Analysis of Compositional Conflicts in Component-Based Systems

vorgelegt von
Diplom - Informatiker

Andreas Leicher

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zu Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Hartmut Ehrig
Berichter:	Prof. Dr. Herbert Weber
Berichter:	Prof. Dr. Frantisek Plasil

Tag der wissenschaftlichen Aussprache: 21. September 2005

Berlin, 11. November 2005
D83

Zusammenfassung

Ein Ziel der komponentenbasierten Softwareentwicklung besteht in der Wiederverwendung bereits entwickelter Komponenten zur Verbesserung der Qualität und zur Verminderung der Kosten des Softwareentwicklungsprozesses. Die Wiederverwendung von bestehenden Komponenten erfordert jedoch die sorgfältige Integration in das zu entwickelnde System und wird erschwert durch unterschiedliche Middlewaretechnologien, unterschiedliche Kommunikations- und Interoperationsmechanismen dieser Technologien sowie durch unterschiedliche und zum Teil unvollständige Komponentenspezifikationen.

Das grundlegende Ziel dieser Arbeit ist die Unterstützung der Komponentenintegration durch eine ausführliche Konfliktanalyse, welche bestehende Inkompatibilitäten von zu integrierenden Komponenten aufzeigt. Dazu wird ein Framework bereitgestellt, welches es erlaubt, Komponenten unterschiedlicher Technologien im Rahmen eines UML-basierten Entwicklungsprozesses zu prüfen und die (teil-) automatische Generierung von Konnektoren vorzubereiten.

Im Gegensatz zu bisherigen Ansätzen ermöglicht dieses Framework die Prüfung von Komponenten unterschiedlicher Middlewaretechnologien in Hinblick auf strukturelle, semantische und kommunikative Differenzen. Insbesondere die Einbeziehung von Kommunikationseigenschaften, welche die von den beteiligten Komponenten verwendeten Kommunikationsmechanismen der zugrundeliegenden Middlewaretechnologien beschreiben, wird zur Zeit von keinem anderen Integrationsframework angeboten.

Das vorgestellte Framework basiert auf einem modellzentrierten Ansatz im Rahmen der Model Driven Architecture (MDA). Dies bedeutet, dass sowohl Komponentenspezifikationen auf verschiedenen Ebenen verwaltet werden können als auch dass Modelltransformationen unterstützt werden. Die Konfliktanalyse basiert auf einem kanonischen plattformunabhängigen Komponentenmodell, welches von Plattformspezifika abstrahiert. Komponenten, die auf Grundlage einer speziellen Middlewaretechnologie definiert wurden, werden durch Modelltransformationen in das kanonische Modell abstrahiert. Umgekehrt können jedoch auch plattformunabhängige Komponentenspezifikationen in Spezifikationen spezieller Middlewaretechnologien spezialisiert werden. Als Besonderheit unterstützt das Framework dabei parametrisierbare Modelltransformationen, so dass sich je nach Nutzervorstellungen optimale Transformationen durchführen lassen.

Um die Verwendung des Frameworks in unterschiedlichen Anwendungsszenarien und -domänen zu ermöglichen, wurde es möglichst flexibel definiert. Somit können sowohl unterschiedliche Middlewaretechnologien und Typsysteme als auch unterschiedliche Spezifikationssprachen in das Framework integriert werden. Zudem können zusätzliche Daten als RDF Statements an beliebige Elemente von Komponenten notiert und über zusätzliche Analyseverfahren ausgewertet werden.

Abstract

Component Based Software Engineering is an emerging discipline that aims at improving software development by means of artifact reuse within a systematically applied construction process. The idea of reuse involves integrating components rather than reinventing and reimplementing existing artifacts. Unfortunately, reuse of existing components is a complex undertaking because of different technologies, different communication forms, and incomplete component specifications.

The objective of this work concerns the improvement of component integration by providing extended conflict analysis capabilities, which help to identify mismatches between components that hinder a direct integration. The work defines a Framework for Component Conflict Analysis and Composition that serves as a basis of component integration. The framework is able to handle components of different technologies and it is compatible with a UML-based development process. As a result, it provides the inevitable preparatory work that is required for a semi-automatic connector generation.

