Information Sciences 259 (2014) 396–411

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Reusing design experiences to materialize software architectures into object-oriented designs

Germán Vazquez [a,b,*], J. Andres Diaz Pace [c], Marcelo Campo [a,b]

[a] ISISTAN Research Institute, Facultad de Cs. Exactas, UNCPBA Campus Universitario, Paraje Arroyo Seco, Tandil 7000, Argentina
[b] CONICET, Comisión Nacional de Investigaciones Científicas y Técnicas, Argentina
[c] Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Ave., Pittsburgh, PA 15232, USA

ARTICLE INFO

Article history:
Received 7 March 2008
Received in revised form 11 March 2010
Accepted 12 March 2010
Available online 20 March 2010

Keywords:
Architecture design
Object-oriented design
Architecture materialization
Software reuse
Case based reasoning

ABSTRACT

Software architectures capture early design decisions about a system in order to fulfill relevant quality attributes. When moving to detailed design levels, the same architecture can accept many different object-oriented implementations. A common problem here is the mismatches between the quality-attribute levels prescribed by the architecture and those realized by its object-oriented materialization. A significant step towards reducing those mismatches is the provision of tool support for assisting developers in the materialization of software architectures. Prerequisites to develop materialization tools are the organization of a body of design knowledge and the definition of quality-driven reasoning procedures. Since materialization activities are mainly driven by past developers' experiences, we propose a Case-based Reasoning (CBR) approach that, through the codification of design experiences, permits to establish links between software architecture structures and object-oriented counterparts. This approach is supported by an Eclipse-based tool, called SAME (Software Architecture Materialization Environment), which is a reuse-oriented assistant to the developer. SAME is able to recall and adapt successful architecture materializations for particular quality attributes, in order to help the developer to derive an appropriate object-oriented design for his/her architecture.

1. Introduction

The design of software systems is a very complex activity as it is strongly influenced by the quality required in the final products. Consequently, software designers are compelled to make the right design decisions in early development stages, in order to fulfill quality-attribute requirements such as: modifiability, performance, or security, among others [9]. Over the last years, the software community has paid great attention to software architectures as a key abstraction in the software design process. A software architecture enables the description of the high-level organization of a system independently from implementation issues, so as to capture key design decisions and allow stakeholders to reason about quality attributes [12].

Being abstract design models, software architectures happen to be usually realized by means of object-oriented designs. Given an architecture and a set of quality-attribute drivers as input, there is a variety of object-oriented designs that can be derived from that input. We refer to this mapping between architecture and object-oriented designs to as *object-oriented materialization of software architectures* [13]. Ideally, the quality attributes prescribed at the architectural level must be

* Corresponding author at: ISISTAN Research Institute, Facultad de Cs. Exactas, UNCPBA Campus Universitario, Paraje Arroyo Seco, Tandil 7000, Argentina.
E-mail addresses: gvazquez@exa.unicen.edu.ar (G. Vazquez), adiaz@sei.cmu.edu (J. Andres Diaz Pace), mcampo@exa.unicen.edu.ar (M. Campo).

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2010.03.013

preserved at the object-oriented level in the final design. That is, if the architecture prescribes a layered design as a mechanism for achieving modifiability, the object-oriented design should be implemented in such a way the modifiability concerns still hold. However, due to the differences in the abstractions used by architecture design and object-oriented design, mismatches between the architecture "as intended" and the architecture "as implemented" are common. In spite of the current body of design knowledge and technologies, we believe that research is still needed to clarify the relationships between the architecture and the object-oriented worlds. Furthermore, little progress has been made regarding tool support for materialization activities.

In practice, the quality of a software design depends on the developer's knowledge and experience. Knowledge in the form of design patterns is important here as they codify guidelines for deriving object-oriented designs [21]. However, a pattern-based approach still presents some problems. A comprehensive catalog of design patterns has not yet been developed. The common template for expressing patterns is not conceived to put the pattern's knowledge directly into a design tool. Besides, patterns do not explicitly link architectural knowledge to quality attributes, so the choice among candidate patterns for achieving a quality-attribute goal in an architecture materialization is not straightforward. Most organizations rely on "gurus" that know how to implement predefined architectures in object-oriented terms [31,8], and these people are able to adapt their object-oriented solutions to new situations. In this context, a tool approach able to integrate the experiences from both the pattern community and expert developers can be very valuable for managing both architectural and object-oriented knowledge. We envision a design assistant that, fed with appropriate design experiences, helps developers in the exploration of object-oriented design alternatives for a given architecture. In order to implement these ideas, we face two challenges. First, we need a way to codify developers' design experiences along with their quality drivers in a knowledge repository. Second, we need a process to operationalize that knowledge in order to make it applicable to architecture materializations with similar characteristics. Among the different AI reasoning mechanisms, the case-based reasoning (CBR) paradigm fits well with the memory-based process employed by expert developers, in which certain types of materialization problems tend to recur and in which similar problems have often similar solutions [19].

In this article, we describe a novel tool approach based on the CBR paradigm, in which materialization experiences carried out by developers to implement a previous architecture provide insights for the implementation of a new architecture with similar quality-attribute goals. The ideas behind this approach were initially outlined in a previous work [40]. In [40], we proposed the use of architectural connectors as the focal entities for structuring the materialization experiences. Thus, a materialization experience is a case that describes the design context around a connector (the problem), and also captures the object-oriented design devised to implement that context (the solution). The design context includes the quality-attribute goals that drive the materialization (e.g., performance, modifiability, etc.). In the present work, we elaborate on the algorithms used for retrieving, comparing and adapting connector-based experiences, and present a prototypical integration of these algorithms into a tool called SAME (**S**oftware **A**rchitecture **M**aterialization **E**xplorer). SAME is an Eclipse-based tool that implements the typical CBR process: retrieve, reuse, revise and retain [30], and it is able to propose alternative object-oriented designs to the user. The main contribution of this article is a knowledge-based automation perspective to bridge the gap between architecture and object-oriented design. We have also performed a case-study to test the quality of the object-oriented solutions explored by the tool, and we have compared the time spent by developers to derive materialization alternatives with and without the tool.

The rest of the paper is organized around 8 sections as follows. Section 2 introduces the concepts underlying the SAME approach using a blackboard design example. In Section 3, we focus on the representation of designers' experiences in terms of architectural connectors. Sections 4 and 5 discuss the algorithms used to compare and adapt connector-based experiences respectively. Section 6 describes the SAME tool that we developed to support the approach. Section 7 discusses preliminary experimental results and lessons learned. Section 8 analyses related work. Finally, Section 9 comments on lines of future research and rounds up the conclusions of the paper.

2. Architectural connectors as reusable elements

Existing approaches to the problem of architecture materialization have been either focused on solutions given by architectural styles [32,15] or based on object-oriented frameworks applicable to specific domains [17,25] (see Related Work). Unfortunately, architectures are often heterogeneous and it is difficult to consider a particular style as the predominant structure for them [1]. This suggests that the materialization process must consider the elements that compose the architecture rather than the architecture as a whole. In several software development projects [13], we have observed that an architecture materialization is a combination of fine-grained design elements that separately reify the components and connectors [14] of the base architecture. An architectural component is a computational entity with runtime presence (e.g., process, client, server, data repository), while an architectural connector is a communication path among components (e.g., information flow, access to shared data, procedure call). Moreover, we have found that the same set of object-oriented elements used to reify specific connectors occur recurrently across different domains. The concrete implementations of a connector may differ (e.g., the specific communication protocols in a socket, or the particular method being invoked in a remote procedure call) but the generic object-oriented structure that makes the connector work remains constant across system implementations. These observations led us to revisit our materialization approach and move its focus from architectural styles to architectural connectors [40].

The recognition of connectors as first-class citizens is one of the most important contributions of the software architecture community [33–36]. As mentioned before, connectors are the medium through which components interact with each other and allow components to perform their assigned responsibilities. A well-known set of connector types recurrently used in practice include: push/pull procedure call, push/pull abstract servers, producer–consumer connectors, event channel and observer connectors, brokers, SOAP connectors, semaphores, and transporters, to name some of them [18,34,35].

At the architectural level, quality is primarily achieved by an adequate distribution of the system responsibilities among components and by how these components interact in order to realize their responsibilities [9]. However, since software architectures do not usually make implementation commitments [12,7], design details are deferred to a later object-oriented design stage, which still affects the final quality of the system. For example, an architecture can describe the interaction between two components as "component *A* exchanges data with component *B* through connector *X*" but it will say nothing about the data type or the particular method for exchanging the information. These details are deferred to later object-design stages, which still affect the final quality of the system. At the object-oriented level, we may have different materializations for the inter-component mechanisms provided by connectors [26]. For example, some connectors defer the establishment of the identity of the components until runtime. This mechanism supports modifiability, integrability and testability, although at the cost of performance. Other connectors schedule component interactions based on priorities to promote performance. Others support the remote interaction among components, allowing a system to interoperate with other systems. In other words, we argue that the features of connectors serve us to choose among possible object-oriented structures for realizing component interactions, and therefore, are the main contributors to the materialization process.

