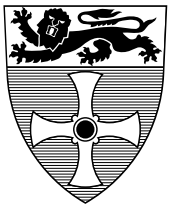
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Towards an Engineering Approach to Component Adaptation

S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, M. Tivoli

TECHNICAL REPORT SERIES

No. CS-TR-939 January, 2006

TECHNICAL REPORT SERIES

No. CS-TR-939 January, 2006

Towards an Engineering Approach to Component Adaptation

Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, Massimo Tivoli

Abstract

Component adaptation needs to be taken into account when developing trustworthy systems, where the properties of component assemblies have to be reliably obtained from the properties of its constituent components. Thus, a more systematic approach to component adaptation is required when building trustworthy systems. In this paper, we illustrate how (design and architectural) patterns can be used to achieve component adaptation and thus serve as the basis for such an approach. The paper proposes an adaptation model which is built upon a classification of component mismatches, and identifies a number of patterns to be used for eliminating them. We conclude by outlining an engineering approach to component adaptation that relies on the use of patterns and provides additional support for the development of trustworthy component-based systems.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

BECKER, S., BROGI, A., GORTON, I., OVERHAGE, S., ROMANOVSKY, A., TIVOLI, M

Towards an Engineering Approach to Component Adaptation
[By] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, M. Tivoli

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-939)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-939

Abstract

Component adaptation needs to be taken into account when developing trustworthy systems, where the properties of component assemblies have to be reliably obtained from the properties of its constituent components. Thus,a more systematic approach to component adaptation is required when building trustworthy systems. In this paper, we illustrate how (design and architectural) patterns can be used to achieve component adaptation and thus serve as the basis for such an approach. The paper proposes an adaptation model which is built upon a classification of component mismatches, and identifies a number of patterns to be used for eliminating them. We conclude by outlining an engineering approach to component adaptation that relies on the use of patterns and provides additional support for the development of trustworthy component-based systems.

About the author

Steffen Becker is a Research Assistant at the University of Oldenburg (Germany). His research interests are Component Based Software Architectures and Engineering, Quality of Service of Component Based Applications, Adaptation of Software Components, Generation of Adaptors and Design Patterns for Software Architectures. More information about Steffen's work can be found at http://se.informatik.uni-oldenburg.de/Members/steffen

Massimo Tivoli is a research scientist at INRIA Rhone Alpes Montbonnot (Grenoble, France) supported by a 1-year post-doctoral program. He is a member of the POP-ART project team and is working on building an integrated component-based development framework for real-time synchronous systems. He received is M.Sc and Ph.D. degrees in Computer Science from the University of L'Aquila (Italy) with a first-class honours degree in April 2001 and June 2005, respectively. His research interests are mainly concentrated in the application of formal methods to the verification, analysis and automatic synthesis of complex systems. He is also a member of the SEALab at the Computer Science Department of the University of L'Aquila (Italy) where he is involved in working groups focused on the domain of Software Architectures and Component-based Software Engineering. He was also involved in the organization of WCAT 05.

Alexander (Sascha) Romanovsky is a Professor in the CSR. He received a M.Sc. degree in Applied Mathematics from Moscow State University and a PhD degree in Computer Science from St. Petersburg State Technical University. He was with this University from 1984 until 1996, doing research and teaching. In 1991 he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993 he was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, the University of Newcastle upon Tyne. In 1992-1998 he was involved in the Predictably Dependable Computing Systems (PDCS) ESPRIT Basic Research Action and the Design for Validation (DeVa) ESPRIT Basic Project. In 1998-2000 he worked on the Diversity in Safety Critical Software (DISCS) EPSRC/UK Project. Prof Romanovsky was a co-author of the Diversity with Off-The-Shelf Components (DOTS) EPSRC/UK Project and was involved in this project in 2001-2004. In 2000-2003 he was in the executive board of Dependable Systems of Systems (DSoS) IST Project. Now he is coordinating Rigorous Open Development Environment for Complex Systems (RODIN) IST Project (2004-2007).

Suggested keywords

ADAPTATION,
WRAPPERS,
PATTERNS

Towards an Engineering Approach to Component Adaptation

Steffen Becker[1], Antonio Brogi[2], Ian Gorton[3], Sven Overhage[4], Alexander Romanovsky[5], and Massimo Tivoli[6]

[1] Software Engineering Group, University of Oldenburg,
OFFIS, Escherweg 2, 26121 Oldenburg, Germany
steffen.becker@informatik.uni-oldenburg.de
[2] Department of Computer Science, University of Pisa,
Largo B. Pontecorvo 3, 56127 Pisa, Italy
brogi@di.unipi.it
[3] Empirical Software Engineering Group, National ICT Australia
Bay 15 Locomotive Workshop, Australian Technology Park Eveleigh, NSW 1430 Australia
ian.gorton@nicta.com.au
[4] Dept. of Software Engineering and Business Information Systems,
Augsburg University, Universitätsstraße 16, 86135 Augsburg, Germany
sven.overhage@wiwi.uni-augsburg.de
[5] School of Computing Science, University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, United Kingdom
alexander.romanovsky@ncl.ac.uk
[6] Dept. of Computer Science, University of L'Aquila,
Via Vetoio N.1, 67100 L'Aquila, Italy
tivoli@di.univaq.it

Abstract. Component adaptation needs to be taken into account when developing trustworthy systems, where the properties of component assemblies have to be reliably obtained from the properties of its constituent components. Thus, a more systematic approach to component adaptation is required when building trustworthy systems. In this paper, we illustrate how (design and architectural) patterns can be used to achieve component adaptation and thus serve as the basis for such an approach. The paper proposes an adaptation model which is built upon a classification of component mismatches, and identifies a number of patterns to be used for eliminating them. We conclude by outlining an engineering approach to component adaptation that relies on the use of patterns and provides additional support for the development of trustworthy component-based systems.

1 Introduction

In an ideal world, component-based systems are assembled from pre-produced components by simply plugging perfectly compatible components together, which jointly realize the desired functionality. In practice, however, it turns out that the constituent components often do not fit one another and adaptation has to be done to eliminate the resulting mismatches.

Mismatches between components, for example, always need to be addressed when integrating *legacy systems*. Thereby, the impossibility of modifying large client applications is a major reason for the need to employ some form of adaptation during the development of component-based systems. Besides that, the most structured way to deal with *component evolution* and upgrading, which is likely to result in new mismatches at the system level, arguably is by applying adaptation techniques. Finally, adaptation becomes a major task in the emerging area of *service-oriented computing*, where mismatches must be solved to ensure the correct interoperation among different Web services, which have been assembled according to a bottom-up strategy.

For these reasons, the adaptation of components has to be recognized as an unavoidable, crucial task in Component-Based Software Engineering (CBSE). Until now, however, only a number of isolated approaches to eliminate mismatches between components have been proposed. They introduce adapters, which are capable of mediating the interaction between components (e.g., to transform data between different formats or to ensure a failure-free coordination protocol). In [1–9], for instance, the authors show how to automatically derive adapters in order to reduce the set of system behaviors to a subset of safe (e.g., lock-free) ones. Other papers [9–14] show how to plug a set of adapters into a system in order to augment the system behaviour by introducing more sophisticated interactions among components. The presented protocol transformations can be applied to ensure the overall system dependability, improve extra-functional characteristics, and properly deal with updates of the system architecture (e.g., insertion, replacement, or removal of components).

The approaches mentioned above only address some forms of component mismatch types, employ specific specification formalisms, and usually do not support any reasoning about the impact that component adaptation has on the extra-functional properties (e.g., reliability, performance, security) of the system. For these reasons, employing these approaches to adapt components in an ad hoc strategy typically is error prone, reduces the overall system quality, and thus increases the costs of system development. Above all, employing such an ad hoc strategy to component adaptation hinders the development of *trustworthy component-based systems*, since it is impossible to reliably deduce the properties of component assemblies from the properties of the constituent components and the created adapters.