Contrary to existing approaches, the framework allows analyzing components of different middleware technologies by checking compatibility between their type, behavior, and communication related property specifications. Especially, analysis of required and provided communication properties, which takes the middleware context of components into account is not available in comparable integration frameworks.

Components of different technologies cannot be directly compared as they are defined relative to different programming languages, type systems, and middleware technologies. Therefore, the framework is based on a model centric approach within the scope of the Model-Driven Architecture (MDA). Consequently, the framework distinguishes platform independent and platform dependent component specifications. Conflict analysis is based on a platform independent canonical model that represents the least common denominator of components in middleware technologies. Furthermore, the framework provides model transformations that allow abstraction and refinement of components between different abstraction levels. Thereby, the framework supports parametric model transformations as a special feature, which take user requirements into account and result in optimal translations.

In order to gain a solution, which is adaptable to different situations and application domains, the framework offers a highly flexible architecture. Consequently, we can integrate new middleware technologies, their type systems as well as additional specification languages into the framework.

Acknowledgments

First of all, I'd like to thank my family. Without their support and especially without the help of Silke I hadn't been able to finish this work.

Special thanks goes to Susanne Busse, Uli Kriegel, Ralf Kutsche, Martin Große-Rhode, Jörn Guy Süß, and Prof. Weber for their support, critique, and suggestions as well as for all the discussions regarding my approach. With Uli Kriegel, I mainly discussed platform related issues, whereas Martin Große-Rhode and Ralf Kutsche helped me with the theoretical aspects. With Susanne Busse and Jörn Guy Süß, I mainly discussed issues in the areas of UML and MDA.

Further, I thank Andreas Billig, who convinced me to use an object-oriented reasoning language and who provided me with the ODIS environment that I used throughout the work for model transformations and for conflict analysis.

The members of the Distributed Systems Research Group from Charles University also offered me a lot of helpful critique related to aspects of the core model and behavioral analysis. Especially, I thank Prof. Plasil, Tomas Bures, Jiri Adamek and Vladimir Mencl.

I also want to mention the fruitful discussions with Jan Gädicke, Felix Schmid, and Yong Liao who helped me to discover how to formalize the communication taxonomy.

Furthermore, I thank Thomas Brehm and Cedavis Inc. for their support regarding a case study of their system.

At last, I'd like to thank the members of the CIS group, especially Claudia Gantzer and Lutz Friedel for their help and the working atmosphere.

Contents

Part I

Foundations of Conflict Analysis

Chapter 1

Introduction

1.1 Motivation

Component Based Software Engineering is an emerging discipline that aims at improving software development by means of artifact reuse within a systematically applied construction process. The idea of reuse involves integrating components rather than reinventing and reimplemening existing artifacts.

We restrict the focus of this work to the technical aspects of component integration. Precisely, we consider the question whether two components are compatible by analyzing several relationships between their specifications. Two components are compatible if they can communicate without encountering any mismatches regarding types, behavior and semantics. The relationships we use for analysis originate in syntactic and semantic categories. Thereby, syntactic aspects are defined via type systems and subtype relationships, whereas semantic specifications are further distinguished into pre- and post-conditions, behavioral protocols and property-based descriptions of the semantics (functionality) and the communication requirements of components.

We further distinguish component integration by two steps: conflict analysis and connector generation. Conflict analysis identifies mismatches between components. Conflicts are differences between component descriptions that hinder a direct integration. A connector resolves the identified mismatches and therefore integrates the components. This work concentrates on the first integration step. The second step is considered insofar as connector generation is prepared and the approaches of conflict resolution that result in the specification of connector parts are discussed.

Unfortunately, conflict analysis as needed for component integration is difficult to achieve for several reasons:

Different interpretations of the term component. Similar to computer literature, where several definitions of the term component have been proposed, technologies also diverge in their handling of components. For example, entities such as programs, COM components, Java classes, or Enterprise JavaBeans can be interpreted as components. Each of these entities provides different structures and features as well as imposes different requirements on the environment. Therefore, features defined for a component in a particular technology are not necessarily present in a component specification of another technology. Therefore, we can neither directly compare components of different technologies nor decide if a composition will fail.