2.1. A motivating example

Let's introduce an example to clarify the role that connectors play in the achievement of the final quality of an object-oriented materialization. Fig. 1 shows a component-and-connector (C&C) architecture of a chess game system based on a blackboard architectural pattern [11]. The architecture is depicted as a UML2 component diagram [5] where connectors are stereotyped with custom visualization. The system consists of three types of components. A *Board* component maintains the internal state of the game and notifies state changes to *KnowledgeSource* components. A *KnowledgeSource* component query and update the *Board* component. Finally, the *BoardControl* component coordinates the execution of the *KnowledgeSource* components. The key architecting principle for blackboard systems is that the knowledge sources work decoupled from each other in order to support modifiability. When implementing the system, we can still preserve, augment, or decrease the levels of modifiability of the system (and indirectly affect the system's performance), by choosing alternative mechanisms to materialize the interactions among the *Board*, *KnowledgeSource* and *BoardControl* components.

In the following, we describe two object-oriented materialization alternatives for the architecture in Fig. 1. First, let's suppose that we want to derive a design alternative having good response times. If so, the object-oriented design depicted in Fig. 2.a is a good starting point for implementing our system architecture. Note that the lack of synchronization mechanisms in the interactions among the classes and the small number of intermediaries between them makes the solution behave well in terms of performance. Nevertheless, the computation schema in class *BoardControl* is fixed, so the architecture cannot easily accommodate configuration changes. For instance, if we want to configure additional components to receive notifications from the *Board* component, we will have to make considerable changes in the source code.

Let's now assume that we have other quality-attribute preferences in mind, so we aim at deriving a materialization alternative that provides higher levels of modifiability than the previous alternative. We might decide to change the abstract ser-

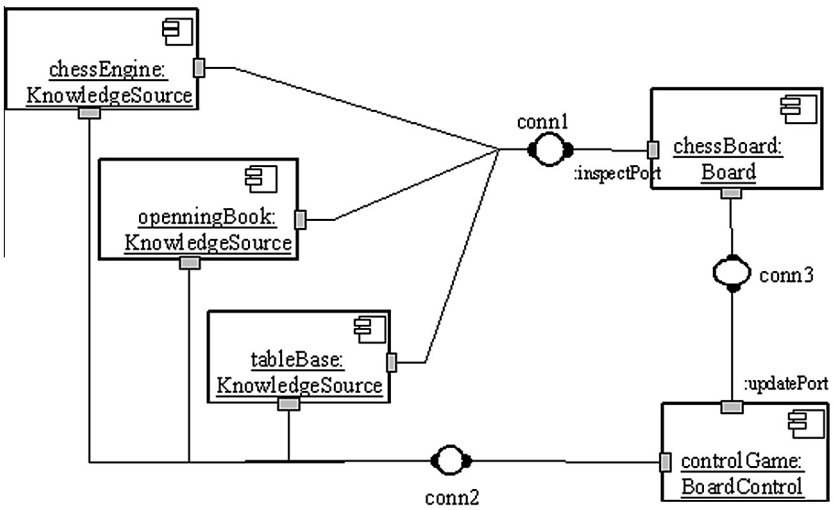

Fig. 1. A blackboard-based architecture for a chess game system.

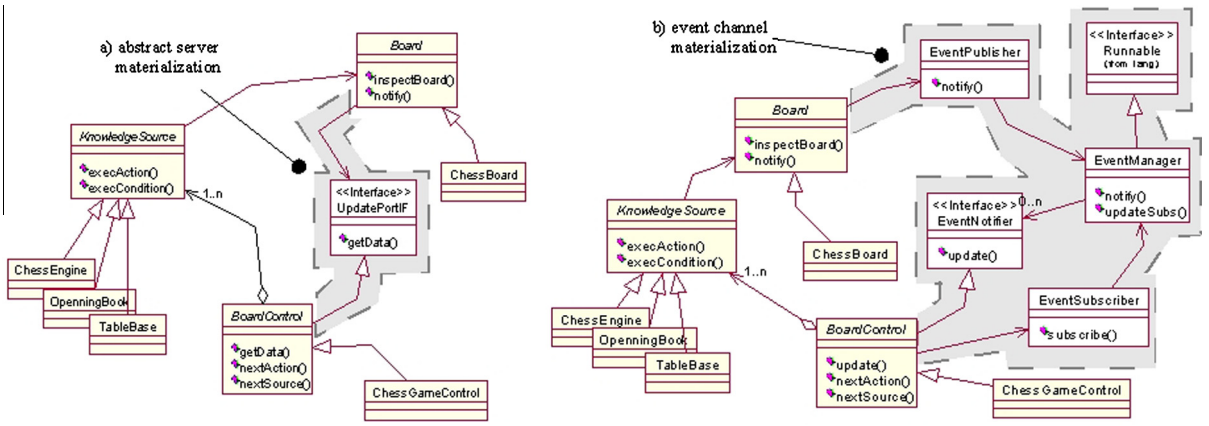

Fig. 2. Two alternatives that materialize the blackboard-based architecture.

ver mechanisms realizing the notifications of the *Board* component (in alternative 2.a) to an event channel mechanism, as depicted by the alternative in Fig. 2.b. In this new alternative, the *Board* class knows neither the number nor the identity of the objects receiving the event notifications. As a result, we have enhanced the system's ability to deal with future changes, because the derived design allows materializations of the *KnowledgeSource* components to evolve without affecting other elements. However, a main drawback of implementing the system with events is the potential performance degradation, due to the many intermediaries between the objects involved in the game interactions. In fact, we have little guarantees about the global performance of event-based systems because the order in which particular objects subscribe to events and receive notifications (e.g., *KnowledgeSource* objects being notified by the *BoardControl*) is not determined until runtime. In summary, this example shows how two implementations of the same connector lead the developer to a design tradeoff between modifiability and performance.

The reader is referred to [11,15] for other examples of multiple materialization alternatives for a given architecture. The references discuss different object-oriented materializations of a pipe-and-filter system, and provide a brief analysis of the quality-attribute consequences of each solution.

3. Representation of design experiences

In practice, developers face situations in which they have to choose between different alternatives in order to achieve specific design goals. Expert developers tend to use proven solutions to solve recurrent design problems. The catalogs of design patterns provide well-known solutions to recurrent design problems in particular contexts [21]. Although in some cases design patterns are useful to derive materializations of an architecture, the patterns actually deal with problems at the object-oriented design level, paying not explicit attention to architectural aspects and quality-attribute goals. Therefore, we think it is necessary to broaden the pattern perspective to consider also information about architecture-to-object mappings. Additionally, we need to operationalize the way expert developers use their own experiences in new problems, so that a novice developer (eventually assisted by a computer agent) can take advantage of that knowledge when exploring design solutions for materialization problems.

The CBR paradigm supports the experience-based design process mentioned above and gives us the foundations to describe design problems and reuse their solutions when solving other similar (but not exactly the same) problems [29]. In this context, we propose to codify design experiences into cases that we call *materialization experiences*. A materialization experience is a representation of a piece of design knowledge, tied to a particular materialization situation (the problem), which makes it explicit how an architecture fragment was mapped to an object-oriented fragment (the solution). Particularly, the characterization of an architectural connector and the context around that connector (i.e., the components attached to the connector) is the basis to discriminate among a variety of object-oriented implementation alternatives and, therefore, to explicitly achieve the quality-attribute levels pursued by the system.

Materialization experiences work in two ways within the surroundings of a connector. On one hand, the problem part of an experience describes the context in which a connector and its attached components are reified. This description comprises features such as: the desired quality-attribute levels, the interaction services provided by the connector, and the architectural topology around that connector. On the other hand, the solution part of an experience specifies the object-oriented skeleton to develop the materialization of the connector and its interacting components. This object-oriented skeleton is captured in terms of a collection of generic roles, which can be instantiated into concrete object-oriented elements reifying together the input architectural elements. The use of generic roles is what allows the object-oriented solutions of an experience to be reused and adapted to new similar materialization problems, as we will show in Section 5.

Fig. 3 depicts the structure of two materialization experiences to implement the interactions between the *Board* and *BoardControl* components in our blackboard-based architecture. As it can be seen in Fig. 3, the performance level specified by the quantitative values of the *Abstract Server* experience (on the left) is greater than the performance level of the *Event Channel* experience (on the right). Nonetheless, the response to future changes (i.e., modifiability) in the *Abstract Server* experience is worse than in the *Event Channel* experience. Also note that the *Abstract Server* experience identifies the participants in its connector interaction at design time (*identification binding*), whereas the *Event Channel* experience defers the identification of its connector's participants to runtime. Another characteristic in which both experiences differ is the cardinality of their respective connectors. As a result, the object-oriented solution provided by each materialization experience is developed in agreement with all these specific characteristics.

Our main goal is to reuse the knowledge applied in a proven experience to solve a new materialization problem. That knowledge should be modeled in such a way we can evaluate the extent to which experiences resemble each other. To this end, we have developed a framework for comparing materialization problems based on three dimensions: quality, feature and structure dimensions. The remaining of this section describes those dimensions, and the following section explains how they are used to measure similarity between two materialization experiences.

3.1. The quality dimension

The *quality dimension* of an experience specifies the quality-attribute levels achieved by the materialization of the interaction services provided by the connector. The quality level for each quality attribute is expressed by a numerical scale in the range [0–1]. In this scale 0 represents no satisfaction of a quality attribute, 1 represents full satisfaction of a quality attribute, and values between 0 to 1 represent intermediate levels of quality satisfaction.