To counter these problems, it is the objective of this paper to initiate the development of an *engineering approach* to component adaptation that provides developers with a systematic solution consisting of methods, best practices, and tools. As the basis for such an approach we suggest the usage of *adaptation patterns*, since they provide generic and systematic solutions to eliminate component mismatches. Before establishing the details of the proposed engineering approach, we start by clarifying important concepts (section 2). After introducing an initial taxonomy of component mismatches (section 3), we describe a generic process model for component adaptation and discuss relevant patterns that have emerged both in literature and in practice (section 4). To illustrate the employment of patterns to eliminate component mismatches, we additionally present some examples (section 5). After discussing related work we conclude by outlining some of the remaining challenges. They will have to be solved to establish

a fully-fledged engineering approach, capable of supporting the development of trustworthy component-based systems.

2 Component Adaptation: Coming to Terms

Component mismatches originate from contradicting assumptions about the context, in which interacting components should be used, and the real context, in which they are being deployed. These contradicting assumptions have been made by the developers of individual components and become obvious during the assembly of the system, when individual components are brought together. Component mismatches have been examined both from an architectural [15] and a reuse-oriented perspective [16].

From a reuse-oriented perspective there always is a tension between the goals of extending the functionality of a component on the one hand and keeping it reusable on the other hand. These are contradicting goals, since reuse typically requires simple, well-defined and well-understood functionality. Because of this reason, it is likely that a reused component will not exactly fit the required context. In the software reuse community, component mismatches are usually called *component incompatibilities*.

From a software architecture perspective, problems occur when components have different assumptions about normal and abnormal behaviour of other components or when a software architect makes decisions which contradict individual assumptions of the components and connectors [17]. Problems of this kind are called *architectural mismatches*. In our paper, we summarize the terms ”component incompatibilities” and ”architectural mismatches” as *component mismatches* to emphasize that they relate to the same problem.

Before we elaborate the proposed engineering approach to component adaptation, some terms have to be clarified as they are not used consistently in the domain of adaptation techniques.

Software adaptation is the sequence of steps performed whenever a software entity is changed in order to comply with requirements emerging from the environment in which the entity is deployed. Such changes can be performed at different stages during the life cycle. Therefore, we distinguish requirement adaptation, design-time adaptation, and run-time adaptation (see [18]):

- Requirement adaptation is used to react to changes during requirements engineering, especially when new requirements are emerging in the application domain.
- Design-time adaptation is applied during architectural design whenever an analysis of the system architecture indicates a mismatch between two constituent components.
- Run-time adaptation takes place when parts of the system offer different behaviour depending on the context the parts are running in. This kind of adaptation is therefore closely related to context-aware systems.

In the following, we restrict ourselves to design-time adaptation.

Software Component Adaptation is the sequence of the steps required to bridge a component mismatch. According to the common definition, components offer services to the environment, which are specified as provided interfaces [19, 20]. In addition, components explicitly and completely express their context dependencies [19], i.e.

their expectations on the environment. Context dependencies are stated in the form of required interfaces [19, 20]. Using the concept of provided and required interfaces, a component mismatch can be interpreted as a mismatch between properties of required and provided interfaces, which have to be connected (see figure 1). Consequently, identifying mismatches between components is equivalent to identifying mismatches between interfaces. A component mismatch thus occurs, when a component, which implements a provided interface, and a component, which uses a required interface, are not cooperating as intended by the designer of the system.

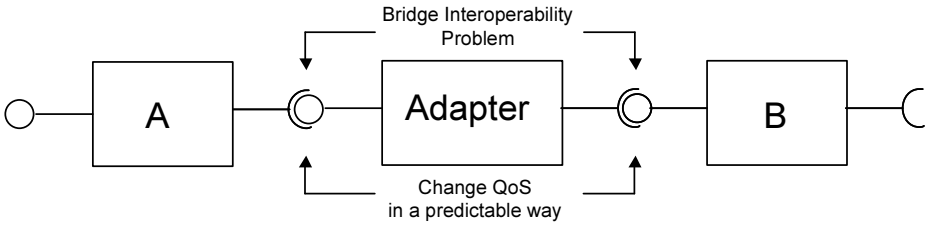


Fig. 1. A software component adapter and its QoS impact.

Note that component mismatches explicitly refer to interoperability problems which have not been foreseen by the producer of one of the components. Many components offer so called *customization interfaces* to increase reusability. These interfaces allow changes to the behaviour of the component during assembly time by setting parameters. As they are foreseen by the component developer and thus planned in advance, we do not consider parameterization as adaptation. Therefore, in the following customization is disregarded.

In accordance with the term *adaptation* we define a *software component adapter* as a software entity especially constructed to overcome a component mismatch.

3 A Taxonomy of Component Mismatches

Although an efficient technique to adapt components is of crucial importance to facilitate CBSE, there currently exist only a few approaches to enumerate and classify different kinds of component mismatches [21]. Moreover, many of the existing approaches just broadly distinguish between *syntactic*, *semantic*, and *pragmatic mismatches* and put them into relation to various aspects of compatibility like *functionality*, *architecture*, and *quality* [22, 23]. In order to get a more detailed understanding of the problem domain, we start implementing the proposed engineering approach to component adaptation by introducing a *taxonomy of mismatches*. The introduced taxonomy enumerates different types of component mismatches which will be taken into consideration when we develop a pattern-based approach to adaptation later on.

In addition, the provided taxonomy summarizes the different types of component mismatches into categories and classifies them according to a hierarchy of *interface models* (see figure 2). Each of the distinguished interface models determines a (distinct)

set of properties which belongs to a component interface [24, 25]. Because component mismatches originate from mismatching properties of connected interfaces (the so-called provided and required interfaces, cf. section 2), the hierarchy of interface models underlying the interface descriptions simultaneously determines our ability to diagnose and eliminate a certain type of component mismatch.

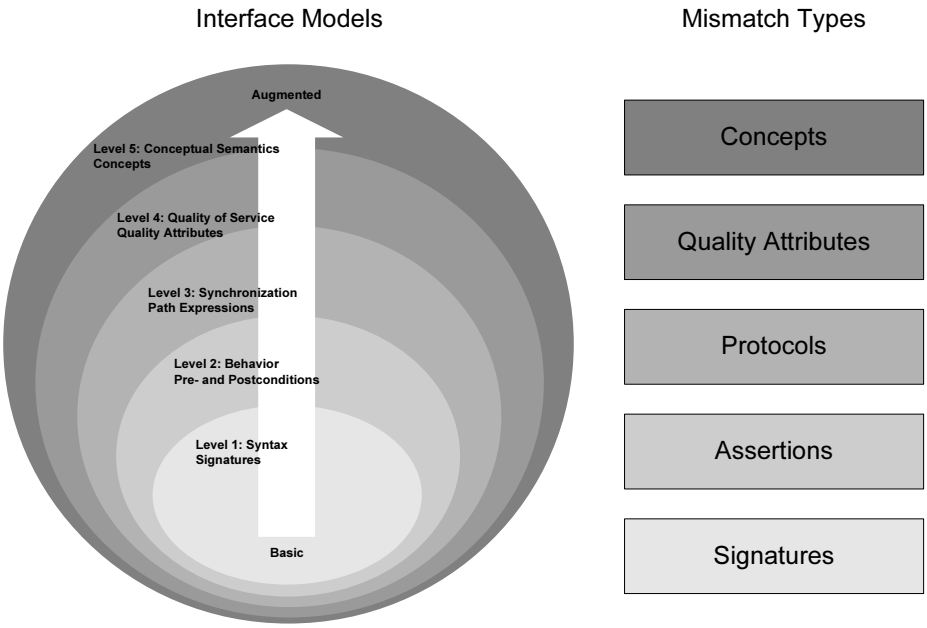


Fig. 2. A hierarchy of interface models (based on [24]), which orders interface properties according to their specification complexity, supports the identification and elimination of different types of component mismatches.