Different technologies hinder direct comparison. Components of different technologies cannot be directly compared, as they are defined relative to different programming languages, type systems, and middleware technologies. Furthermore, constraints are often defined in different specification languages, which disallow direct checks.

Different communication forms. Middleware technologies define similar but slightly different communication mechanisms. Application components depend on the given mechanisms of their underlying technology. A seamless interoperation requires the same communication mechanism used by the involved application components. Consequently, an integration of components of different technologies requires the analysis of communication mechanisms. However, respective descriptions are often not available.

Incomplete component specifications. A sound integration requires the identification of all conflicts between two or more components. Unfortunately, this is often a problem because of under-specification of components, lack of formal methods, and unknown communication properties of the components and the underlying technology.

Lack of support for software development. The natural environment to compare the capabilities and requirements of components is provided by modeling tools. Unfortunately, most tools do not support modeling of components of different technologies. Therefore, a comparison cannot directly be integrated in such tools, but must be manually performed by the developer.

1.2 Prerequisites

A prerequisite for compositional analysis consists of defining an appropriate communication model. This model needs to exactly specify the way components communicate.

In general, we perceive a component as an independent entity which provides and requires particular interfaces. These interfaces describe 'all' dependencies of a component. Consequently, each communication is based on the composition of these interfaces. A communication is established by a connector that binds the participating components based on the provided and required interfaces. Figure 1.1 shows an abstract representation of a component composition.

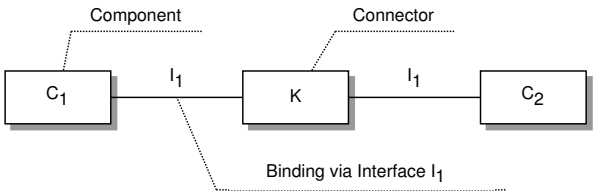

Figure 1.1: A Simple Component Binding

This model is a simplified version of communication models used in software architecture. It abstracts from particular technologies and solely describes communication between abstract entities.

A communicational model that aims to describe technology-specific components needs to take into account the requirements of the underlying technologies. Communication between components invariably takes place in the context of a technology. Each technology is represented by a kind of 'middleware' component, i.e. a binary artifact. Each application component is bound via a precisely defined mechanism to that middleware and cannot be used independently. Figure 1.2 shows a typical mechanism that uses stub and skeleton objects to integrate components with respect to a particular middleware.

In this scenario, middleware plays a dual role: It is at the same time a connector that facilitates the communication and a component that can be physically deployed in an appropriate location.

This indirect interpretation of communication holds true for many middleware technologies. For example, Java RMI provides a transparent distributed communication by only imposing two requirements: A server component must be a subclass of 'java.rmi.UnicastRemoteObject' and the communication interfaces must be lq extended' from 'java.rmi.Remote'[1].

[1] Additional interfaces are required for exceptions, etc.

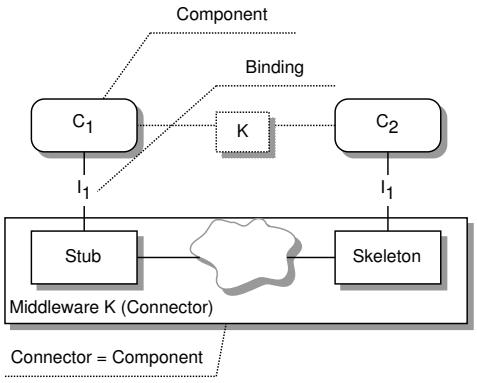

Figure 1.2: Component Binding is Subject to the Underlying Middleware

In principle, communication mechanisms such as procedure calls in Java or C can also be interpreted within this model: A Java procedure call is interpreted by the Java Virtual Machine (VM). Additional actions such as security checks can be transparently performed by the VM. C procedure calls are directly translated into object code. On this level the processor interprets the call. Even here, additional actions such as testing for stack overflows can be performed transparently. Consequently, communication can often be interpreted as dependent from underlying objects. These objects create an indirection layer that hides complexity of communication. In other words, a connection between two application components is essentially a logical connection between these two components. The actual connection is realized by components on a lower level. These components are at the same time connectors on a higher abstraction level $(n + 1)$.