3.2. The feature dimension

The feature dimension comprises what we call *interaction features* that characterize experiences in terms of the inherent properties shown by connectors. Particularly, the interaction features specify how the connector' interactions are mapped into an object-oriented design. For example, some connectors block components as they interact one with the other, or some other connectors defer the binding among interacting components until runtime. The way in which connectors' interactions are materialized contributes to promote or hinder the quality-attribute levels of the implementation of a system architecture [26,1,36]. In the following, we describe some interaction features that characterize architectural connectors (the list is not exhaustive, and other features can be defined as we get more insights into the architecture-to-object mappings):

- Blocking: This feature specifies whether components are suspended while transferring data and/or while transferring control to other components;
- Identification binding: This feature indicates the time at which sources and destinations (i.e., components attached to a given connector) are bound together. The bindings can happen at design time, at configuration time, at runtime, etc. The earlier in time the connection is established, the less the cost to retain the connection and to access target components;
- Cardinality: This feature specifies the number of component instances that can be attached to a connector interface. For instance, connectors with multiplicity greater than 1 often derive into materializations with a good response to future changes;
- Scope: This feature identifies the largest address space that a connector allows components to interact with each other. For instance, remote connectors usually favor system integrability.

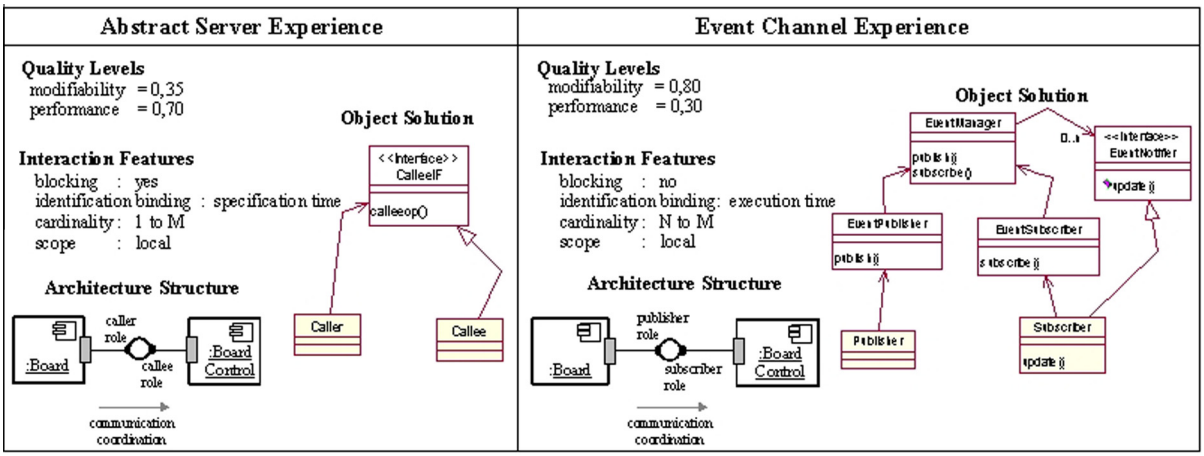

Fig. 3. Two sample materialization experiences.

Table 1
Some typical connectors and their interaction features.

	Blocking	Identification binding	Cardinality	Scope
Procedure call	Yes	compile-time	1 to 1	local
Abstract server	Yes	compile-time	1 to M	local
Remote procedure call	Yes	compile -time	1 to 1	remote
Producer–consumer	No	execution-time	N to M	local
Event channel	No	execution-time	N to M	local
Observer	Yes	execution-time	1 to M	local

Table 1 shows some features of widely-used connectors, namely: procedure call, remote procedure call (RPC), abstract servers, producer–consumer, event channel and observer connectors. Of course, we can define other types of connectors as we incorporate new experiences into our base of knowledge. Furthermore, we can define variants of known connectors with small differences in their object-oriented materialization. These variants will be characterized by the corresponding interaction features. For instance, we can specify a variation of the event channel connector (shown in Table 1) implemented with a blocking event publication mechanism or with a priority scheduler to process events.

3.3. The structure dimension

The structure dimension specifies the configuration of the components attached to the connector for a given materialization experience. Connectors have interfaces called *interface roles* that identify components as participants of the interaction. A component plays a particular role within the context of a connector depending on the interface the component is attached to. For instance, the *Board* component plays the *caller* role in the *Abstract Server* experience shown in Fig. 3; and the same component plays the *publisher* role in the *Event Channel* experience.

Additionally, connector interfaces are characterized by properties that constraint the data flow among components and/or constraint the control flow moving through the connector implementation. The properties of a connector interface express the invariant characteristics of the mechanisms that enable components to interact with each other [37]. Typical interaction services provided by connectors include: (i) communication services that specify how information moves around the system; (ii) coordination services that specify how control is passed among components; and (iii) arbitration services that specify how components synchronize with each other [33].

The interaction services provided by connectors serve as basis to evaluate whether connectors are compatible. For instance, a connector providing communication services can be substituted by another connector providing communication and coordination services, but the other way around is not true. In our approach, the retrieval of materialization experiences uses information about interaction services as indexes into the repository of experiences, in order to select candidate experiences for a materialization problem.

4. Assessing similarity of design experiences

Our automation of the materialization process is based on the typical CBR problem-solving metaphor [28]. Basically, the process comprises two key activities: (i) retrieval of relevant experiences from a repository of design knowledge, and (ii) adaptation of those experiences to fit the new problem specification so that their object-oriented solution can be composed to derive a single object-oriented materialization alternative.

The retrieval activity is performed in two steps. First, we look for compatible candidates and then choose the most promising candidate that maximizes a similarity metric. This step retrieves from the repository those experiences that are potentially useful to solve the target materialization problem. Second, we rank the retrieved candidates with respect to their similarity to the current problem. Before we can compute the overall similarity between two experiences, we evaluate the similarity between their attribute, feature and structure dimensions.

The algorithm to compare the quality dimensions of two experiences computes the Euclidean distance between the quality attributes from the assessed dimensions. Eq. (1) show the comparison criterion for the quality dimensions of experiences SOURCE and TARGET:

$$QUALITY\_SIMILARITY\ (TARGET, SOURCE) = 1 - \frac{\sum_{qa \in QA} DISTANCE\ (qa_{TARGET}, qa_{SOURCE})}{\#QA} \quad (1)$$

where *QA* represents the set of quality attributes of the cases and the *DISTANCE* (qa_{TARGET}, qa_{SOURCE}) function takes the Euclidean distance across the quality-attribute levels of the two cases. The quality assessment algorithm returns a value in the range [0–1], where 1 represents experiences with exactly the same quality levels and 0 represents experiences with completely dissimilar quality levels.

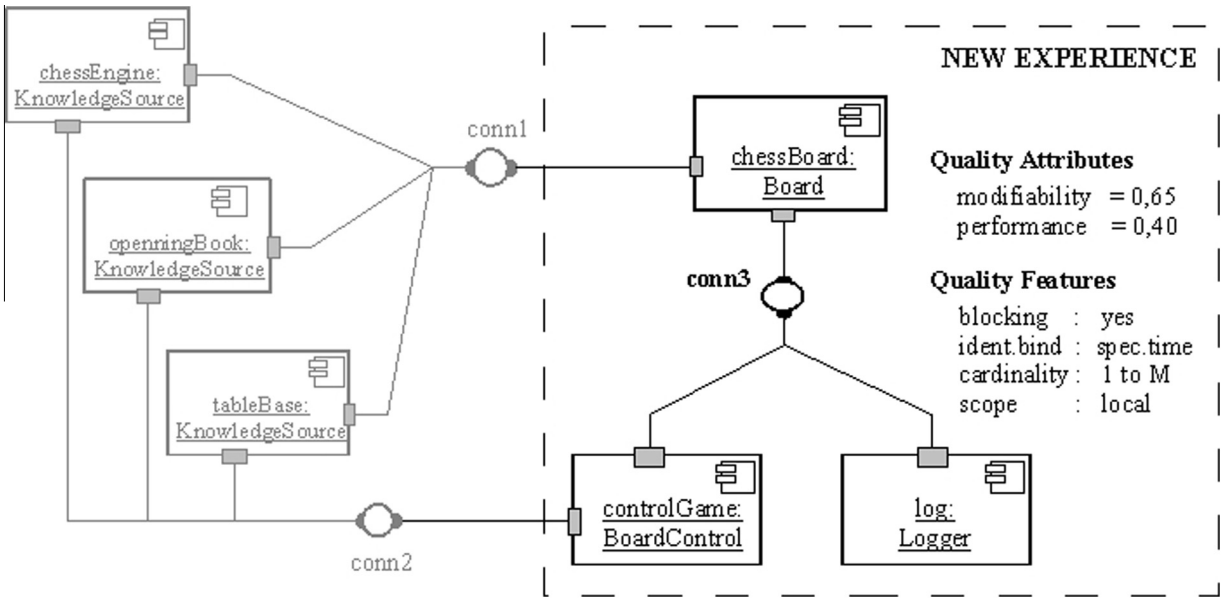

Fig. 4. A new system architecture to be materialized.

Coming back to our example from Section 2, let's consider a new architecture similar to the one shown in Fig. 1. A partial view of this new architecture is depicted in Fig. 4. A new component called *Logger* was added to the connector, so as to facilitate notifications of changes in the state of the *Board* component. The *Logger* component is in charge of keeping record of those notifications.