The (classical) *syntax-based interface model*, which focuses on signatures as constituent elements of component interfaces, supports the identification and elimination of *signature mismatches*. By using such a syntax-based interface model, the following types of (adaptable[1]) mismatches can be distinguished when connecting the required interface of a component "A" with a provided interface of a component "B" as shown in figure 1 [26, 27]:

- Naming of methods. Methods, which have been declared in the provided and required interface, realize the same functionality but have different names.
- Naming of parameters. Parameters of corresponding methods represent the same entity and have the same type but have been named differently in the provided and required interface.

[1] An adaptable mismatch can eventually be eliminated by adaptation.

- Naming of types. Corresponding (built-in or user-defined) types have been declared with different names.
- Structuring of complex types. The member lists of corresponding complex types (e.g. structures) declared both in the provided and required interface are permutations.
- Naming of exceptions. Exceptions thrown by corresponding methods have the same type, but have been declared with different names.
- Typing of methods. The method declared in the provided interface returns a type, which is a sub-type of the one that is returned by the method declared in the required interface.
- Typing of parameters. Parameters of methods declared in the provided interface have a type, which is a super-type of the one that belongs to corresponding parameters declared in the required interface.
- Typing of exceptions. Exceptions thrown by methods declared in the provided interface have a type, which is a sub-type of the one that belongs to corresponding exceptions declared in the required interface.
- Ordering of parameters. The parameter lists of corresponding methods declared both in the provided and required interface are permuted.
- Number of parameters. A method declared in the provided interface has fewer parameters or additional parameters with constant values compared to its corresponding method declared in the required interface.

Compared to this basic interface model, a *behavioral interface model* also contains *assertions* (i.e. pre- and postconditions) for the methods, which have been declared in the required and provided interfaces. With a behavioral interface model in place, it becomes principally conceivable to additionally search for (adaptable) mismatches between assertions when comparing provided and required interfaces. However, we chose *not* to consider the detection and adaptation of mismatching assertions as part of the proposed engineering approach, since they usually cannot be statically identified in an efficient manner [28, p. 578]. Instead, we refer to [29] for details about existing techniques, which can be applied to identify and adapt mismatching assertions, as well as their principal limitations.

By making use of an *interaction-based interface model*, which focuses on describing the interaction that takes place between connected components in the form of message calls, developers are able to diagnose and eliminate *protocol mismatches*. Provided that the interaction protocols belonging to the provided and the required interface are specified in a way that supports an efficient, i.e. statically computable, comparison, the following (adaptable) mismatches can be distinguished [2, 30]:

- Ordering of messages. The protocols belonging to the provided and required interface contain the same kinds of messages, but the message sequences are permuted.
- Surplus of messages. A component sends a message that is neither expected by the connected component nor necessary to fulfil the purpose of the interaction.
- Absence of messages. A component requires additional messages to fulfil the purpose of the interaction. The message content can be determined from outside.

Since it is generally possible to specify interaction protocols as pre- and postconditions [28, p. 981-982], we have to admit that introducing a behavioral interface model already

would have been sufficient to cover the interaction aspect as well. Nevertheless, we chose to view interaction protocols as a separate aspect that has to be distinguished from pre- and postconditions. This decision is mainly motivated by the problems that arise when trying to statically compare assertions. We have to admit, however, that our decision to view interaction protocols as a separate aspect only is profitable, if the specified interaction protocols can be statically compared in a more efficient way than assertions. To ensure a better comparability of protocol specifications, we eventually have to prefer notations of limited expressive power (e.g. finite state machines).

The *quality-based interface model* instead focuses on describing an aspect that has not been covered so far. It documents the Quality of Service (QoS) which is being provided by each of the interface methods by describing a set of quality attributes. The set of quality attributes that is to be described is determined by the underlying quality model, e.g. the ISO 9126 quality model [31, 32], which is one of the most popular. By making use of an interface model that is based on the ISO 9126 quality model, it is possible to detect and eliminate the following *quality attribute mismatches*:

- Security. The component requiring a service makes assumptions about the authentication, access, and integrity of messages that differ from the assumptions made by the component which provides the service.
- Persistency. The component requiring a service makes assumptions about the persistent storage of computed results that differ from the assumptions made by the component which provides the service.
- Transactions. The component requiring a service makes assumptions about the accompanying transactions that differ from the assumptions made by the component which provides the service.
- Reliability. The service required by component A needs to be more reliable than the one that is being provided by component B. Reliability is a trustworthiness attribute characterizing the continuity of the service, e.g. by measuring the meantime between failure, mean downtime, or availability [32, p. 23]. Typically reliability is achieved by employing fault tolerance means.
- Efficiency (Performance). The service required by component A needs to be more efficient than the one that is being provided by component B. The efficiency of a service is typically characterized by its usage of time and resources, e.g. the response time, throughput, memory consumption, or utilization of processing unit [32, pp. 42-50].

It is important to stress the fact that, with respect to adaptation, the quality aspect is a *cross-cutting concern*. This means, creating and inserting an adapter to eliminate one of the other component mismatches mentioned in this paper probably influences the quality properties, e.g. by delaying the response time of a service that now has to be invoked indirectly. In fact, the quality attributes distinguished above are even cross-cutting concerns among each other, which means that adapting one of the quality attributes is likely to influence the others.

A *conceptual interface model*, which describes the conceptual semantics of component interfaces as an ontology (i.e. a set of interrelated concepts), supports the identification and elimination of so-called *concept mismatches*. Thereby, concepts can principally characterize each of the elements contained in a syntactical interface model. Thus, they

may refer to entities (such as parameters, type declarations etc.), functions (methods), and processes (protocols). By making use of a concept model that consists of a term (denominator), an intension (definition), and an extension (corresponding real objects), the following *concept mismatches* can be principally distinguished [33, 34]:

- Synonyms. Two concepts, which characterize corresponding interface elements of a provided and required interface, are identical with respect to their definition, but have been used with different terms (e.g. customer and buyer).
- Sub- and Superordination. Two concepts, which characterize corresponding interface elements of a provided and required interface, are in a specialization or generalization relationship to each other.
- Homonyms. Two concepts, which characterize corresponding interface elements of a provided and required interface, are named with the same term but have different definitions (e.g. price as price including value-added tax and price as price without value-added tax).
- Equipollences. Two concepts, which characterize corresponding interface elements of a provided and required interface, have the same extension. However, they have different definitions which only share some common aspects (e.g. customer and debitor).

Both conceptual interface models, which make use of ontologies to describe the semantics of component-interfaces, as well as their usage for compatibility tests and adaptation are still under research. Consequently, there currently is little substantial support that can help in detecting and adapting concept mismatches (an overview of approaches can be found in [35, 36]). However, conceptual interface models are helpful in detecting and eliminating certain kinds of signature mismatches, like e.g. methods with identical functionality and different namings.

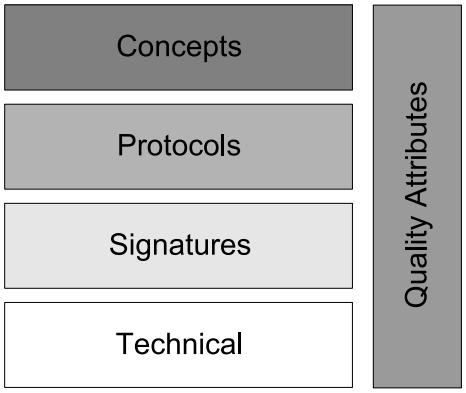


Fig. 3. The taxonomy contains five distinct classes of component mismatches.

