and most of them lack support by appropriate tools. However the recent trend is the use of the widely accepted Unified Modeling Language as an ADL, which has become the 'lingua franca' of software design. We have adopted this approach and we have been working on the emergent UML 2.0 standard [26], which claims to provide large support for modeling software architectures.
Our approach suggests therefore to model the component and connector view using UML 2.0 elements and especially those from the Composite Structures and Components packages. In specific the elements used for the component and connector view are:

(i) **Components**, which are specializations of classes and therefore have attributes and operations, but are also associated with provided and required interfaces. Components are also allowed to have an internal structure comprised of **properties** that in turn describe sets of instances of particular classes. Finally components may own ports that formalize their interactions points,

(ii) **Connectors**, which are either *assembly connectors* that connect the required interface of one component to the provided interface of a second, or *delegation connectors* that link the ports of a component to its internal parts,

(iii) **Interfaces**, which serve as contracts that components must comply with. An interface is either *provided* that describes a set of functionalities offered by a component, or *required* that describes a set of functionalities that a component expects from its environment.

(iv) **Ports**, which specify a distinct interaction point between the component that owns it and its environment or between the (behavior of the) component and its internal parts. Ports may specify required and provided interfaces of the component that owns them. A behavior port is a special case of a port, which sends all the incoming requests to the classifier that owns the port, rather than to its internal parts.

(v) **Classes**, that represent the constituents which realize the internal structure of components. These are not used in general-purpose class diagrams, but in composite structure diagrams, showing how the required and provided interfaces of a component delegate to or from its internal parts via the corresponding ports. Usually the composite structure diagrams do not contain the classes themselves, but sets of their instances in the form of properties.

Figure 3 illustrates the metamodel for the component-and-connector view, which is a subset of the UML 2.0 metamodel, and specifically the Components and Composite Structures packages. For reasons of simplicity some elements and other details have been omitted.

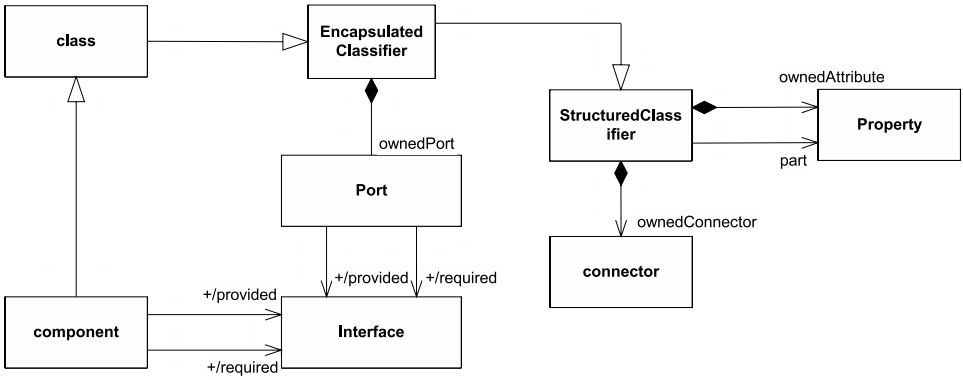

Fig. 3. UML 2.0 Metamodel for the Component and Connector View Elements (adapted from [26])

2.3 Formal Description of the Architectural Models

The syntax and semantics of UML are only semi-formally defined through a four layer meta-modeling architecture [25,26], that uses UML, natural language and OCL. The trend to formally specify the semantics of the language stems from the need to provide rigorous analysis and automatic transformations of UML models [17,34]. Different aspects of UML (e.g. class/component diagrams for modeling system structure, activity/statechart diagrams for modeling system behavior) require different forms of formalization. Process algebras [24] and Petri-Nets [1] allow for the formalization of system behavior while logic, sets and algebraic specifications [5,3,8] are used to formalize structural models. Graph grammars techniques [31] are also being used to formalize UML class and component diagrams [35,7].