We will focus on the connector labeled *conn3* as a materialization experience that has not been yet solved. To solve this experience, let's assume that we have at hand the materialization experiences shown in Fig. 3. To evaluate which of those previous experiences is more appropriate to solve the new problem, we assess their similarity regarding the new target experience. In the following, we compute the quality similarity between the new target experience (NEW) regarding the *Abstract Server* (AS) and *Event Channel* (EVT) experiences:

$$QUALITY\_SIMILARITY\ (\text{New}, AS) = 1.0 - (\text{DISTANCE}\ (\text{New}_{\text{modifiability}}, \text{AS}_{\text{modifiability}}) + \text{DISTANCE}\ (\text{New}_{\text{performance}.}, \text{AS}_{\text{performance}}))/2 = 1.0 - (\text{DISTANCE}\ (0.65, 0.35) + \text{DISTANCE}\ (0.40, 0.60))/2 = 1.0 - ((0.25 + 0.15)/2) = 1.0 - (0.40/2)) = 1.0 - 0.20 = \mathbf{0.70}$$

$$QUALITY\_SIMILARITY\ (\text{New}, EVT) = 1.0 - (\text{DISTANCE}\ (\text{New}_{\text{modifiabiity}}, \text{EVT}_{\text{modifiability}}) + \text{DISTANCE}\ (\text{New}_{\text{performance}.}, \text{EVT}_{\text{performance}}))/2 = 1.0 - (\text{DISTANCE}\ (0.65, 0.80) + \text{DISTANCE}\ (0.40, 0.30))/2 = 1.0 - ((0.15 + 0.10)/2) = 1.0 - (0.25/2)) = 1.0 - .12 = \mathbf{0.88}.$$

We can see that the *Event Channel* experience returns a quality level of (0.88) that is a closer match to the new problem than that of the *Abstract Server* experience (0.70).

The algorithm for computing the similarity between the feature dimensions of two materialization experiences proceeds by iterating through the pairs of interaction features from the assessed experiences and evaluating whether their values match each other. Eq. (2) represents the algorithm for assessing feature dimension similarity as the ratio between the number of matching features and the number of evaluated features:

$$FEATURE\_SIMILARITY\ (TARGET, SOURCE) = \frac{\sum_{if \in IF} MATCHING\_FEATURES\ (if_{TARGET}, if_{SOURCE})}{\#IF} \tag{2}$$

where *IF* represents the set of interaction features of the experiences, and MATCHING_FEATURES (if_{TARGET}, if_{SOURCE}) returns 1 if the value of the interaction feature on the target experience matches the value of the same feature in the source experience, otherwise it returns 0.

In the following, we compute the interaction feature similarity between the two sample experiences and our new target experience:

FEATURE_SIMILARITY (New,*AS*) = (MATCHING_FEATURES ($New_{blocking}$, $AS_{blocking}$) + MATCHING_FEATURES ($New_{ident.bind}$, $AS_{ent.bind}$)+ MATCHING_FEATURES ($New_{cardinality}$, $AS_{cardinality}$)+ MATCHING_FEATURES (New_{scope}, AS_{scope}))/4 = (MATCHING_FEATURES ("yes", "yes") + MATCHING_FEATURES ("spec_time", "spec_time") + MATCHING_FEATURES ("1toM", "1toM")+ MATCHING_FEATURES ("local", "local"))/4 = (1 + 1 + 0 + 1)/4 = **0.75**

FEATURE_SIMILARITY (New,*EVT*) = (MATCHING_FEATURES ($New_{blocking}$, $EVT_{blocking}$) + MATCHING_FEATURES ($New_{ident.bind}$, $EVTi_{dent.bind}$) + MATCHING_FEATURES ($New_{cardinality}$, $EVT_{cardinality}$) + MATCHING_FEATURES (New_{scope}, EVT_{scope}))/4 = (MATCHING_FEATURES ("yes", "yes") + MATCHING_FEATURES (spec_time", "exec_time") + MATCHING_FEATURES ("1toM", "NtoM")+ MATCHING_FEATURES ("local", "local"))/4 = (1 + 0 + 0 + 1)/4 = **0.50**

In this computation, the *Abstract Server* experience is a closer to the target experience since they have more matching features than the *Event Channel* experience.

Finally, the similarity between the structure dimensions of two materialization experiences is computed by evaluating the ratio between the number of matching components from the assessed experiences and the average numbers of components in the experiences. Eq. (3) shows the algorithm to compute the similarity between the structure dimensions of two materialization experiences:

$$STRUCTURE\_SIMILARITY\ (TARGET, SOURCE) = \frac{\#MATCHING\_COMPONENTS\ (TARGET, SOURCE)}{(\#COMPONENTS_{TARGET} + \#COMPONENTS_{SOURCE})/2} \tag{3}$$

where the term *#MATCHING_COMPONENTS (TARGET, SOURCE)* returns the number of components attached to the connector in one experience that matches a component in the other experience[1], and the terms $\#COMPS_{TARGET}$ and $\#COMPS_{SOURCE}$ represent the number of components in the *TARGET* and *SOURCE* experiences, respectively. To normalize the value of the structure similarity, the equation divides the number of matching components by the average number of components specified by each experience (this is the reason of the division by 2 in the equation).

The architectural structures of our two previous experiences are identical, so their structural similarity with respect to our new experience is also identical. On the one hand, the experiences have two matching components: the *Board* and *Board-Control* components. On the other hand, the *Logger* component in the new experience does not match with any element in our past experiences. Thus, the structure similarity is computed by the ratio between the matching components in the assessed experiences (2) and the average number of components (2.5), that is: *STRUCTURE_SIMILARITY*(*New*, *AS*|*EVT*)= 2/2.5 = **0.80**.

The three Eqs. (1)–(3) are normalized to yield values in the range [0–1], where 0 represents entirely dissimilar experiences and 1 represents identical experiences. After the computation of each of these components, we use a weighted sum to evaluate the global similarity between the target experience and our past experiences. Weights represent the relative importance of each measured dimension into the overall similarity assessment. The following equation expresses the global similarity between two assessed experiences:

$$GLOBAL\_SIMILARITY\ (TARGET, SOURCE) = \omega_1 * QUALITY\_SIMILARITY + \omega_2 * FEATURE\_SIMILARITY + \omega_3 * STRUCTURE\_SIMILARIY \tag{4}$$

The experiences in the repository are ranked, and the experience that maximizes Eq. (4) is chosen for reuse. We have defined a set of predetermined weights: ω_1 = 0.40; ω_2 = 0.30; ω_3 = 0.30, but these weights are not fixed in our CBR engine. The developer can modify them in order to vary the results of the similarity algorithm and obtain, therefore, a different ranking of experiences to work with. For instance, let's suppose that the developer wants to derive an optimal object-oriented materialization in terms of quality levels. Then, she should increase the ω_1 weight and decrease ω_2 or ω_3 accordingly. This way, the engine will retrieve and rank better those experiences that are closer to the target experience in terms of quality levels (regardless their structure and feature similarity). Alternatively, the developer might want to make the engine match the experiences that are more similar in structure to a target experience. If so, she should increase the ω_2 weight.

To conclude the example, we compute the experiences' global similarity using the values determined previously:

GLOBAL_SIMILARITY (New, *AS*) = 0.40 * QUALITY_SIMILARITY (New, AS) + 0.20 * FEATURE_SIMILARITY (New,*AS*) + 0.40* STRUCTURE_SIMILARITY (New,*AS*) = 0.40 * 0.70 + 0.20 * 0.75 + 0.40 * 0.80 = **0.75**

[1] Two components are a match if they belong to the same type and if they play compatible roles in the connector they are attached to.

$$GLOBAL\_SIMILARITY\ (\text{New}, EVT) = 0.40 * \text{QUALITY\_SIMILARITY}\ (\text{New}, \text{AS}) + 0.20 * \text{FEATURE\_SIMILARITY}\ (\text{New}, \text{AS}) + 0.40 * \text{STRUCTURE\_SIMILARITY}\ (\text{New}, \text{AS}) = 0.40 * 0.88 + 0.20 * 0.50 + 0.40 * 0.80 = \mathbf{0.77}$$

As we can see, the *Event Channel* experience (0.77) is more similar to our new target experience than the *Abstract Server* experience (0.75). Consequently, the first experience is chosen for the next stage of the materialization process in which the selected experience is adapted so that it can be reused to solve the new problem.

5. Adapting design experiences

The object-oriented solutions of materialization experiences are intended be potentially reused in arbitrary configurations of architectural elements. To do so, we codify the object-oriented solution into a generic template that we refer to as the *interaction model*. An interaction model defines a collection of generic roles plus a set of constraints on those roles. These roles and constraints embody the generic structure that reifies the interaction mechanisms provided by a connector within a materialization experience. The idea of generic roles, inspired by work developed elsewhere [23,24], makes it feasible to reuse object-oriented solutions into new materialization problems.

Interaction models are agnostic with respect to the number and types of the components attached to a connector. However, we still need a mechanism to bridge together the individual materialization of each component attached to the connector. This is done by means of a special kind of roles called *adaptation points*. An adaptation point acts as a placeholder from which the object-oriented elements that implement the components are hooked into the elements that implement the interaction services provided by the connector.

In Fig. 5, we use a "pseudo" UML notation to represent the interaction model of an *Event Channel* experience. The notation intends to highlight the relationships among regular roles and adaptation points. The type of the nodes in the figure is marked by a stereotype. The edges denote relationships between the roles. The model defines two adaptation points: *PublisherComp* and *SubscriberComp* (in general, an interaction model can define more than one adaptation point per connector interface). On one hand, the *PublisherComp* adaptation point ensures that the materialization of any component attached to the publisher interface of an event connector will be able to publish event notifications. This publication is implemented by means of the *notify* operation and the association to the *EventPublisher* class role shown in the figure. The *SubcriberComp* adaptation point, on the other hand, guarantees that the materialization of any component attached to the subscriber interface of event connectors will be able to subscribe itself to the event mechanism and, hence, to receive event notifications. This reception is implemented by means of the *update* operation, the association to the *EventSubscriber* class role, and the implementation of the *EventNotifier* interface.