To complete our taxonomy of component mismatches, we finally introduce *technical mismatches* as additional component mismatch type. Technical mismatches between

components occur, if two interacting components have been developed for different platforms (i.e. operating systems, frameworks etc.). Since technical dependencies of the former kind usually are not described as interfaces and instead remain as implicit component properties, they have not been covered by the introduced taxonomy so far, which builds upon the hierarchy of interface models to classify component mismatches. They represent an important mismatch type, however, and have to be considered accordingly when developing an engineering approach to adaptation. Figure 3 shows the classification of component mismatches that results from the inclusion of technical mismatches. It shows extra-functional mismatches as cross-cutting concern, whereas the other concerns can be summarized as functional mismatches.

4 Relevant Patterns for Component Adaptation

Patterns - either on the component design or on the architectural level - have become popular since the Gang of Four [37] published their well-known book on design patterns. According to our classification there are a lot of possible component incompatibilities. Therefore, it is reasonable that there are several patterns for bridging those incompatibilities. As patterns are established and well known solutions to reoccurring problems we decide to utilize patterns for adaptation problems. Thus, in this section we highlight some of the relevant patterns - mainly taken from literature [37–40].

Before we go into details, we focus on the basic structure of some of the patterns. Many patterns look similar or even identical at the design or source code level. This leads to the assumption that there are even more basic concepts used in the patterns than the patterns itself. For example, delegation is such a concept. Delegation takes place whenever a component wrapping another component uses the wrapped components service to fulfil its own service. For example, an adapter (see below) converting currencies from Euro to US Dollar. It first converts the input currency, then it delegates the call to the wrapped component using the right currency, and afterwards the currency is translated back again. The same idea is used also in, i.e., the Decorator pattern. Therefore, we try to identify in the following text these basic structures as well to build a taxonomy of the basic building blocks of the patterns introduced and also to capture the *basic technique* used in the patterns mentioned. An analysis of the basic techniques can also lead to a more engineering-based approach to adaptation in future work.

We classify the introduced patterns according to section 2 basically in adapters dealing primarily with functional aspects and extra-functional aspects respectively.

4.1 Functional Adaptation Patterns

This section gives an overview of the most often used patterns to bridge functional component incompatibilities. Most are well known to experienced developers and used quite frequently - even without the knowledge that a pattern has been used.

Adapter The adapter or wrapper pattern is described in [37, p. 139]. The pattern directly corresponds to the definition given in section 2 as its main idea is to bridge between two different interfaces. The pattern is used in different flavours: a variant using

inheritance and a second one based on delegation. The latter can be used in component-based development by using the concepts on the component instance level instead of the object instance level. The adapter pattern is very flexible as theoretically every interface can be transformed into every other interface. Thus, the range of adapters is infinite.

Decorator The decorator pattern [37, p. 175] can be seen as a special class of adapters where the adapter's interface is a subtype of the adapted component. This enables the use of a decorated component instead of the undecorated. Additionally, it is possible to decorate a single component as often as necessary. As the adapted component has the same list of signatures as the original component, a decorator can only change or add functionality to the methods already offered by the original component. As the decorator is a special kind of adapter it also uses delegation as main technique.

Interceptor Often the term interception is used when implementing aspect oriented programming (AOP) techniques. Interception is a technique which intercepts method calls and presents the call to some pre- or post-code for additional processing [39, p. 109]. It can be realized by the afore-mentioned decorator pattern but is often part of component runtime environments. For example, the J2EE container technology uses interception to add advanced functionality to components during deployment like container managed persistency or security. Basically, it also uses delegation but as said before often hidden in the runtime environments.

Wrapper Facade The wrapper facade pattern is used to encapsulate a non-object oriented API using wrapper objects [39, p. 47]. Therefore, it can also be used to encapsulate services in a component-based framework. The basic idea is to encapsulate corresponding state and functions operating on this state in a single component. For example, consider a file system component encapsulating a file handle and the operations which can be performed on the respective file. Basic principles used in this pattern are delegation and the encapsulation of state.

Bridge The bridge pattern is used to decouple an abstraction and its implementation [37, p. 151]. Thus, it is often used to define an abstract interface on a specific technology and its implementations deal with vendor specific implementations. Abstract GUI toolkits like Swing which can be used on top of different GUI frameworks can be seen as example. The basic technique here is the use of the subtype relation and polymorphism.

Microkernel The Microkernel pattern uses a core component and drivers to build an external interface to emulate a specific environment [38, p. 171]. It can be used to simulate a complete target environment on a different technological platform. The pattern has been used for adaptation in writing emulation layers or virtual machines.

Mediator The Mediator pattern is used to encapsulate how a given set of objects interact [37, p. 283]. A typical scenario in the context of adaptation is to use several components to provide a service, e.g., querying multiple database servers to return a single result set. The components can interact using the mediator's coordinating role. Often mediation is used simultaneously with the adapter pattern to transform data passed to or from the service in formats being expected by the respective interfaces. With a focus on data transformations the pattern is often also called *Coordinator* pattern [40, p. 111].

4.2 Extra-Functional Adaptation Patterns

The extra-functional patterns selected here are often used to increase a single or several quality attributes of the components being adapted. We give examples of properties that are often addressed by the patterns in the respective paragraphs.

Proxy A Proxy is put in front of a component to control certain aspects of the access to it [37, p. 207]. Security issues like access control, encryption, or authentication are often added to components by respective Proxys. Additionally, it can be used to implement caching strategies [40, p. 83] or patterns for lazy acquisition of resources [40, p. 38] to increment response times. The basic technique used in this pattern is delegation.

Component Replication The component replication pattern is derived from the object replication pattern [41, p. 99]. The idea is to distribute multiple copies of the same component to several distinct computation units to increase response time and throughput. Additionally, you might get an increased reliability in the case the controller coordinating the replicated components is not the point of failure. The basic technique in this pattern is based on copying the state of a component.

Process Pair The process pair pattern runs each component twice so that one component can watch the other and restart it in case of a failure [41, p. 133]. The pattern is used to increase the availability of components in high availability scenarios, e.g., whenever safety is an important aspect of the system design. The basic principle of this pattern is based on timeouts.

Retransmission Retransmission is used when a service call might vanish or fail [41, p. 187]. In case the failure lasts for a short period of time, e.g., a network transmission failure, a retransmission results in successful execution. Thus the pattern increases the reliability of the system - especially when unreliable transactions are involved. The pattern is based on timeouts combined with a respective retry strategy.

Caching The cache pattern keeps data retrieved from a slower memory in a faster memory area to allow fast access if an object is accessed twice [40, p. 83]. Therefore, the pattern is used to increase response time and throughput. The benefits are acquired by accepting a larger memory footprint. The basic technique of the pattern uses memory buffers to increase performance.

Pessimistic Offline Lock The pessimistic offline lock is a pattern used to control concurrent access to components or resources controlled by components [42, p. 426]. The lock is used to ensure that solely one single thread of execution is able to access the protected resource. Hence, the lock ensures certain safety criteria on the cost of performance as concurrent threads have to wait before they can execute. The basic principle used in the pattern is based on blocking the control flow using the process scheduler.

Unit of Work The unit of work pattern is used to collect a set of sub-transactions in memory until all parts are complete and then commits the whole transaction by accessing the database only a short time [42, p. 184]. Like the cache pattern there is a trade-off between memory consumption and efficiency. As in the caching pattern the basic idea is to use a memory buffer.

4.3 Classification of Patterns

The collection of patterns does not claim to be complete, there are more patterns which we could look at. We introduced it to show that there are a lot of patterns which can be used to adapt components - mostly in a way which is not producing hand written glue code. In the table in figure 4 we show which patterns can be used to solve problems of the introduced mismatch classes.