We have concentrated on formalizing the component and connector view,

that was described in the previous section, and makes use of UML 2.0 structural models. In specific we have defined a formal metamodel using set theory and first order logic, that specifies the syntax of models in the component and connector view. This formal metamodel is based upon a similar work by Richters [30], that provides a non-ambiguous definition of an object model (classes, attributes, operations, associations and a generalization hierarchy) and defines the semantics of OCL. We have extended that metamodel in order to define the elements of the component-and-connector view, namely components, ports, connectors, interfaces and properties, and specify their composition. We therefore use this formal metamodel to formalize the reconciliation Re as a relation between three UML models satisfying a set of logical formulas φ that are inferred from the architect's tradeoff decisions: $Re = \{m, m = (FAM, RAM, JAM) | m \models \varphi\}$. The entire formal metamodel can't be included due to space restrictions, but a sample of these formulas are given both in logic and OCL in the next section.

3 A Case Study

The system that was chosen for this case study, is a popular open-source Learning Management System, named Ganesha [http://www.anemalab.org/ganesha], that 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 at will 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.

For illustrative purposes, this section focuses on the reconciliation of a particular component of this system, the chat component, which allows for basic chat functionality, such as sending and receiving messages, and viewing the list of connected users. The reverse-engineered architectural model of this component was designed, based on the JAM of the previous iteration and the existing system implementation. As shown on Figure 4, the chat component is implemented through a number of PHP files, communicating through connectors, which are merely calls between them. These components "hide" design decisions that may serve as implementation constraints during evolution. In particular:

(i) **index.php** creates the overall frameset consisting of frames for displaying the messages, information about connected users and GUI elements for entering and formatting messages.

(ii) **title.php** checks which browser is used and calls list.php with the correct browser details.

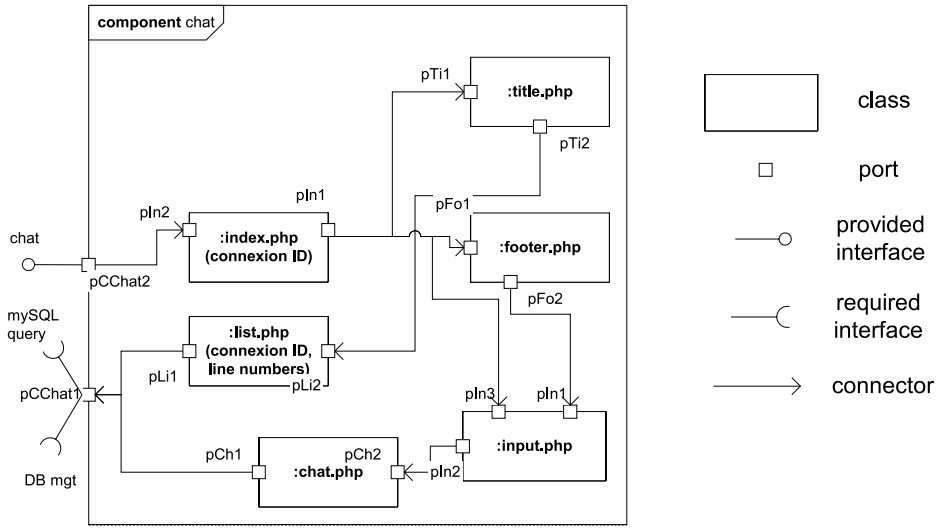

Fig. 4. The Reverse Architectural Model of the Chat Component

(iii) **footer.php** is the frame at the bottom of the frameset which displays all the GUI elements for changing font colors and style, and notifies input.php when such changes take place.

(iv) **list.php** updates the database for a new connected user and displays who has arrived and the set of currently connected members.

(v) **input.php** displays the GUI elements for entering messages, transforms these messages into the correct format according to footer.php and calls chat.php to display the messages.

(vi) **chat.php** is called by input.php if a message has been written in the text box, and displays that message along with information on who posted it. It also stores the messages to the database.