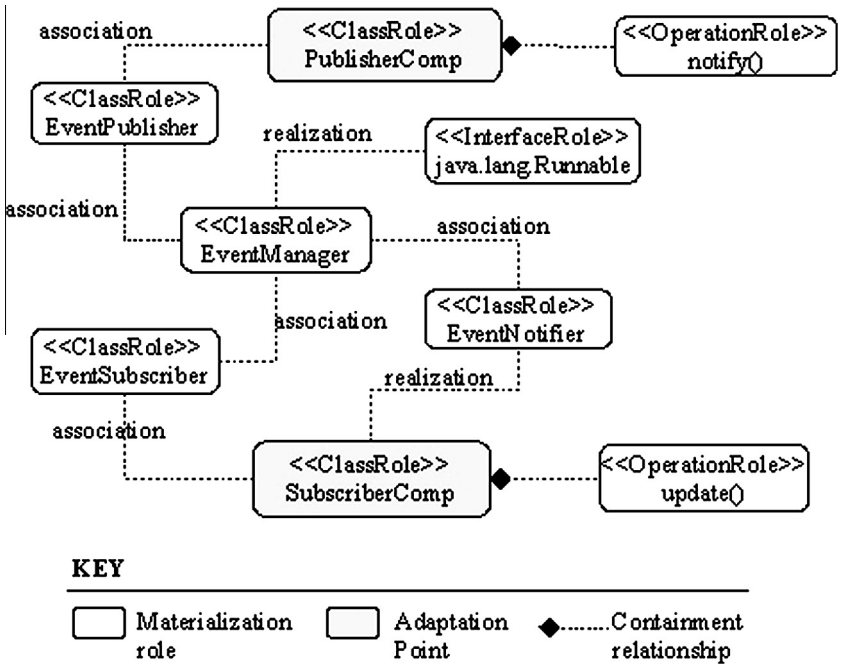

Fig. 5. The Event Channel interaction model.

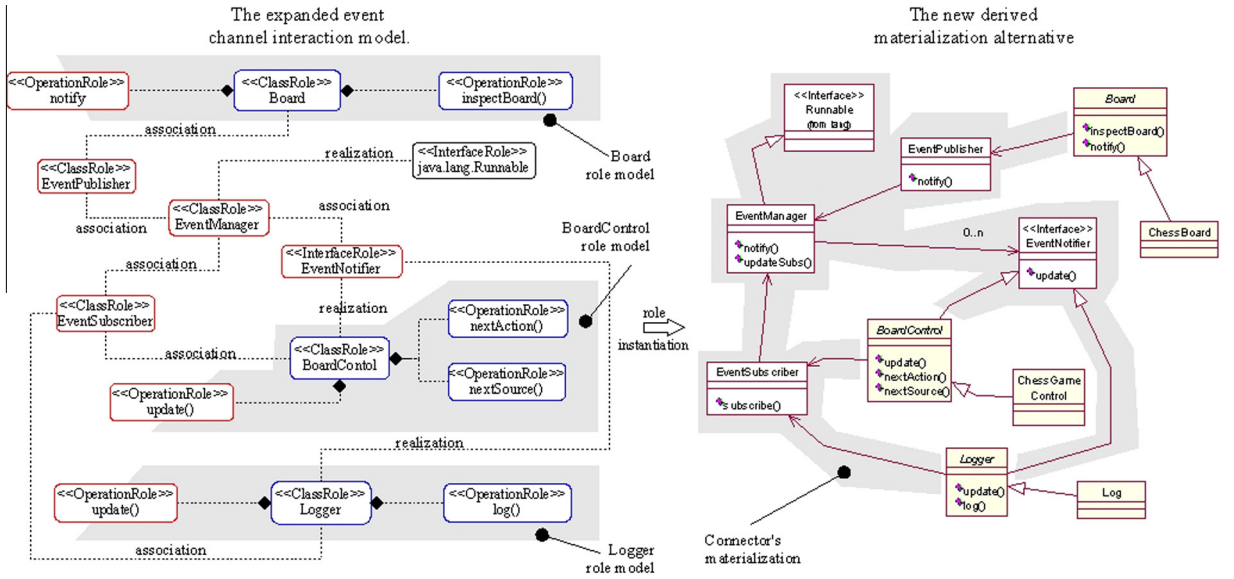

Fig. 6. The adaptation of the Event Channel interaction model.

The generic nature of interaction models, defined in terms of adaptation points, makes each model an abstract, and incomplete solution. In this context, adaptation points need to be expanded into regular roles according to a current configuration of components around a target connector. This expansion results into a new model that unifies the roles reifying the connector's interaction services and the roles reifying the involved components. The new model is entirely defined in terms of regular roles that, when properly instantiated, derive the actual object-oriented design that materializes the problem at hand. This procedure provides guidance for adapting cases (both what to adapt and how to do it) using previous experiences, which in the CBR jargon is called derivational adaptation [28].

To continue with our running example, let's see how the adaptation procedure expands the adaptation points of the *Event Channel* interaction model and generates the new model as a possible materialization design for our input architecture. Initially, the procedure pulls from the repository of design knowledge all the roles that are used to reify the components currently linked to the adaptation points in the source interaction model[2]. Afterwards, the procedure iterates through every adaptation point in the source model and injects the roles owned by each point (e.g., methods or attributes) into the roles used to materialize the associated components. The left side of Fig. 6 depicts the *Event Channel* interaction model with its adaptation points expanded according to the target architectural configuration shown in Fig. 4. We can see, for example, that the *Board* class role contains the *inspectBoard* and *notify* method roles. The former method is part of the original materialization of the *Board* component. The *notify* method is part of the materialization of the *Event Channel* interaction model and, therefore, it was injected in the *Board* class by the adaptation procedure. Finally, the procedure replicates all the relationships that have an adaptation point as a source or destination, and it connects the materialization of the connector with the materialization of its attached components. In our example, the realization relationship shown in the interaction model of Fig. 5 that goes from the *SubscriberComp* adaptation point to the *EventNotifier* class role is replicated into two relationships, as show in Fig. 6. In the resulting design, the *BoardControl* and *Logger* class roles both implement the interface role *EventNotifier*.

To summarize, the adaptation procedure instantiates the expanded model by binding each generic role to a concrete object-oriented element that helps to materialize the target architectural context. The right side of Fig. 6 shows the object-oriented model resulting from the instantiation phase. A generic role can be instantiated either by binding it to a newly created object-oriented element or by selecting an existing element in the derived object-oriented design. As it can be seen in Fig. 6, the constraints enforced by the interaction model of the *Event Channel* experience are satisfied in the derived object-oriented design, so that the materializations of the involved components are able to interact one with the other.

The adaptation procedure and the similarity assessment algorithm are the core functionality of the SAME tool that we have developed to support our approach.

6. Overview of the SAME tool

SAME is based on the CBR problem-solving metaphor in which materialization experiences are stored in a repository of knowledge. This repository provides developers with a corpus of design solutions that are suitable to reify new similar archi-

[2] For each component, the repository of design knowledge stores a collaboration of roles that is used as basis to derive the component's materialization.

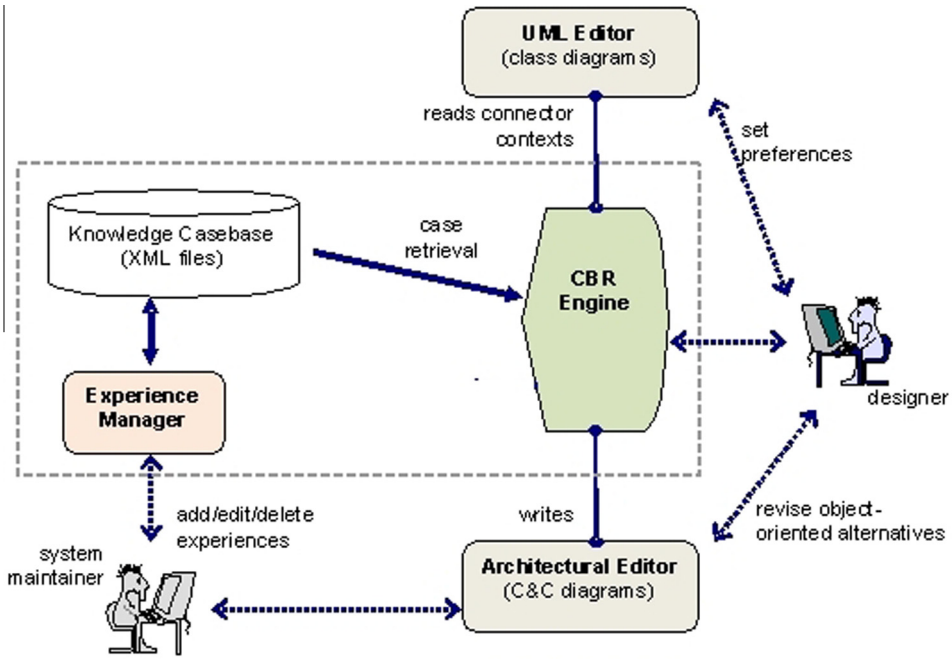

Fig. 7. Conceptual view of the SAME tool.

tectural contexts. The main goals of SAME are to manage the knowledge repository and to provide the reasoning procedures for retrieving and adapting materialization experiences.