Figure 1.3 shows a generic communication stack that describes a generalization of this principle. Components are placed on different levels relative to their position in a communication. A component on a higher level depends on the components below. It needs to respect the properties and constraints imposed by these components and can only communicate by means of provided communication mechanisms. Alternatively, it can specialize and customize these mechanisms (rarely done by application components).

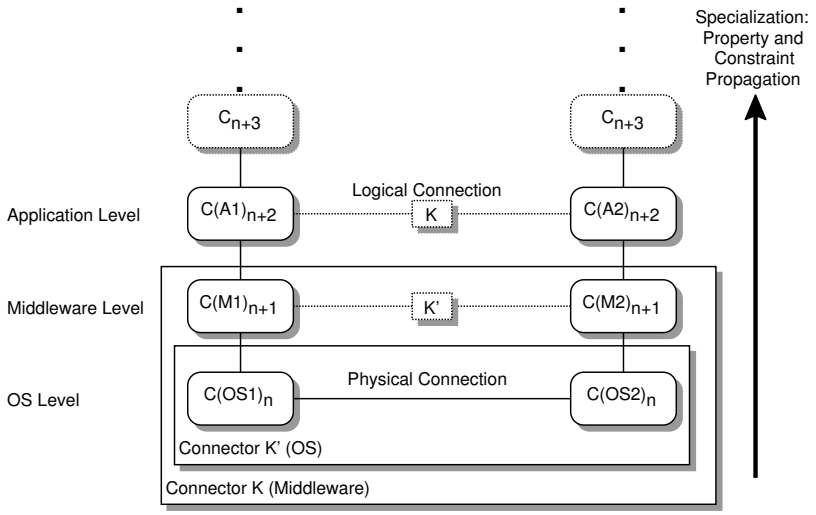

Figure 1.3: Component Communication Stack

Up to now, the communication stack does not give a precise answer to the problem of composing entities originating in different technologies.

In order to compose components from different domains, an interceptor as defined in Section

2.2 needs to be used. The key characteristic of an interceptor consists of the fact that its constituting elements are located in both domains to be connected. We use this feature to define a virtual interceptor domain:

> A virtual interception domain specifies a virtual path between two different domains that establish the basis for interoperability between these domains. The domain can but need not be materialized by physical components.

Figure 1.4 shows the communication stack augmented with a virtual interceptor domain. If components $A1$ and $A2$ are defined in different technologies and these technologies cannot directly interoperate, components $I1$ and $I2$ represent physical components that are defined according to one of the integration patterns as defined in Section 2.2.3. In this case, $I1$ and $I2$ need to rely on the features of the underlying middleware and operating systems to establish an interoperation. If components $A1$ and $A2$ are defined in the same technology, the interception level can either be omitted or be interpreted as consisting of the communication mechanism elements of the middleware M. The presented connector implements an interceptor mechanism on the middleware. However, an interceptor can also be implemented between the middleware and the operating system.

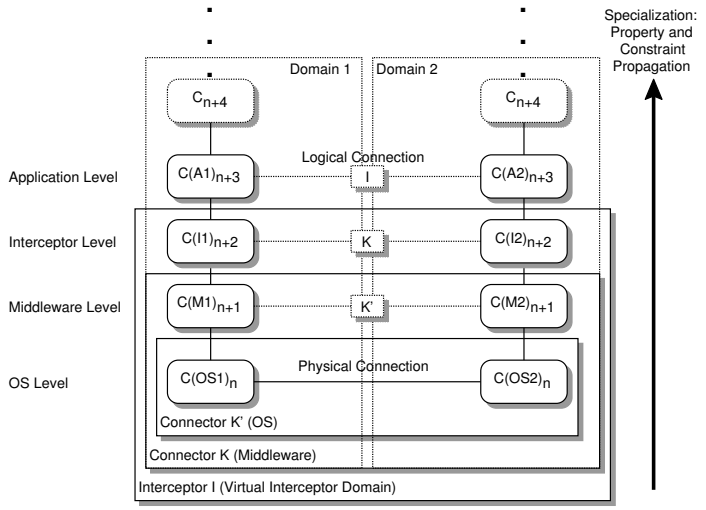

Figure 1.4: Component Communication Stack

In summary, conflict analysis has to investigate the whole underlying communication stack to identify conflicts. It is not sufficient to regard only application components. An analysis needs to comprise properties and restrictions of the components below. A virtual interceptor domain is normally not considered in conflict analysis. It usually represents the solution of an integration of two mismatched components.