In the current evolution cycle, new requirements mandated the implementation of multiple chat rooms, that correspond to different topics of conversation, as well as the ability to exchange files between users. The architect therefore designed the FAM that satisfies these new requirements, starting from the JAM of the previous iteration. The forward architectural model, which is depicted on Figure 5, is naturally quite similar to the RAM, except for two new added components:

(i) **file.php** that implements the functionality for file management through

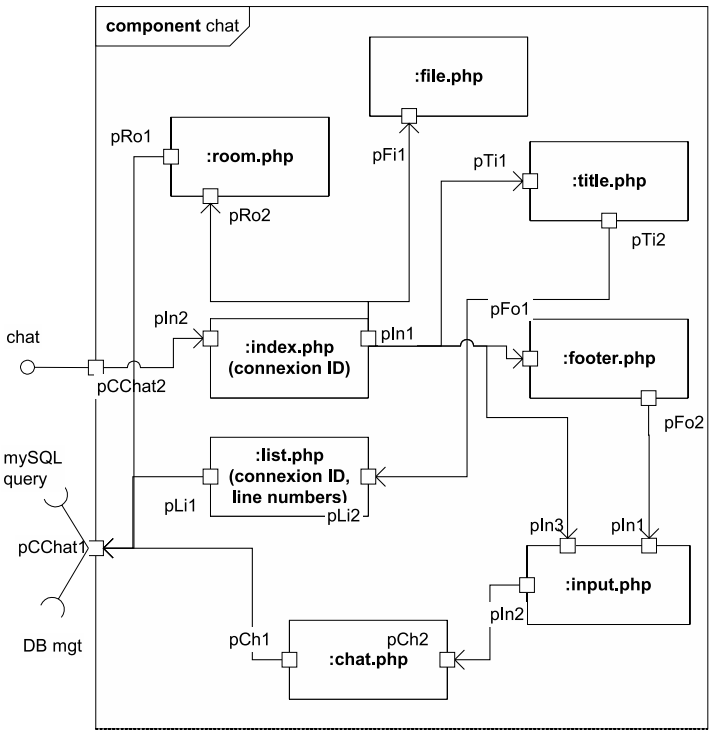

Fig. 5. The Forward Architectural Model of the Chat Component

an appropriate Graphical User Interface.

(ii) **room.php** that implements the functionality for changing rooms and displaying users' location. It is called by index.php and it also accesses the database to retrieve and store information about the room.

In order to perform the reconciliation we need to look at the two models, and try to resolve the potential problems, caused by implementation constraints. In this particular case, there were three implementation constraints that caused problems with the FAM and the architect needed to tradeoff on how to best resolve them. We describe the architect's decisions in three forms: natural language, logic formulas and OCL. For the OCL part we consider a model composed of the package "evol" with three sub-packages containing model elements from FAM, RAM and JAM.

The three implementation constraints and the corresponding reconciliation actions, which resulted to the joint model (Figure 6), are the following:

(i) The GUI for entering text already exists and is included in input.php. Ideally we would want to put the GUI for changing rooms in the same place. Therefore the code for GUI in room.php should be moved to input.php, which should forward this information to the former for implementing the application logic. So these two components are modified into input2.php and room2.php, while they are also connected through a connector for the information exchange.

Logic: $room.php \in class_{FAM}, input.php \in class_{RAM} \Rightarrow room2.php \wedge input2.php \in class_{JAM} \wedge \exists\ pIn2, pRo3 \in ports_{JAM} \wedge \exists c \in conns_{JAM}$ $with\ connends(c) = \{pIn2, pRo3\}$

OCL:

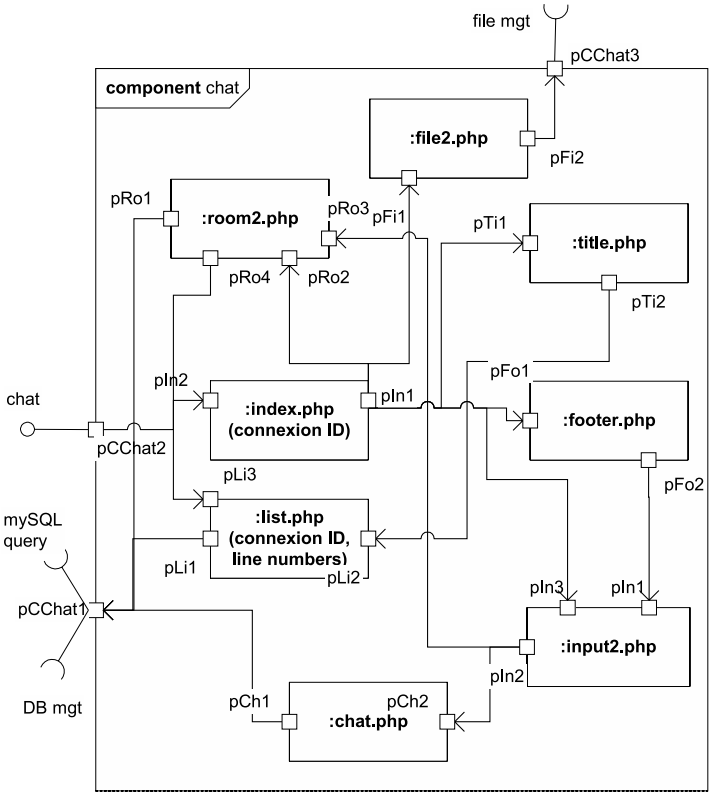

Fig. 6. The Joint Architectural Model of the Chat Component

```
package evol
context FAM::chat inv:
self.ownedMember->exists(c|c.oclIsTypeOf(Class)
and c.name='room.php') and
RAM::chat.ownedMember->exists(c|c.oclIsTypeOf(Class)
and c.name='input.php') implies
JAM::chat.ownedMember->exists(c1,c2|c1.oclIsTypeOf(Class)
and c2.oclIsTypeOf(Class) and c1.name='input2.php'
and c1.name='room2.php') and
JAM::chat.ownedMember->exists(c|c.oclIsKindOf(Connector)
and c.end.role->size()=2 and
c.end.role->includesAll(Set{pIn2,pRo3}))
endpackage
```

(ii) List.php displays the currently connected users in the right frame and that is where we also want to display information about what room the users are into. Thus room2.php needs to pass this information to list.php through a connector.

logic: $\exists\ pLi3, pRo4 \in ports_{JAM} \wedge \exists c \in conns_{JAM} \wedge$ $connends(c) = \{pLi3, pRo4\}$

OCL:

```
package evol
context JAM::chat inv:
self.ownedMember->exists(c|c.oclIsKindOf(Connector)
and c.end.role->size()=2 and
c.end.role->includesAll(Set{pLi3,pRo4}))
endpackage
```

(iii) There is already a component in Ganesha that provides file management such as uploading and downloading files. This is not visible in the RAM for the chat component but in the RAM for the entire system. In this sense, file.php should reuse this component through its file management interface rather than implement this functionality from scratch. Therefore this component is slightly modified into file2.php and requires interface *file mgt* to operate, which is added as a required interface of the chat component.

logic: $file.php \in class_{FAM} \Rightarrow file2.php \in class_{JAM} \wedge \exists\ pFi2, pCChat3 \in$ $ports_{JAM} \wedge \exists c \in conns_{JAM}, connends(c) = \{pFi2, pCChat3\} \wedge$ $\exists file\ mgt \in intfs_{JAM} \wedge file\ mgt \in requires(pCChat3)$

OCL:

```
package evol
context JAM::chat inv:
FAM::chat.ownedMember->exists(c|c.oclIsTypeOf(Class)
and c.name='file.php') implies
self.ownedMember->exists(c|c.oclIsTypeOf(Class)
and c.name='file2.php' and c.ownedPort->includes(pFi2))
and self.ownedPort->includes(pCChat3)
and
self.ownedMember->exists(c|c.oclIsKindOf(Connector)
and c.end.role->size()=2
and c.end.role->includesAll(Set{pFi2,pCChat3})) and
pCChat3.required->includes(file mgt)
endpackage
```

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 [28] 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 [36], 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 where the opposite takes place. Even though they have been mostly working with Open Source Software, where architectural drift is more likely to happen, they claim that their results can be generalized in commercial systems as well. They do not propose an approach for performing the design of the conceptual architecture but they do suggest tools such as those in [29,33] for reverse-architecting.

Roughly, the same problem has been dealt with in [18], 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 [19] that transforms the requirements into a handful of simple architectural elements that are 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 of the aforementioned approaches since we have adopted the intermediate steps of FAM and RAM that [18] proposes and we have devised actions for the architectural reconciliation that are similar to those for *forward* and *reverse repair* [36]. Our work extends these approaches in the sense that we provide formalisms for the definition of the architectural models and subsequently their transformations in order to derive the final model. This formal transformation-based process is only part of our philosophy which states that everything can be considered as transformations between different artifacts, as will be explained in the final section.