Fig. 7 shows a conceptual view of the SAME tool that we have implemented in Eclipse. It comprises five main components: the architectural editor, the UML editor, the repository of experiences, the CBR engine and the experiences manager. The figure also shows two different types of users: software designers and system managers. A software designer defines and characterizes an input architecture, and then manipulates the CBR engine to derive an object-oriented materialization alternative. A system maintainer is responsible for keeping the knowledge base up-to-date and consistent.

The architectural editor provides the user with a graphical interface for defining UML2-compliant architectural models. The designer defines the configuration of components and connectors that comprise the input architecture. The developer also specifies architectural characteristics in terms of types of elements, quality-attribute levels and the characterization of the interaction services between components.

The CBR engine is the reasoning part of SAME. The CBR paradigm establishes a reasoning framework based on the retrieve, reuse, revise and retain phases [30]. Initially, the CBR engine traverses all the connectors in the input architecture and gathers the specific characteristics of the environment around them (that is, their attached components, interaction features and quality levels). For each connector being visited, the CBR engine retrieves from the repository a collection of relevant experiences that are potentially useful to materialize the target architecture. The engine then ranks those experiences by assessing their similarity with the connectors in the architecture (*retrieve* phase of the CBR process). Based on that ranking, the engine selects the most promising experiences as candidates to work with. Then, it adapts the object-oriented solutions of the experiences to fit the architectural elements to be materialized, and generates a set of partial object-oriented designs (*reuse* phase). Finally, these individual designs are assembled together into the object-oriented alternative that implements the input architecture.

The proposed alternatives are defined as EMF-based implementations of the UML 2.x meta-model for the Eclipse platform (http://www.eclipse.org/uml2/). The tool uses an integrated UML editor which provides a graphical interface to show and edit class diagrams. At this point, the designer is engaged in an exploratory process in which s/he assesses the "goodness" of the solutions being proposed by the tool (*revise* phase). If a solution satisfies the designer's expectations, then the CBR engine incorporates the new adapted experience (*retain* phase) to the knowledge repository for future use. Conversely, if the designer considers the proposed alternative as inadequate, s/he can ask the engine for another solution by selecting other candidate experiences until an adequate solution is obtained. If this is not the case, the designer can draw a personal solution that will be added to the knowledge base as a new materialization experience.

The setup of the tool requires filling the repository of design experiences with an initial set of experiences from which all other experiences can be derived. We have identified a set of seed experiences based on well-known connector types found in the literature, such as procedure calls, abstract servers, event channels, the observer pattern, producer–consumer, RMI and SOAP connectors, brokers, semaphores, among others [18,21,26,33,35]. The system maintainer imports primitive experi-

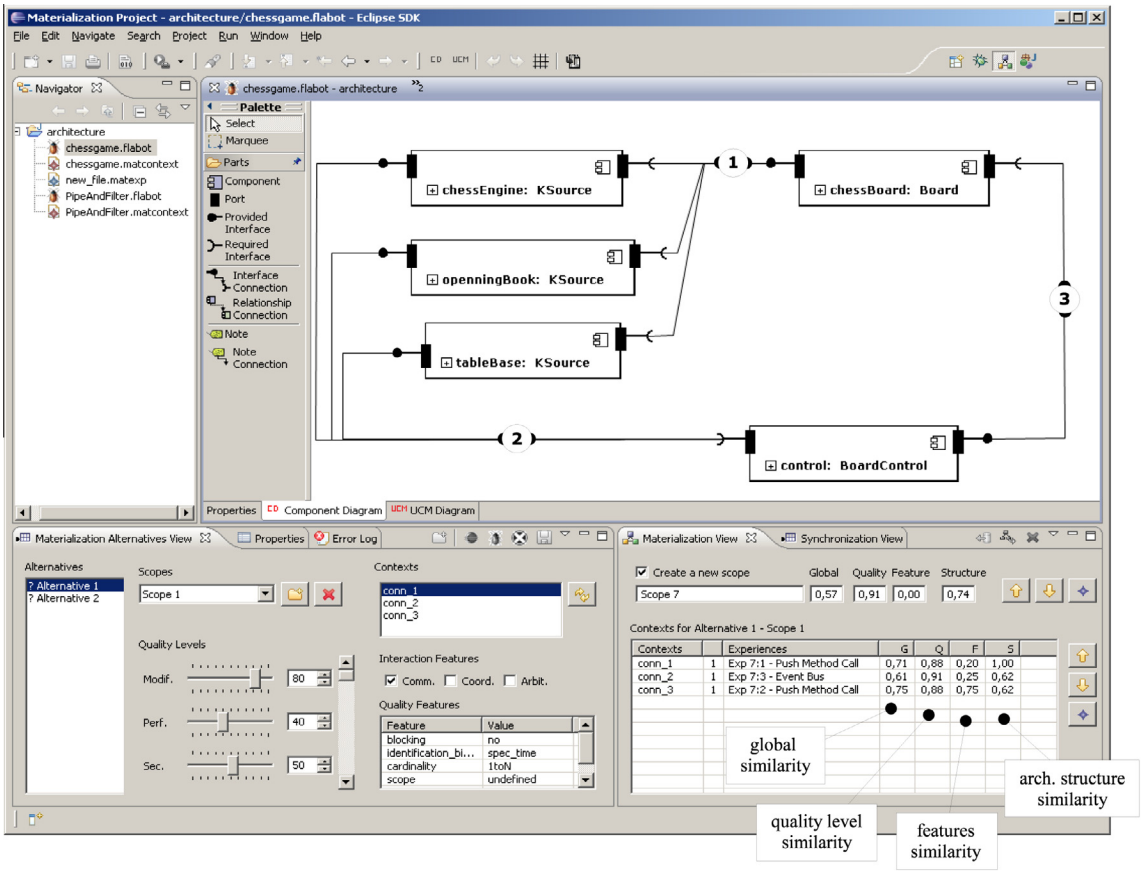

Fig. 8. A snapshot of SAME at work.

ences into the repository by means of the experience manager component. However, at any time, more primitive experiences can be imported into the repository in order to augment and diversify its design knowledge.

Fig. 8 shows a snapshot of the SAME tool at work. In the view at the left-bottom corner of the Eclipse perspective, the tool presents to the user all the connectors extracted from the input architecture. The user specifies the characteristics of each connector's context, which are used by the tool to retrieve relevant experiences from the repository. In the materialization view on the right-bottom corner in Fig. 8, the tool shows the list of retrieved experiences along with their similarity with the corresponding connector. The CBR engine uses those experiences to derive an object-oriented alternative for materializing the input architecture. However, the user might choose other experiences, less similar than the ones proposed by the tool, but that s/he considers valuable in the current context. This way, SAME allows users to explore multiple alternatives until an adequate object-oriented materialization is achieved.

7. Experiments

In this section, we present preliminary results of using SAME in two analysis axis: recall/precision, and user experiments. The experiments were conducted in the context of a real system for an embedded system for controlling telephone booths. We used a repository with 34 experiences describing known materialization problems. In the initial set of experiences, 16 of them were primitive experiences describing generic architecture specifications with their respective object-oriented interaction patterns. The remaining experiences were extracted from materializations of architectural styles and defined in terms of their specific component types (for instance, filter components from the pipe-and-filter style) [11,15].

To exercise the tool, we specified 10 system architectures distributed into two categories in the following way: 8 style-based system architectures and 2 application-specific system architectures for the embedded system. The input architectures were based on the same configuration of components and connectors, although we varied the characteristics of the connectors and the quality levels.

The evaluation process consisted of two different scenarios, which were presented in sequence. In the first scenario, we tested the tool with 5 of the target system architectures (4 style-based and 1 application-specific architectures) and the repository was filled with the initial set of materialization experiences. The new experiences resulting from the first scenario

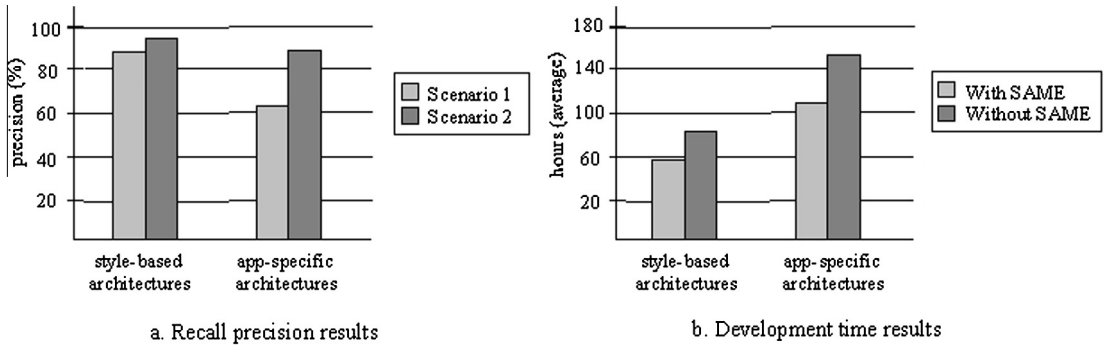

Fig. 9. Evaluation results.

evaluation were stored into the repository. Then, we performed the evaluation of the second scenario, so the remaining system architectures were materialized using the augmented repository.