1.3 Objectives

The objective of this work is to define a Framework for *Component Conflict Analysis and Composition* that serves as a basis of component integration and thus helps to provide answers for the problems stated above. In doing this, we focus on conflict analysis of the technical aspects of small to medium-sized components. In particular, we pursue the following objectives:

Support for a Canonical Component Model. The framework targets at the technical comparison of components that originate in object-oriented middleware technologies, such as CORBA, COM, .NET, J2EE, Jini, etc. As component specifications of these technologies

differ in form and content, a canonical representation is required that represents the least common denominator of components in these technologies.

Support for Component Abstraction. As existing artifacts are specified in terms of concrete technologies, the framework has to provide transformation functionality to demerge platform specific details of components such as type systems and to create a canonical representation of those in order to create a basis for conflict analysis.

Conflict Analysis identifies mismatches between the technical specifications of two components. The more kinds of conflicts are identified, the more accurate a cost assessment of an integration effort will be. A solution can be generated in the form of a connector which mediates the identified incompatibilities between a number of components. However, the accuracy of a generated connector depends on the precision of conflict identification.

We therefore aim to check for as many conflict categories as possible and additionally to create a highly customizable conflict analysis framework that is adaptable to different requirements and for different technologies. At present, we support compatibility and substitutability analysis of types, behavior and communication properties.

Support for Model Refinement. We aim to support model refinement because it is required in order to generate platform specific connectors. Furthermore, it also realizes a proposed advantage of Model-Driven Development: Reuse of Platform Independent Models (PIMs) by providing transformations into arbitrary technologies.

Integration into Software Development. We believe that a framework for conflict analysis and composition will only be useful in the context of CBSE if it seamlessly integrates with design tools that are used in software development. As UML is the de facto modeling language, we aim to support conflict identification from within UML tools.

A possible application scenario for the framework concerns integration and reuse of existing artifacts. In this sense, components are often termed as Commercials Off-the-Shelf (COTS) that can be obtained from a component market, as it was first proposed by McIlroy [91] in 1968. Unfortunately, reuse of COTS is not easily realized. There are two major problems of COTS integration: The first problem concerns selection and identification of COTS that offer required functionality and quality. The second problem concerns incompatibilities among current technologies that make it difficult to compose COTS. For these reasons, integration approaches normally only consider COTS based on the same technology. Integration of components that originate in different technologies is avoided, as it is a difficult and time-consuming task. The proposed framework targets at the second problem as it mainly provides conflict analysis for COTS integration.

1.4 Approach

We engage component integration and conflict analysis in particular by defining a framework that provides the necessary functionality to satisfy each of the five objectives stated above. The central aspect of the framework consists of a platform independent component model on which conflict analysis is based. The components to be analysed and integrated, however, are specified relative to a platform. The connector that integrates these components also has to be specified according to particular technologies.

Therefore, an integration process as shown in Figure 1.5 consists of several sequential steps:

(1) Extraction of component specifications from artifacts.

In the first step, for each artifact a corresponding model is created. In terms of the MDA, this model is a platform specific model (PSM), as it represents a direct mapping of a technology's relevant properties into a formal (model-based) representation. The OMG already provides UML profiles for several technologies such as a CORBA profile or a profile for entity relationship models.

(2) Abstraction into a canonical representation.

PSM components are difficult to compare if they were defined in different technologies. Therefore, Platform Specific Model (PSM) descriptions must be abstracted to a PIM level. A canonical component model compensates differences in type systems or component structures and provides a uniform view of communication properties.

(3) Conflict analysis.

The framework compares components on the platform independent level. Comparison is based on formally defined relationships: subtypes, simulation, and property convergence. Section 6 describes these relationships in more detail. As a result, conflicts are generated to document incompatibilities - mismatches - between components.