5 Conclusions

In this paper we have argued that the evolution of a system cannot be performed effectively in a forward engineering style, e.g. transforming new system requirements into architecture and then into code. The problem is that the existing system implementation may place significant constraints to the new requirements and therefore must be taken under account. These implicit implementation constraints cannot be made explicit unless they are reverse-engineered. 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 logic-based transformations. The added value of our approach concerns two issues: the adoption of architectural reconciliation in the context of software evolution in order to overcome the problems of forward engineering and the formalization of both the architectural models and the transformations required to perform the reconciliation.

This approach constitutes a representative part of our holistic view on model-based software engineering. In particular we believe that transformation is a key mechanism during the entire development lifecycle and everything can be placed in the context of transforming artifacts into other artifacts. These transformations can take place: a) at a different level of abstraction, e.g. from design to code: b) at the same level of abstraction, e.g. from an architectural model to another architectural model that refines the former. The transformations are of paramount importance since they provide an association between the artifacts created; they are the conceptual ‘glue’ that binds

everything together in a coherent set. This binding mechanism can provide the rationale for the decisions taken during the development process by tracing forward or backwards to the various artifacts.
We are currently elaborating other kinds of such transformations of the various artifacts, and we intend to end up with a fairly complete set of transformations that cover the entire process. We also intend to work on providing tool support for specifying and subsequently analyzing the formal models.

References

[1] Baresi, L., *Some preliminary hints on formalizing uml with object petri nets*, in: *Integrated Design and Process Technology*, 2002.

[2] Bosch, J., "Design and use of software architectures: adopting and evolving a product-line approach," ACM Press/Addison-Wesley Publishing Co., 2000.

[3] Breu, R., U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe and V. Thurner, *Towards a formalization of the unified modeling language*, Lecture Notes in Computer Science **1241** (1997).

[4] Buckley, J., T. Mens, M. Zenger, A. Rashid and G. Kniesel, *Towards a Taxonomy of Software Change*, Journal on Software Mainteance and Evolution: Research and Practice (2004).

[5] Clark, T. and A. Evans, *Foundations of the unified modeling language*, in: anonymous, editor, *2nd Northern Formal Methods Workshop* (1998).

[6] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, "Documenting Software Architectures: Views and Beyond," Addison-Wesley, 2002.

[7] Engels, G. and R. Heckel, *Graph transformation as unifying formal framework for system modeling and model evolution*, in: *European Conference on Software Maintenance and Reengineering (CSMR 2001), International Special Session on Formal Foundations of Software Evolution*, Lisbon, Portugal, 2001.

[8] Favre, L. M., *A formal mapping between UML static models and algebraic specifications*, in: A. Evans, R. France, A. Moreira and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Workshop of the pUML-Group, UML 2001, Toronto, Canada*, LNI **P-7** (2001), pp. 113–127.

[9] Guo, G. Y., J. M. Atlee and R. Kazman, *A software archtiecture reconstruction method*, in: *WICSA-1*, San Antonio, TX, USA, 2001.

[10] Hofmeister, C., R. Nord and D. Soni, "Applied software architecture," Addison-Wesley Longman Publishing Co., Inc., 2000.

[11] IEEE, *Recommended Practice for Architectural Description of Software Intensive Systems*, Technical Report IEEE-std-1471-2000, IEEE (2000).

[12] Kemerer, C. F. and S. Slaughter, *An empirical approach to studying software evolution*, IEEE Trans. Softw. Eng. **25** (1999), pp. 493–509, IEEE Press.

[13] Kruchten, P., *The 4+1 view model of architecture*, IEEE Softw. **12** (1995), pp. 42–50.

[14] Lehman, M. M., *Laws of software evolution revisited*, in: *Proceedings of the 5th European Workshop on Software Process Technology* (1996), pp. 108–124.

[15] Lehman, M. M. and J. F. Ramil, *An Approach to the Theory of Software Evolution*, in: *International Workshop on Principles of Software Evolution IWPSE 2001*, Vienna, Austria, 2001.