To evaluate the retrieval algorithm, we exercised the tool by retrieving candidate experiences to materialize the input architectures. These architectures and the proposed materialization alternatives were then presented to test users for evaluation. A group of 4 test users (with software engineering experience) were asked to evaluate the solutions, based on information about the quality of the designs regarding the problem that it was supposed to solve. The retrieved experiences were analyzed and the following measure was computed:

$$\text{Precision} = \frac{\#\text{SuccesfulExperiences}}{\#\text{RetrievedExperiencies}} \tag{5}$$

Fig. 9a presents the precision results obtained from scenario 1 and scenario 2. The second scenario shows better global precision results (88.02%) than the first scenario (74.62%). This situation is expected, because when performing the second scenario evaluations the repository contained already new useful experiences that were incorporated during the execution of the first scenario. Therefore, we conjecture that the precision of the algorithm increases by exercising the tool. Additionally, due to the fact that the repository of experiences was initially loaded with seed experiences extracted from known architectural styles, the retrieve algorithm performed better for style-based architectures than for application-specific ones. However, we can observe that in the second scenario the relative growth of the algorithm precision is bigger for the application-specific architectures. This gives us an idea about how experiences used in previous systems can help to materialize current related system architectures (as is the case using SAME in the materialization of the related embedded systems). We argue that this aspect of the tool enables the building of a "design memory" of an organization in a corporate environment.

We have also measured the time that designers spent to develop systems that reify the target architectures used in the evaluation scenarios. This experiment was intended to corroborate our presumptions about how the use of SAME can aid developers during the software design process. Each target system was assigned to 2 groups of undergraduate students with limited experience in software development. The first group of students used the SAME tool to develop their assigned systems; whereas the second group of students developed the systems from scratch without using the tool.

Fig. 9b presents the average time in hours that the students spent to develop the systems. Although the first group of students needed time for learning how to use the tool, they spent considerable less time to develop the system implementations. Basically, the tool already makes certain design decisions to derive the object-oriented materialization alternatives that otherwise developers should make on their own. Our first group of students used the designs derived by the tool as the starting point to begin the implementation of their systems. Finally, we would like to remark that the students involved in the experiments have experience designing and implementing style-based system architectures. This is the reason why they spent less time developing the style-based systems than the time they spent developing the application-specific ones.

8. Related work

The problem of architectural materialization has been traditionally addressed by application frameworks that capture architectural abstractions of related systems providing extensible templates for applications within particular domains [17,25]. An application framework is designed as a set of semi-complete object-oriented elements, capturing the common features defined by a reference architecture that is specified to fit a given application domain. Then, a custom object-oriented materialization of the architecture is derived via instantiation/specialization of the framework. Application frameworks are a well-known software development technology with benefits such as: potential for reuse, reduction of development effort and cost, modularity and extensibility. Nevertheless, application frameworks are still tied to related application domains, therefore, the applicability of their underlying design knowledge is restricted to systems of the same kind. In a similar line, Medvidovic et al. [32] describe a family of implementation frameworks supporting materialization of C2-based software architectures. C2 is a particular architectural style used for building message-based applications, typically in graphical user

interface applications. The authors have proposed a set of object-oriented frameworks for materializing a C2 system through instantiation of one of those frameworks, so that an object-oriented implementation can be generated. ArchMatE is another tool that addresses the materialization problem focusing on architectural styles as the main entities to achieve software reuse [15]. The tool materializes style-based architectures by means of mini-frameworks, called framelets, which are selected from a repository based on users' quality preferences. Other approaches have aimed at deriving implementations from software architectures by means of specific programming technologies. The ArchJava approach [3,4] extends the Java language by incorporating architectural features such as components and ports. Given an ArchJava-compliant program specification, a special compiler maps the architectural features defined in the program into standard Java language. Thus, the compiler generates a running system implementation and ensures that this implementation conforms to the properties of the derived architecture. Unlike ArchJava, Tibermacine et al. [39] outlines an architecture conformance approach that combines architectural constraints with model transformations. Tibermacine's approach includes a tool that reminds the developer about possible loss of quality-attribute properties during system evolution. The links between quality attributes and architectural decisions are documented using contracts, which are captured in a special language called ACL and based on the UML Object Constraint Language (OCL). This way, the tool is able to point out inconsistencies in terms of ACL constraints that are not satisfied by the current architecture. Although the interaction models used in SAME seem similar to contracts, SAME differs from Tibermacine's approach in two aspects. First, ACL defines constraints for design models while SAME does not deal with constraints. Second, the constraint-based approach is not concerned with the exploration of materializations as SAME does. Nonetheless, we think that ArchJava and ACL can work as good complements for SAME by formalizing the context for materialization experiences. A related proposal is that of [6], where the authors use grey-box connectors as static meta-programs for implementing component interactions. Grey-box connectors produce adequate glue code providing a mean to adapt component interactions and to compose together heterogeneous components.

The main drawback of existing approaches addressing the materialization problem resides in that it is not clear how they deal with quality attributes both at the architecture design and object-oriented design levels. Additionally, a generic method for reifying heterogeneous architectural designs is still a necessity since current methods relies on specific technologies or deal with application-specific or style-based system architectures. By focusing on the quality characteristics and the quality of services of architectural connectors and by deriving technology-independent object-oriented designs, we argue that the approach developed in this article overcomes the disadvantages of current approaches. In the context of quality attributes, a recent experience that integrates both architectural design and object-oriented design techniques is described in [38].

Regarding the characteristics of architectural elements, we can mention a number of efforts that provided an important knowledge background for the foundations of our work. In [2], a formal basis for specifying connectors is described, by means of a formal notation that characterizes the protocols of interaction between architectural components. A small set of high-level transformations are defined in [36] that, when applied compositionally to primitive connectors, produce a variety of complex connectors. In [33] and [35], the authors classify common connector types regarding the interaction services they provide. An exhaustive set of connectors used in the development of real-time systems is described in [18]. Also, the work in [26] defines a featured-based taxonomy for a set of well known architectural elements - both components and connectors - commonly used in architecture composition. In [37], the authors describe a framework for classifying architectural styles by means of a set of features regarding control and data issues of each style.

Nowadays, the concepts and techniques of architectural description languages are considered as a form of model-based engineering (MDE) [16]. MDE focuses on open standards and supporting tools for system modeling and transformation, usually based on UML profiles. In principle, MDE tools can generate implementations from architectural designs in such a way the correspondence between the architecture and its implementation is given by construction. However, current MDE tools still have limitations with respect to exploration of alternative designs. MDE tools usually provide transformations that only support one-to-one mappings between architecture and implementation. These transformations do not consider quality-attribute concerns explicitly. Anyway, since SAME generates implementations from an architectural specification, it can be seen as a MDE tool in a broad sense.

On the other hand, the use of CBR techniques in software design has been mostly related to code reuse. In [19], the authors describe a tool to reuse source code by locating software components based on the functionality these components realize and by extending the implementation of the components according to particular requirements. Here, a case comprises a lexical description representing the component functionality, the source code implementation and the code properties to justify the use of the component in a new context. CAESAR is another CBR tool [20] that uses data-flow analysis to decompose a program into functional slices and to extract the code structures that performs those functions. Those slices are stored in the case library, and they are retrieved and adapted to match a specific user specification. REBUILDER UML [22] is a tool that uses domain knowledge to analyze system requirements and to search in a repository of class diagrams for a class diagram that is useful to realize those requirements. Basically, we have observed that current CBR tools deal with software reuse at the object-oriented level, aiming at deriving object-oriented implementations from functional requirements. However, it is not clear how these approaches manage the variability imposed by quality-attribute requirements in the derived object-oriented materializations. Additionally, we think that the transition from requirement analysis to object-oriented design is not straightforward because the analysis and design activities deal with different things: entities in the problem domain and entities in the solution domain [27]. A reasonable transition from analysis to object-oriented design should be supported by an intermediate design model – the software architecture - that serves as the container of the system functionality, and at the same time, it facilitates the achievement of quality-attribute requirements. This aspect is precisely

the contribution of SAME, as it provides semi-automated support to assist developers in the architecture-to-object mappings.

9. Conclusions and future work

We have described a novel CBR approach for deriving object-oriented implementations of software architectures. The approach represents a step towards a systematic design process that links the architecture and the object-oriented design worlds by means of the codification of design experiences. We have developed a tool, called SAME, for semi-automating the architecture materialization process driven by architectural connectors and quality attributes. The tool is not intended to derive a running implementation of an architecture, since this is a very complex task and involves an active participation from developers. Instead, SAME is used by developers to derive an object-oriented design structure that serves as a skeleton for implementing the system functionality. In that sense, SAME is an assistant that improves developer's work in the software design process.

Besides tool support, other benefits arise from the use of SAME as a design assistant. It is well-known that the implementation of connectors is frequently spread among the implementation of components, so the connections between objects in the resulting object-oriented design are not always clear [35]. Therefore, tracing connector implementations is not trivial in an object-oriented design. Our approach proposes a mechanism for individually materializing components and connectors, and for unifying their object-oriented counterparts into a single design. Materialization experiences codify the mapping between architectural elements and their corresponding object-oriented elements. These mappings make the realizations of architectural elements explicit in the object-oriented design, being the medium by which design decisions can be traced throughout the materialization process. Additionally, the conceptual integrity of the architectural design [10,12] can be preserved in the final object-oriented design by guiding developers in the materialization process through a set of proved design structures used to generate the implementation of the elements defined by a given architecture. Due to its fine-grained approach focused on architectural connectors, we believe that SAME becomes very beneficial when object-oriented implementations for arbitrary architectural designs need to be explored.