	Technical	Signatures	Protocols	Concepts	Quality Attributes
Adapter		✔	✔	✔	✔
Decorator				✔	
Interceptor				✔	✔
Wrapper Facade		✔		✔	
Bridge	✔				
Microkernel	✔				
Mediator			✔		
Proxy					✔
Replication					✔
Process Pair					✔
Retransmission					✔
Caching					✔
Pessimistic Lock					✔
Unit of Work					✔

Fig. 4. A classification of Patterns and Mismatches

5 Using Patterns to Eliminate Component Mismatches

After introducing a set of patterns in the previous section, we will now discuss how to use the patterns in a software engineering process. First, we will introduce a generic process which is supposed to serve as a guideline for adaptation. We will illustrate its usage by giving an example of a functional- and an extra-functional adaptation. In particular, we will show an application of the Adapter/Wrapper pattern and of the Caching pattern.

The process of adapting components in order to construct trustworthy component assemblies using software engineering consists of the following steps:

1. *Detect mismatches*: First the mismatch between the required and provided interface has to be detected. As stated above, this directly depends on the specifications available, i.e., if no protocol specification is available then we can not detect protocol mismatches.
2. *Select measures to overcome the mismatch*: Second, we select from a set of established methods the one which is known to solve the specific mismatch. Note, that this choice also depends on the specifications available as some patterns can only be distinguished by examining subtle differences in the target setting (as already mentioned in section 4). This can sometimes require semantic information which is hard to analyze automatically. It is therefore necessary in many cases to leave the final choice to the developer. Nevertheless, it is possible to filter unsuitable patterns out in advance.
3. *Configure the measure*: Often the method or pattern selected can be fine-tuned as patterns are described since *abstract* solutions to problems. Thereby, we can for instance utilize the specifications and query the developer for additional input. If the specification is complete the solution to the mismatch problem is analyzed.
4. *Predict the impact*: After determining the solution of the problem we predict the impact of the solution on our setting. This is common in other engineering disciplines.
5. *Implement and test the solution*: If the prediction indicates that the mismatch is fixed, the solution is implemented, either by systematic construction or by using generative technologies.

5.1 Adapting Functional Mismatches with the Adapter Pattern

This section shows how functional adaptation can be implemented by utilizing the Adapter/Wrapper pattern [37, p. 139]. As shown in the table in figure 4, this pattern might be used to repair syntax, protocol and semantics mismatches.

The Adapter pattern (also known as Wrapper pattern) maps the interface of a component onto another interface expected by its clients. The Adapter lets components work together that could not otherwise because of incompatible interfaces. The participants in the "schema" of this pattern are: (i) the existing component interface that needs to be adapted, usually denoted as *Adaptee*; (ii) *Target* is the interface required by a client component and it is not compatible to Adaptee; (iii) *Client* denotes any client whose

required interface is compatible to Target and (iv) *Adapter*, which is the component responsible for making *Adaptee* compatible to *Target*.

Here, we discuss an example of a possible application of the Adapter pattern seen as a means to overcome only protocol mismatches. Let us suppose that we want to assemble a component-based cooling water pipe management system that collects and correlates data about the amount of water that flows in different water pipes. The water pipes are placed in two different zones, denoted by *P* and *S*, and they transport water that has to be used to cool industrial machinery. The system we want to assemble is a client-server one. The zones *P* and *S* have to be monitored by a server component denoted as *Server*. *Server* allows the access to a collection of data related to the water pipes it monitors. It provides an interface denoted as *IServer*. Since some of the water pipes do not include a *Programmable Logic Controller* (PLC) system, *Server* cannot always automatically obtain the data related to the water that flows in those water pipes. Therefore, *IServer* exports the methods *PCheckOut* and *SCheckOut* to get an exclusive access to the data collection related to the water which flows in the pipes. This allows a client to: (i) read the data automatically stored by the server and (ii) manually update the report related to the water which flows in the pipes that are not monitored by a PLC. Correspondingly, *IServer* exports also the methods *PCheckIn* and *SCheckIn* to both publish the updates made on the data collection and release the access gained to it. We want to assemble the discussed client-server system formed by the following selected components: *Server* and one client denoted as *Client*. The interface required by *Client* is compatible to *IServer* at level of both signature and semantics.

According to step 1 of the presented process, we need to be able to detect possible protocol mismatches. These days, we can utilize UML2 *Sequence Diagrams* and *Interaction Overview Diagrams* (i.e., the UML2 *Interaction Diagrams* suite) to extend the IDL specification of a component interface for including information related to the component interaction protocol. UML2 sequence diagrams are for describing a single execution scenario of a component or a system; UML2 interaction overview diagrams can be used to compose all the specified component/system execution scenarios into execution flows to indicate how each scenario fit together different ones during the overall execution of the component/system (see Figure 5).

From the UML2 specification shown in Figure 5, it is possible to check automatically that the interaction protocols expected by *Server* and *Client* mismatch. That is, the selected server component forces its clients to always access to the data collections related to the zone *P* and *S* subsequently and in any possible order, before releasing the access gained for both of them. Instead, the selected client component gains the access and releases it for the data collections related to the zone *P* and *S* separately. This protocol mismatch leads to a deadlock.

According to step 2 of our proposed engineering approach to component adaptation, we have to choose the right type of measure to solve the problem. We decide to deploy an Adapter/Wrapper component to force a "check-out" of the data collection related to the zone *S* (*P*) after the client has performed a *PCheckOut* (*SCheckOut*) method call. The release of the gained access is handled analogously. In doing so, the interaction protocol of *Client* is enhanced in order to match the interaction protocol of *Server* (i.e., to avoid the deadlock). This adaptation strategy can be automatically derived by a tool

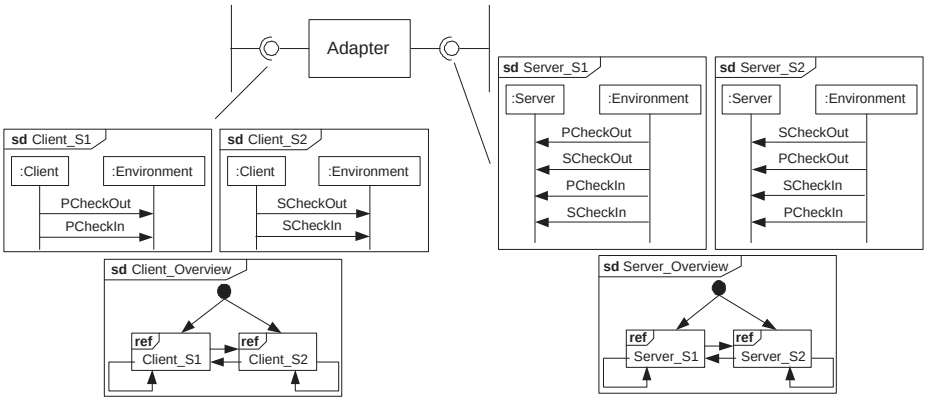


Fig. 5. An example of UML2 Interaction Diagrams specification to detect protocol mismatches

that - by exploiting the UML2 XMI - is able to take in input an XML representation of the UML2 interaction diagrams specification of *Server* and *Client*. This tool might elaborate - in some way - this specification and produce the adaptation strategy that must be implemented by the Adapter. A similar approach can be found in [14].

In the third step of our process we have to customize the pattern to our needs (i.e., protocol adaptation purposes) and choose the right variant of it. We plan to implement the pattern according to Figure 6 depicting the overall structure of our realization.

The following are the participants to the Adapter patter applied to bridge protocol mismatches: (i) **Target Protocol** which is the protocol required by a client component (i.e., the interaction protocol of *Client*); (ii) **Client** which is a component whose protocol is compatible to the Target Protocol (i.e., *Client*); (iii) the **Adapter** which is the component responsible for making an existing protocol compatible to the Target Protocol; and (iv) the **Adaptee Protocol** which is the existing protocol (i.e., the interaction protocol of *Server*). In the figure we also show a portion of the code implementing the method *PCheckOut* as provided by the Adapter component. *SCheckOut*, *PCheckIn* and *SCheckIn* are implemented analogously. This code reflect the adaptation strategy discussed above.