[16] Lehman, M. M. and J. F. Ramil, *Software Evolution: Background, Theory, Practice*, Information Processing Letters **88** (2003), pp. 33–44.

[17] McUmber, W. E. and B. H. C. Cheng, *A general framework for formalizing uml with formal languages*, in: *Proceedings of the 23rd international conference on Software engineering* (2001), pp. 433–442.

[18] Medvidovic, N., A. Egyed and P. Gruenbacher, *Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery*, in: *STRAW'03 : Second International SofTware Requirements to Architectures Workshop at ICSE'03*, Portland, OR, USA, 2003, pp. 61–68.

[19] Medvidovic, N., A. Egyed and P. Gruenbacher, *Reconciling software requirements and architectures with intermediate models*, Journal for Software and Systems Modeling (SoSyM) (2004).

[20] Medvidovic, N., D. S. Rosenblum, D. F. Redmiles and J. E. Robbins, *Modeling software architectures in the unified modeling language*, ACM Trans. Softw. Eng. Methodol. **11** (2002), pp. 2–57.

[21] Medvidovic, N. and R. N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Trans. Softw. Eng. **26** (2000), pp. 70–93.

[22] Mens, T. and M. Wermelinger, *Formal foundations of software evolution: workshop report*, SIGSOFT Softw. Eng. Notes **26** (2001), pp. 27–29, ACM Press.

[23] Mikic-Rakic, M., N. R. Mehta and N. Medvidovic, *Architectural style requirements for self-healing systems*, in: *1st Intl. Workshop on Self-Healing Systems*, Charleston, 2002.

[24] Ng, M. Y. and M. Butler, *Towards formalizing UML state diagrams in CSP*, in: A. Cerone and P. Lindsay, editors, *1st IEEE International Conference on Software Engineering and Formal Methods*, 2003, pp. 138–147.

[25] OMG, *UML 2.0 infrastructure specification*, Technical Report ptc/03-09-15, Object Management Group (2003).

[26] OMG, *UML 2.0 superstructure final adopted specification*, Technical Report ptc/03-08-02, Object Management Group (2003).

[27] Perry, D. E., *Dimensions of software evolution*, in: H. A. Müller and M. Georges, editors, *Proceedings of the International Conference on Software Maintenance* (Victoria, B.C., Canada; September 19-23, 1994), 1994, pp. 296–303.

[28] Perry, D. E. and A. L. Wolf, *Foundations for the study of software architecture*, ACM SIGSOFT Software Engineering Notes **17** (1992).

[29] Portable Bookshelf Website, http://www.swag.uwaterloo.ca/pbs/.

[30] Richters, M., "A Precise Approach to Validating UML Models and OCL Constraints," Ph.D. thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002).

[31] Rozenberg, G., "Handbook of graph grammars and computing by graph transformation: volume I. foundations," 98-102288-48, World Scientific Publishing Co., Inc., 1997.

[32] Shaw, M. and D. Garlan, "Software Architecture: Perpectives on an Emerging Discipline," Prentice Hall., 1996.

[33] SHriMP website, http://shrimp.cs.uvic.ca/.

[34] Smith, J., M. K. Kokar and K. Baclawsi, *Formal verification of UML diagrams: A first step towards code generation*, in: A. Evans, R. France, A. Moreira and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Workshop of the pUML-Group, UML 2001, Toronto, Canada*, LNI **P-7** (2001), pp. 224–240.

[35] Tenzer, J., *A formal semantics of UML class diagrams based on transformation systems*, Technical Report 2001/09, Technische Universität Berlin (2001), publication of Master Thesis.

[36] Tran, J. B., M. W. Godfrey, E. H. S. Lee and R. C. Holt, *Architecture repair of open source software*, in: 8^{th} *IEEE International Workshop on Program Comprehension (IWPC 2000)* (2000).

[37] Tran, J. B. and R. C. Holt, *Forward and reverse architecture repair*, in: *CASCON'99*, Toronto, 1999, pp. 15–24.

[38] Tu, Q. and M. W. Godfrey, *An Integrated Approach for Studying Architectural Evolution*, in: *Intl. Workshop on Program Comprehension (IPWC-02)*, Paris, France, 2002.