Currently, the user evaluates the quality of the object-oriented designs derived by SAME. S/he can accept or reject the solutions proposed by the tool. This method of evaluation is subjective and the results can vary depending on the developer. To reduce this subjectivity, we are exploring the automation of the quality-attribute evaluation of the derived designs. This way, the tool could guide the user in the evaluation procedure, providing more reliable feedback to the CBR engine and improving the confidence of the solutions proposed by the tool.

Another concern for the SAME tool is that it computes the similarity between cases by considering the particular dimensions of connector catalogues. The attributes and values for these dimensions depend always upon the overall design context, the application domain and the designer's perspective of the problem. As a consequence, the results of the similarity function may be biased. So far, we have taken a simple approach based on the structural characteristics of components playing similar roles when attached to connectors. However, a stronger compatibility check would require the components to be also equivalent from a behavioral point of view. A related drawback here is the lack of behavioral modeling in the C& C architectural specifications. In the current SAME implementation, the designer cannot give details about the way components behave when interacting with each others. This prevents the adaptation of the object-oriented solutions to generate behavioral diagrams - such as sequence diagrams - that would provide a more complete picture of the object-oriented implementation to the designer. The behavioral aspects of materializations are a topic for future work.

Finally, although SAME provides an editor for the creation of materialization experiences, the specification of the interaction models is still a highly manual task. To overcome this situation, we are planning to extend the SAME Eclipse plugin to provide a user-friendly interface that will support the construction of interaction models for the materialization experiences.

Acknowledgements

The authors thank Dr. Analia Amandi for her contributions to the CBR work, and also the anonymous reviewers that helped us to improve the quality of the final manuscript.

References

[1] A.A. Abd-Allah, Composing Heterogeneous Software Architectures, Technical Report, University of Southern California, 1996.
[2] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology. 6 (3) (1997) 213–249.
[3] J. Aldrich, C. Chambers, D. Notkin, Architectural reasoning in ArchJava, ECOOP'02: Proceedings of the 16th European Conference on Object-Oriented Programming, Springer-Verlag, 2002, pp. 334–367.
[4] J. Aldrich, Using Types to Enforce Architectural Structure, Proceedings Working International Conference on Software Architecture (February 18–22, 2008), Vancouver, BC, Canada, 2008.
[5] S. Ambler, The Elements of UML 2.0 Style, Cambridge University Press, 2005.
[6] U. Aßmann, A. Ludwig, D. Pfeifer, Programming connectors in an open language, in: Web Proceedings of WICSA 1, Working IFIP Conference on Software Architecture, 1999.
[7] L.J. Bass, M. Klein, F. Bachmann, Quality attribute design primitives and the attribute driven design method, in: PFE'01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering, Springer-Verlag, 2002, pp. 169–186.
[8] C. Choppy, D. Hatebur, M. Heisel, Architectural patterns for problem frames, in: IEE Proceedings – Software, vol. 152 (4) (2005), pp. 198–208.

[9] B. Boehm, H. In, Identifying quality-requirement conflicts, IEEE Software, vol. 13, IEEE Computer Society Press, 1996. pp. 25–35.
[10] F.P. Brooks, The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition, Addison-Wesley Professional, 1995.
[11] F. Buschmann, Pattern-Oriented Software Architecture, Vol. 1, A System of Patterns Wiley, 2002.
[12] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley, 2003.
[13] M. Campo, A. Diaz-Pace, M. Zito, Developing object-oriented enterprise quality frameworks using proto-frameworks, Software: Practice and Experience. 32 (8) (2002) 837–843.
[14] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, R. Little, Documenting Software Architectures: Views and Beyond, Pearson Education, 2002.
[15] J.A. Díaz-Pace, M.R. Campo, ArchMatE: from architectural styles to object-oriented models through exploratory tool support, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (San Diego, CA, USA, October 16–20, 2005) OOPSLA'05, ACM Press, New York, NY, 2005, pp. 117–132.
[16] G. Edwards, S. Malek, N. Medvidovic, Scenario-driven dynamic analysis of distributed architectures, in: Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE), March 2007.
[17] M. Fayad, D. Schmidt, R. Johnson, Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley, 1999.
[18] J.L. Fernandez, A Taxonomy of Coordination Mechanisms used by Real-time Processes, Ada Letters, vol. XVII, ACM Press, 1997.
[19] C. Fernandez-Chamizo, P.A. González-Calero, M. Gómez-Albarrán, Luis Hernández-Yánez, Supporting object reuse through case-based reasoning, in: EWCBR'96: Proceedings of the Third European Workshop on Advances in Case-Based Reasoning, Springer-Verlag, 1996, pp. 135–149.
[20] G. Fouqué, S. Matwin, A case-based approach to software reuse, Journal of Intelligent Information Systems 1993 (2) (1997) 165–197.
[21] E. Gamma, R. Helm, R. Johnson, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[22] P. Gomes, F.C. Pereira, P. Paiva, N. Seco, P. Carreiro, J.L. Ferreira, C. Bento, Using wordnet for case-based retrieval of UML models, AI Communications 17 (1) (2004) 13–23.
[23] J. Hautamäki, Pattern-Based Tool Support for Frameworks: Towards Architecture-Oriented Software Development Environment, Ph.D. Thesis, Tampere University of Technology, Publication 521, 2005.
[24] I. Hammouda, M. Harsu, Documenting maintenance tasks using maintenance patterns, in: Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR 2004), Tampere, Finland, 2004, pp. 37–47.
[25] R.E. Johnson, Components, frameworks and patterns, in: M. Harandi (Ed.), Proceedings of the 1997 Symposium on Software Reusability (Boston, Massachusetts, United States, May 17–20, 1997), ACM Press, New York, NY, 1997, pp. 10–17.
[26] R. Kazman, P.C. Clements, L. Bass, G.D. Abowd, Classifying architectural elements as a foundation for mechanism matching, in: Proceedings of the 21st international Computer Software and Applications Conference (August 11–15, 1997) COMPSAC, IEEE Computer Society, Washington, DC, 1997, pp. 14–17.
[27] H. Kaindl, Difficulties in the Transition from OO Analysis to Design, IEEE Software, 16, IEEE Computer Society Press, 1999, pp. 94–102.
[28] Kolodner, J.L. Improving Human Decision Making through Case-based Decision Aiding. (1991). AI Magazine, American Association for Artificial Intelligence, 1991, 12, 52-68.
[29] D.B. Leake, Case-Based Reasoning: Experiences, Lessons and Future Directions, MIT Press, 1996.
[30] R. Lopez De Mantaras, D. McSherry, D. Bridge, D. Leake, B. Smyth, S. Craw, B. Faltings, M.L. Maher, M.T. Cox, K. Forbus, M. Keane, A. Aamodt, I. Watson, Retrieval, reuse, revision and retention in case-based reasoning, Knowledge Engineering Review 20 (3) (2005) 215–240.
[31] C. Mattmann, D. Woollard, N. Medcidovic, R. Mahjourian, Software connector classification and selection for data-intensive systems, in: Proceedings of the Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS'07), IEEE Computer Society, 2007, ISBN 0-7695-2966-6, pp. 4–9.
[32] N. Medvidovic, N.R. Mehta, M. Mikic-Rakic, A family of software architecture implementation frameworks, in: Proceedings of the IFIP 17th World Computer Congress - Tc2 Stream/ 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (August 25–30, 2002), in: J. Bosch, W.M. Gentleman, C. Hofmeister, J. Kuusela (Eds.), IFIP Conference Proceedings, vol. 224, Kluwer B.V., Deventer, The Netherlands, 2002, pp. 221–235.
[33] N.R. Mehta, N. Medvidovic, S. Phadke, Towards a taxonomy of software connectors, in: Proceedings of the 22nd international Conference on Software Engineering (Limerick, Ireland, June 04–11, 2000). ICSE'00, ACM Press, New York, NY, 2000, pp. 178–187.
[34] L. Rapanotti, J. Hall, M. Jackson, B. Nuseibeh, Architecture-driven problem decomposition, in: Proceedings of the 12th IEEE International Conference on Requirements Engineering (September 06–10, 2004), RE, IEEE Computer Society, Washington, DC, 2004, pp. 80–89.
[35] M. Shaw, Procedure calls are the assembly language of software interconnection: connectors deserve first-class status, in: Selected papers from the Workshop on Studies of Software Design, Springer-Verlag, 1996, pp. 17–32.
[36] B. Spitznagel, D. Garlan, A compositional formalization of connector wrappers, ICSE'03: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, 2003.
[37] M. Shaw, P. Clements, A field guide to boxology: preliminary classification of architectural styles for software systems, in: COMPSAC'97: Proceedings of the 21st International Computer Software and Applications Conference, IEEE Computer Society, 1997, pp. 6–13.
[38] R. Sangwan, C. Neill, Z. El-Houda, M. Bass, Integrating Software Architecture-Centric Methods into Object-Oriented Analysis and Design, Journal of Systems and Software 81 (5) (2008) 727–746.
[39] C. Tibermacine, R. Fleurquin, S. Sadou, On-demand quality-oriented assistance in component-based software evolution, in: Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06), LNCS 4063, Springer-Verlag, 2006, pp. 294–309.
[40] G.L. Vazquez, M.R. Campo, J.A. Diaz-Pace, A case-based reasoning approach for materializing software architectures onto object-oriented designs, in: Proceedings of the 2008 ACM Symposium on Applied Computing (Fortaleza, Ceara, Brazil, March 16–20, 2008), ACM, New York, NY, 2008, pp. 842–843.