(4) Conflict resolution and connector generation.

Based on the conflicts, the developer can semi-automatically define a connector between two component types. Although not in the scope of this work, the framework can assist in the process of connector generation by offering predefined mapping functions, structures and an algorithm to generate a behavioral description of the connector.

(5) Refinement of canonic components into particular technology representations.

Connectors (as well as components) on the platform independent level can be refined for specific technologies. Thereby, the transformation adds technology specific structures, coordination and conversion logic to a connector. Our framework uses a rule-based approach for model transformations. Rules can be parameterized to meet certain requirements of users.

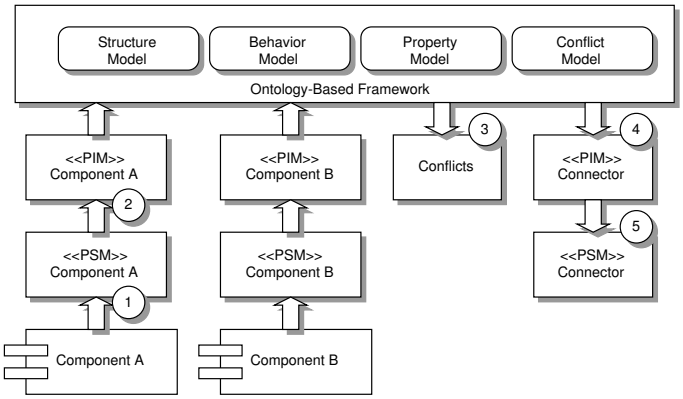

Figure 1.5: General Steps of Component Integration

1.5 Contributions

The contributions of this work arise from the simultaneous treatment of the problems stated above.

(1) Conflict analysis is performed for type, protocol and communication specifications. Consequently, a far better overview concerning the interoperability of components can be achieved than by using analysis tools that deal with only one or two issues. As a result, an estimation of integration costs becomes more accurate and simpler to calculate.

(2) In particular, communication properties are used for deciding compatibility and substitutability. Communication properties are organized in taxonomies for each technology of interest. A conflict analysis based on communication properties was presented in [79].

(3) The framework is extensible for additional analysis methods such as ontology-based checks and pre- and post-conditions. Moreover the language for protocol specifications is exchangeable and the type system customizable. The framework is described in [77], whereas conflict analysis was presented in [78].

(4) The framework can be seen as a 'minimal' model-driven development system, as it supports models - especially component models - on different abstraction levels and further provides transformation functionality between the models. It further provides functionality to store model descriptions and transformation in a component repository.

(5) The framework can be used in a UML-based development process, as UML models can be transformed into the framework's internal representation (RDF) and vice versa. The framework's models (platform independent and specific component models) are expressed by UML profiles in order to describe the necessary transformations. Thus, each UML tool can be used as a front-end if it supports stereotypes and tagged values. This is achieved on the basis of the Evolution and Validation Framework [131], which we have extended by a transformation service. Consequently, we are able to attach arbitrary background information on UML elements and to interpret the background information as RDF statements. This has been demonstrated for communication properties in [79].

We also support a tight integration of the framework and UML tools, as commands can be encoded into UML diagrams. The framework's services analyse the models and execute the appropriate services.

(6) UML does not directly support reasoning. The framework however uses TRIPLE, which is a deductive language based on RDF. As we are able to transform between UML and RDF, we practically realize reasoning on UML.

(7) The framework further introduces flexible refinement operations on components. Therefore, a transformation is adaptable to the requirements of the developer. This is contrary to one-to-one transformations as defined in many software tools. In [25], we demonstrated parameterized refinement transformations for Enterprise JavaBeans. Thereby, the transformation can be optimized for certain situations based on existing J2EE patterns.

1.6 Outline

This work is divided into four parts. The first part describes communication and integration in component-based systems. Chapter 2 describes communication from two perspectives: It first discusses the technical aspects of communication followed by a survey of communication from an architectural viewpoint. The first part also introduces the two operations of interest in this work: checking compatibility and substitutability. Compatibility validates if two components are interoperable, whereas substitutability checks if components are exchangeable. Finally, methods for component integration are introduced. Component integration is realized by connectors that solve the conflicts between otherwise incompatible components.