In the next step, in order to make it an engineering process, we predict the impact of the deployed Adapter in terms of checking whether the detected protocol mismatch has been solved or not. To be able to do so, we do not need any further information beyond the UML2 Interaction Diagrams specification of *Client* and *Server* and the underlined structure of the Adapter component. In fact, from this kind of specification, it is possible to automatically derive a process algebra notation e.g., FSP notation [43], of the interaction behavior of *Client*, *Server* and of the Adapter component. FSP notation might be a useful formalism to check automatically if the insertion of the Adapter in the system will avoid the detected protocol mismatch. In the literature, there are more functional analysis tools that support FSP as input language.

One of these tools is LTSA (*Labeled Transition System Analyser*) [43]. LTSA is a plugin-based verification tool for concurrent systems. It checks automatically that the

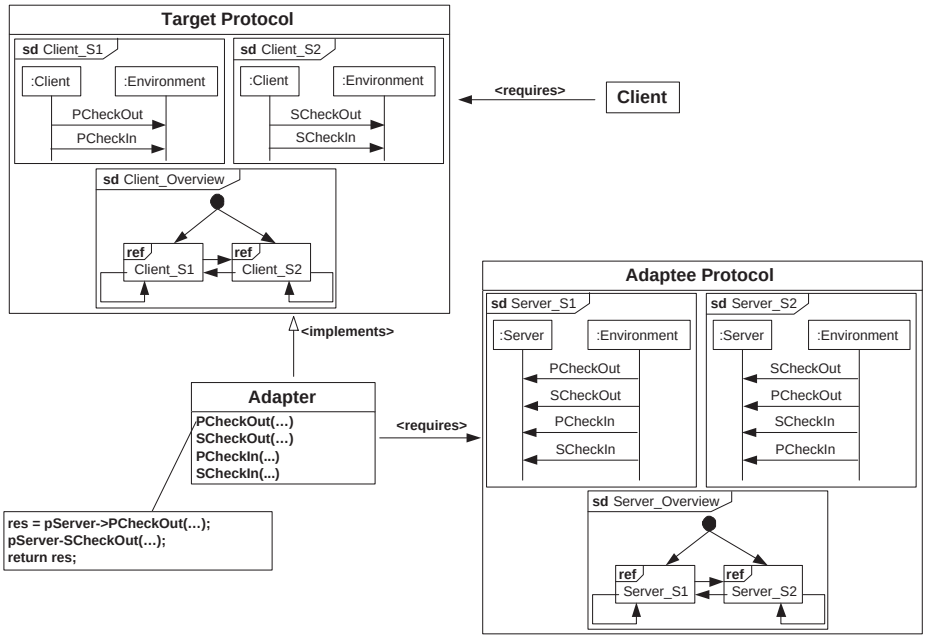


Fig. 6. Overall structure of the Adapter/Wrapper pattern to avoid protocol mismatches

specification of a concurrent system satisfies required properties of its behavior such as deadlock-freeness. Thus, by integrating our process with such tools we can predict whether the detected protocol mismatch will be solved by the Adapter component or not. Moreover - since the Adapter also changes extra-functional properties of the system, e.g., by slowing down accesses because of the injected method calls - we should predict the impact of the Adapter on the performance of method calls. In the next subsection it is very clearly explained how to predict it by using an usage profile of an adapted service. Here, we simply note that performance of method calls should decrease but very little because the Adapter adds only a lightweight extra-level of indirectness.

In the final step, the adapter is built by exploiting the information contained in its pattern description. Depending on the complexity of the Adapter, this can be done either mechanically by a tool or by the developers. Once the Adapter is deployed, tests that validate both the results of the prediction and the adapter correctness are performed.

5.2 Adapting Extra-Functional Mismatches with the Caching Pattern

In the following we show how extra-functional adaptation can be achieved by employing the Caching pattern [40, p. 83]. A cache is used if a service needs some kind of resource whose acquisition is time consuming and the resource is not expected to change frequently but to be used often. The idea is to acquire the resource and to put it in the cache afterwards. The resource can be retrieved faster from the cache than re-acquiring it again. This is often done by utilizing additional memory to store the resource for

faster retrieval. Hence, a trade-off is established between retrieval time and memory consumption. If the resource is needed again, it is retrieved from the cache. Often a validation check is performed in advance to test whether the cached resource is still up to date. Additionally, if the resource is altered by its usage we have to ensure consistency with the non-cached original object. This can be done by either storing it at its original location directly when the resource is altered (write-through-strategy). The other option is that the resource gets stored as soon as it gets evicted from the cache.

According to step 1 of the presented process, we need to be able to detect the mismatching response times. These days, we can utilize QML [44] specifications of the respective interfaces for this task. For example, let's assume an average response time of 3000ms is needed and an average response time of 6000ms is provided for service under investigation (see figure 7).

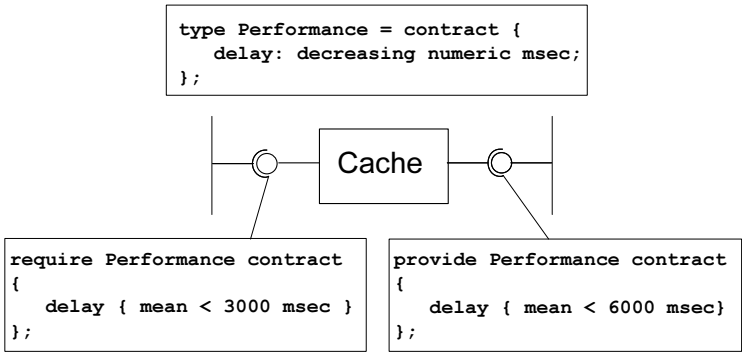


Fig. 7. An example QML specification to detect a QoS mismatch

Additionally, we know that the required service processes requests to a static database. Therefore, we can consider the database table rows in the above stated sense. The database is not updated frequently, so caching the database query results will improve the average response time. Note, that we also need to know that the service fulfills these prerequisites of the cache pattern. It is to automatically determine if the prerequisites are fulfilled as service specifications often state nothing about the resource usage of the specified service.

Second, we have to choose the right type of measure to solve the problem. We decide to deploy a cache to speed up an encapsulated resource access in the component being used. In doing so, the response time is decreased and the components can interoperate as desired.

In the third step we have to customize the pattern to our needs and choose the right variant of the pattern. Referring to the description in [40] we have to

- Select resources: The database query results
- Decide on an eviction strategy: Here we can choose between well-known types like least recently used (LRU), first in - first out (FIFO), and so on.

– Ensure consistency: We need a consistency manager whose task is to invalidate cache entries as soon as the master copy is changed. In the given database scenario it makes no sense to omit that part.
– Determine cache size: How much memory the cache is going to use. Most likely this is specified in number of cacheable resource units.

We plan to implement the pattern according to the following figure depicting the static structure of our realization (see figure 8).

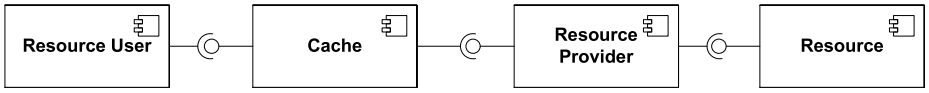


Fig. 8. The cache pattern implemented with components

In the next step, in order to make it an engineering process, we predict the impact of the deployed cache. To be able to do so, the usage profile of the adapted service is needed, as the performance of a cache depends on it. The usage profile information needed in this context, is the (estimated) frequency and type of requests. Together with the decisions taken in the previous step a specialized prediction model for the cache impact can be applied and the result is compared to the requirements. This step is not well researched so that today we often neglect the step and trust on the experience of the deployer. Future work might come up with more prediction models to enable the engineering process as depicted here. To continue, let us assume, that the result is 2500ms and thus, the mismatch is resolved.

In the final step the adapter is finally constructed or generated by using the instructions given in the respective pattern description. Once the adapter is deployed, we perform tests to ensure that the predictions have been right and that everything works as expected.

6 Related Work

Even though Component-Based Software Engineering was first introduced in 1968 [45], developing systematic approaches to adaptation of components in order to resolve interoperability problems is still a field of active research. Many papers are based on the work done by Yellin and Strom [2, 46], who introduced an algorithm for the (semi-) automatic generation of adapters using protocol information and an external adapter specification. Bracciali et al. propose the use of some kind of process calculus to enhance this process and generate adapters using PROLOG [47].

Schmidt and Reussner present adapters for merging and splitting interface protocols and for a certain class of protocol interoperability problems [30]. Besides adapter generation, Reussner's parametrized contracts also represent a mechanism for automated component adaptation [48]. Additionally, Kent et al. [49] propose a mechanism for the handling of concurrent access to a software component not built for such environments.

Vanderperren et al. have developed a tool called PaCoSuite for the visual assembly of components and adapters. The tool is capable of (semi-)automatic adapter generation using signature and protocol information [50]. Gschwind uses a repository of adapters to dynamically select a fitting adapter [51]. Min et al. present an approach called Smart Connectors which allows the construction of adapters based on the provided and required interface of the components to connect [27].

Passerone, de Alfaro and Henzinger developed a game-theoretical approach to find out whether incompatible component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements [4]. This approach is able to automatically synthesize the converter. Their approach can only be applied to a restricted class of component mismatches (protocols and interaction). In fact, they are only able to restrict the system's behavior to a subset of desired ones and, for example, they are not able to augment the system's behavior to introduce more sophisticated interactions among components.

In [10], Garlan et al. have shown how to use formalized protocol transformations to augment the interaction behavior of a set of components. The key result was the formalization of a useful set of interaction protocol enhancements. Each enhancement is obtained by composing wrappers. This approach characterizes wrappers as modular protocol transformations. The basic idea is to use wrappers to introduce more sophisticated interactions among components. The goal is to alter the behavior of a component with respect to the other components in the system, without actually modifying the component or the infrastructure itself. While this approach deals with the problem of enhancing component interactions, it does not provide a support for wrapper generation.

A common terminology for the Quality of Service prediction of systems which are being assembled from components is proposed in [52]. A concrete methodology for predicting extra-functional properties of .NET assemblies is presented in [53]. None of these approaches, however, provides a specialized method for including adapters in their predictions. Engineering Quality of Service guarantees in the context of distributed systems is the main topic of [54].

An overview on adaptation mechanisms including non-automated approaches can be found in [55] (such as delegation, wrappers [37], superimposition [56], metaprogramming (e.g., [57])). Both works also contain a general discussion of requirements for component adaptation mechanisms. Not all of these approaches can be seen as adapters as defined in this paper. But some of the concepts presented can be implemented in adapters as shown here.

7 Conclusions and Future Directions

This paper introduces an engineering approach to software component adaptation. We define adaptation in terms of dealing with component mismatches, introduce the concept of component mismatch, and present a taxonomy to distinguish different types of component mismatches. Afterwards, we discuss a selection of adaptation patterns that can be used to eliminate the different mismatch types. The main contribution of the paper is a presentation of how these patterns can be used during the component adap-

tation process. The presented approach is demonstrated by both a functional and an extra-functional adaptation example.

Future research is directed towards exploring additional interface description languages which enable the efficient checking of the introduced mismatch types. On the basis of the available specification data, algorithms have to be developed to statically check for the identified component mismatch types during a compatibility test. Further on, existing prediction methods, which are based on the available component data, have to be improved to include adaptation and its impact on extra-functional system properties. In doing so, measures have to be developed that assess the impact on the extra-functional properties of systems when applying specific patterns to identified adaptation problems.

The application of generative techniques or concepts of Model-Driven Architecture (MDA) to construct the appropriate adapters is another strand of ongoing work. In this context, dependable composition of adapters and generation of adapters from the specification of the integrated system and the components are emerging areas of research. Finally, to achieve a fully-fledged engineering approach to component adaptation, further effort will be required to develop suitable tools that are capable of supporting the selection of pattern(s) which can be applied to solve specific mismatch types (viz., step 2 of the process proposed in Sect. 5).

Acknowledgments

The authors would like to thank Viktoria Firus, Gerhard Goos, and Raffaela Mirandola for their valuable and inspiring input during our break-out session at Dagstuhl, which preceded this paper. Steffen Becker is funded by the German Science Foundation in the DFG-Palladio project. Alexander Romanovsky is supported by the IST FP6 Project on Rigorous Open Development Environment for Complex Systems (RODIN).

References

1. Balemi, S., Hoffmann, G.J., Gyugyi, P., Wong-Toi, H., Franklin, G.F.: Supervisory Control of a Rapid Thermal Multiprocessor. IEEE Transactions on Automatic Control **38** (1993) 1040–1059
2. Yellin, D., Strom, R.: Protocol Specifications and Component Adaptors. ACM Transactions on Programming Languages and Systems **19** (1997) 292–333
3. de Alfaro, L., Henzinger, T.A.: Interface Automata. In Gruhn, V., ed.: Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01). Volume 26, 5 of ACM SIGSOFT Software Engineering Notes., New York, ACM Press (2001) 109–120
4. Passerone, R., de Alfaro, L., Henzinger, T., Sangiovanni-Vincentelli, A.L.: Convertibility Verification and Converter Synthesis: Two Faces of the Same Coin. In: Proceedings of the International Conference on Computer Aided Design (ICCAD'02). (2002)
5. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption Generation for Software Component Verification. In IEEE, ed.: 17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK, Los Alamitos, CA, IEEE Computer Society (2002) 3–12

6. Inverardi, P., Tivoli, M.: Software Architecture for Correct Components Assembly. In Bernardo, M., Inverardi, P., eds.: Formal Methods for Software Architectures, Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003, Advanced Lectures. Volume 2804 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2003) 92–121
7. Inverardi, P., Tivoli, M.: Failure-Free Connector Synthesis for Correct Components Assembly. In: Proceedings of Specification and Verification of Component-Based Systems (SAVCBS'03). (2003)
8. Tivoli, M., Inverardi, P., Presutti, V., Forghieri, A., Sebastianis, M.: Correct Components Assembly for a Product Data Management Cooperative System. In Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C., eds.: Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings. Volume 3054 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2004) 84–99
9. Brogi, A., Canal, C., Pimentel, E.: Behavioural Types and Component Adaptation. In Rattray, C., Maharaj, S., Shankland, C., eds.: Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings. Volume 3116 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2004) 42–56
10. Spitznagel, B., Garlan, D.: A Compositional Formalization of Connector Wrappers. In IEEE, ed.: Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA, Los Alamitos, CA, IEEE Computer Society (2003) 374–384
11. Tivoli, M., Garlan, D.: Coordinator Synthesis for Reliability Enhancement in Component-Based Systems. Technical report, Carnegie Mellon University (2004)
12. Autili, M., Inverardi, P., Tivoli, M., Garlan, D.: Synthesis of 'Correct' Adaptors for Protocol Enhancement in Component-Based Systems. In: Proceedings of Specification and Verification of Component-Based Systems (SAVCBS'04). (2004)
13. Autili, M., Inverardi, P., Tivoli, M.: Automatic Adaptor Synthesis for Protocol Transformation. In: Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04). (2004)
14. Tivoli, M., Autili, M.: SYNTHESIS: A Tool for Synthesizing 'Correct' and Protocol-Enhanced Adaptors. To appear on L'Objet journal, http://www.di.univaq.it/tivoli/LastSynthesis.pdf (2005)
15. Garlan, D., Allan, R., Ockerbloom, J.: Architectural Mismatch: Why Reuse Is So Hard. IEEE Software **12** (1995) 17–26
16. Mili, H., Mili, F., Mili, A.: Reusing Software: Issues and Research Directions. IEEE Transactions on Software Engineering **21** (1995) 528–561
17. de Lemos, R., Gacek, C., Romanovsky, A.: Architectural Mismatch Tolerance. In de Lemos, R., Gacek, C., Romanovsky, A., eds.: Architecting Dependable Systems. Volume 2677 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2003) 175–194
18. Canal, C., Murillo, J.M., Poizat, P.: Coordination and Adaptation Techniques for Software Entities. In Malenfant, J., Østvold, B.M., eds.: Object-Oriented Technology: ECOOP 2004 Workshop Reader, ECOOP 2004 Workshops, Oslo, Norway, June 14-18, 2004, Final Reports. Volume 3344 of Lecture Notes in Computer Science., Springer (2005) 133–147
19. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. 2 edn. ACM Press and Addison-Wesley, New York, NY (2002)
20. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, Reading, MA, USA (1999)
21. Becker, S., Overhage, S., Reussner, R.: Classifying Software Component Interoperability Errors to Support Component Adaption. In Crnkovic, I., Stafford, J.A., Schmidt, H.W.,