The second part describes the framework for conflict analysis. Chapter 3 introduces the two processes of interest: component analysis and transformation. Chapter 4 further introduces a communication taxonomy that augments type and behavior checks. Chapter 5 describes the framework. It includes an architectural overview as well as a description of the core models of the framework. Chapter 5 further describes the interaction with UML, which is achieved via profiles, and informally describes a morphism between UML and RDF. Chapter 6 introduces conflict analysis as performed by the framework. This part includes segments of previously published papers: [77–79; 131].

The third part exemplifies the usage scenarios of the framework. Therefore, Chapter 7 describes examples for conflict analysis and Chapter 8 describes parameterized model transformation as presented in [25].

The last part concludes the work and provides visions for future work as well as cooperations.

The appendices mainly include a UML Profile for the core models of the Ontology-Based Framework (OBF) as well as taxonomies describing relevant communication properties of J2EE and Microsoft .Net. A list of previously published papers that were integrated in this work is also included as an appendix.

Chapter 2

Communication and Integration

This chapter first describes the technical foundations of communication and integration and second introduces communication and interoperation from an architectural viewpoint. Communication or more precisely interprocess communication enables the composition of components to complex systems, which in turn results in advantages such as higher reliability, concurrent and distributed computations, reuse and integration of preexisting components, or higher throughput of client requests. Sophisticated architectures of distributed systems such as the client-server architecture and its variations provide the basis for these advantages.

Section 2.1 consequently starts with an overview of the foundations of communication. This includes a basic model of communication, communication protocols, basic interprocess communication (Section 2.1.1) as well as middleware communication mechanisms and services (Section 2.1.2).

Distributed applications are generally composed of several components. If some of the components are pre-fabricated, reused or incompatible because of mismatched specifications, the issue of component integration arises. We base the decision if two components are compatible on formally defined relationships that identify mismatches in different areas of component specifications: subtype relationships, simulation of behavioral specification and relationships between communication requirements. Section 2.2 describes component integration and interoperation. This includes an overview of the important relationships deciding component compatibility and substitutability. The section also includes some hints regarding the question of how to integrate mismatched components.

Section 2.3 describes integration scenarios used throughout this work. This includes a short description of legacy components (Section 2.3.1) and COTS (Section 2.3.2) as well as some notes regarding program understanding. These aspects form potential application scenarios of the framework developed in this work. As we assume that the components to be analysed for compatibility and substitutability are specified in UML, we include a short description of program understanding (Section 2.3.3). Program understanding is required in order to extract the necessary information for a UML specification from legacy components and COTS.

Section 2.4 provides an overview of software architecture, type systems and behavioral specification languages. These are the foundations specifying communication and interoperation at the architectural level and are critical in deciding compatibility and substitutability of components.

2.1 Communication

According to the Reference Model of Open Distributed Processing (RM-ODP), we define communication as

Definition 1 (Communication) *"The conveyance of information between two or more objects as a result of one or more interactions, possibly involving some intermediate objects.*

[Thereby] an interaction takes place with the participation of the environment of the object" [67, pp. 4-5].

Considering the definition of communication proposed by the RM-ODP, we can identify three elementary parts out of which a communication is composed:

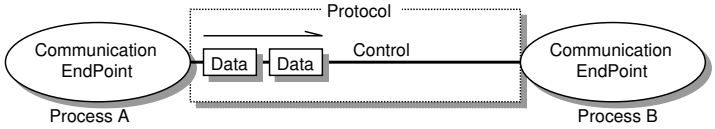

Figure 2.1: Communication Model

Communication Endpoints. Communication requires at least two participants. Both participants must agree upon the protocols used and the data to be submitted in order to successfully initiate a communication.

Data. Communication involves the conveyance of information between participants. This includes the exchange of data in a well-defined format which is understood by both participants. The format describes the types of data that can be exchanged as well as the encoding of the data. For example, object-oriented systems such as CORBA use protocols that describe data exchange encoded as binary data. On the contrary, Web Services describe data exchange on the basis of XML (SOAP), which is based on text encoded messages. This introduces overhead, as data encodings require more space and processing time, but also allows for a simpler integration of different systems and middleware technologies.