Wallnau, K.C., eds.: Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings. Volume 3054 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2004) 68–83
22. Yakimovich, D., Travassos, G., Basili, V.: A classification of software components incompatibilities for COTS integration. Technical report, Software Engineering Laboratory Workshop, NASA/Goddard Space Flight Center, Greenbelt, Maryland (1999)
23. Overhage, S., Thomas, P.: WS-Specification: Specifying Web Services Using UDDI Improvements. In Chaudhri, A.B., Jeckle, M., Rahm, E., Unland, R., eds.: Web, Web Services, and Database Systems. NODe 2002 Web- and Database-Related Workshops. Volume 2593 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2003) 100–118
24. Beugnard, A., Jezequel, J.M., Plouzeau, N., Watkins, D.: Making Components Contract Aware. IEEE Computer **32** (1999) 38–45
25. Overhage, S.: UnSCom: A Standardized Framework for the Specification of Software Components. In Weske, M., Liggesmeyer, P., eds.: Object-Oriented and Internet-Based Technologies, 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, NODe 2004, Proceedings. Volume 3263 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2004) 169–184
26. Zaremski, A.M., Wing, J.M.: Signature Matching: A Tool for Using Software Libraries. ACM Transactions on Software Engineering and Methodology **4** (1995) 146–170
27. Min, H.G., Choi, S.W., Kim, S.D.: Using Smart Connectors to Resolve Partial Matching Problems in COTS Component Acquisition. In Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C., eds.: Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings. Volume 3054 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (2004) 40–47
28. Meyer, B.: Object-Oriented Software Construction. 2. edn. Prentice Hall, Englewood Cliffs, NJ (1997)
29. Zaremski, A.M., Wing, J.M.: Specification Matching of Software Components. ACM Transactions on Software Engineering and Methodology **6** (1997) 333–369
30. Schmidt, H.W., Reussner, R.H.: Generating Adapters for Concurrent Component Protocol Synchronisation. In: Proceedings of the Fifth IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems. (2002)
31. ISO/IEC: Software Engineering - Product Quality - Quality Model. ISO Standard 9126-1, International Organization for Standardization (2001)
32. ISO/IEC: Software Engineering - Product Quality - External Metrics. ISO Standard 9126-2, International Organization for Standardization (2003)
33. Horwich, P.: Wittgenstein and Kripke on the Nature of Meaning. Mind and Language **5** (1990) 105–121
34. Paolucci, M., Kawamura, T., Payne, T., Sycara, K.: Semantic Matchmaking of Web Services Capabilities. In Horrocks, I., Hendler, J., eds.: First International Semantic Web Conference on The Semantic Web. Volume 2342 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2002) 333–347
35. Noy, N.F.: Tools for Mapping and Merging Ontologies. In Staab, S., Studer, R., eds.: Handbook on Ontologies. Springer, Berlin, Heidelberg (2004) 365–384
36. Noy, N.F.: Semantic Integration: A Survey Of Ontology-Based Approaches. SIGMOD Record **33** (2004) 65–70
37. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, USA (1995)
38. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture – A System of Patterns. Wiley & Sons, New York, NY, USA (1996)

39. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture – Volume 2 – Patterns for Concurrent and Networked Objects. Wiley & Sons, New York, NY, USA (2000)
40. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture: Patterns for Distributed Services and Components. John Wiley and Sons Ltd (2004)
41. Grand, M.: Java Enterprise Design Patterns: Patterns in Java (Patterns in Java). John Wiley & Sons (2002)
42. Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., Stafford, R.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (2002)
43. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley and Sons (1999)
44. Frølund, S., Koistinen, J.: Quality-of-Service Specification in Distributed Object Systems. Technical Report HPL-98-159, Hewlett Packard, Software Technology Laboratory (1998)
45. McIlroy, M.D.: "Mass Produced" Software Components. In Naur, P., Randell, B., eds.: Software Engineering, Brussels, Scientific Affairs Division, NATO (1969) 138–155 Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.
46. Yellin, D., Strom, R.: Interfaces, Protocols and the Semiautomatic Construction of Software Adaptors. In: Proceedings of the 9th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-94). Volume 29, 10 of ACM Sigplan Notices. (1994) 176–190
47. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software **74** (2005) 45–54
48. Reussner, R.H.: Automatic Component Protocol Adaptation with the CoCoNut Tool Suite. Future Generation Computer Systems **19** (2003) 627–639
49. Kent, S.D., Ho-Stuart, C., Roe, P.: Negotiable Interfaces for Components. In Reussner, R.H., Poernomo, I.H., Grundy, J.C., eds.: Proceedings of the Fourth Australasian Workshop on Software and Systems Architectures, Melbourne, Australia, DSTC (2002)
50. Vanderperren, W., Wydaeghe, B.: Towards a New Component Composition Process. In: Proceedings of ECBS 2001 Int Conf, Washington, USA. (2001) 322 – 331
51. Gschwind, T.: Adaptation and Composition Techniques for Component-Based Software Engineering. PhD thesis, Technische Universität Wien (2002)
52. Hissam, S.A., Moreno, G.A., Stafford, J.A., Wallnau, K.C.: Packaging Predictable Assembly. In Bishop, J.M., ed.: Component Deployment, IFIP/ACM Working Conference, CD 2002, Berlin, Germany, June 20-21, 2002, Proceedings. Volume 2370 of Lecture Notes in Computer Science., Springer (2002) 108–124
53. Dumitrascu, N., Murphy, S., Murphy, L.: A Methodology for Predicting the Performance of Component-Based Applications. In Weck, W., Bosch, J., Szyperski, C., eds.: Proceedings of the Eighth International Workshop on Component-Oriented Programming (WCOP'03). (2003)
54. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo (2001)
55. Bosch, J.: Design and Use of Software Architectures – Adopting and evolving a product-line approach. Addison-Wesley, Reading, MA, USA (2000)
56. Bosch, J.: Composition through Superimposition. In Weck, W., Bosch, J., Szyperski, C., eds.: Proceedings of the First International Workshop on Component-Oriented Programming (WCOP'96), Turku Centre for Computer Science (1996)
57. Kiczales, G.: Aspect-oriented programming. ACM Computing Surveys **28** (1996) 154–154