Control. Communication can be described by a protocol that the participants must respect in order to interact with each other. In this respect control describes the kinds of messages that can be exchanged and prescribes the order of the messages to be exchanged.

Figure 2.1 shows a respective model that exemplifies communication in terms of the described elements.

Our view of a communication stem from software architecture, which distinguishes explicitly between data and control issues. Shaw and Garlan [124] use, for example, a categorization of architectural styles, which mainly consists of control and data-related criteria.

Focusing on distributed systems, both issues are mingled together in the definition of protocols. According to Coulouris et. al. [40] a protocol is defined as follows:

Definition 2 (Protocol) *"The term protocol is used to refer to a well-known set of rules and formats to be used for communication between processes in order to perform a given task" [40, p.76].*

The definition mainly covers two aspects of communication - the exchange of messages between communication endpoints that are described (ordered) by certain rules and the specification of formats of exchanged data. Both issues correspond exactly to control and data as described above.

Establishing communication between distributed processes is a difficult concern. It involves for example the physical conveyance of information, marshaling of information to the formats supported by different processes in different environments, verification of correctness of submitted data, ordering of transmitted data packets, resubmission of lost data and messages, etc.

As a response to these complex and difficult to handle problems, protocols are normally divided into several layers from which each handles a particular problem. The OSI reference model [45] defines seven layers that describe the most fundamental problems of communication. Each layer uses the functionality provided by the next lower level to realize its functionality. The seven layers of the OSI reference model can be described as follows:

Physical Layer. The physical layer describes data exchange as electrical signals between physical communication endpoints. ISDN is an example of a protocol that is located on this layer.

Data Link Layer. The data link layer mainly provides functionality to check the correctness of the exchanged data between directly connected communication nodes (endpoints). An example protocol for this layer is PPP.

Network Layer. The network layer specifies the routing of information packages in a network. The de facto protocol on this layer is the IP protocol.

Transport Layer. The transport layer describes certain constraints between two or more communication partners. It describes, for example, if a communication is reliable or not. A reliable communication prescribes that communication packages are received in the correct order and that no package can be lost. The de facto protocols on this layer are TCP for a reliable communication and UDP for an unreliable communication. This layer also distinguishes between connection-oriented (TCP) and connection-less (UDP) forms of communication.

Session Layer. The session layer extends the transport layer as it supports process synchronization in communication sessions.

Presentation Layer. The presentation layer defines platform-independent data formats that can be used to exchange information between communication endpoints in different environments such as Windows and UNIX. This level can also be used to introduce encryption in a communication. An example protocol on this level is SSL.

Application Layer. Protocols on this layer resolve the needs of particular applications. This includes protocols such as FTP and SMTP.

Existing protocols do not exactly follow this reference model as they were often developed before the model was released (e.g. TCP/IP) or they structure the levels slightly different according to the requirements of a particular situation. Consequently, it is difficult to identify protocols designed for the session layer or the representation layer. Therefore, we introduce a five layer model as defined by Tannenbaum [135]. His model introduces a middleware layer instead of the representation and the session layers. The middleware layer provides the communication services used by application components to communicate with each other.

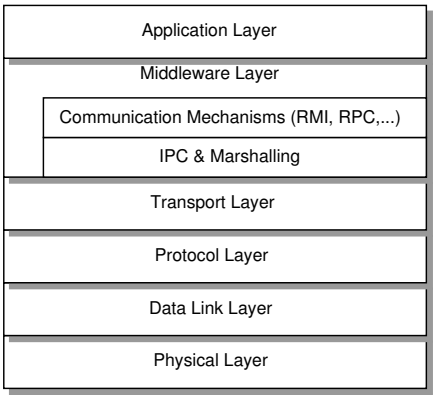

Figure 2.2: Adapted Protocol Stack

Figure 2.2 shows the resulting model. This view also corresponds to the view of Coulouris et. al. They also recognize the gap in the upper layers of the reference model. According to Coulouris, *" the application, presentation and session layers are not clearly distinguished in the Internet protocol stack"* [40, p.78]. Consequently, they amalgamate the application and

the presentation layers as a middleware layer[1] and further the session and the transportation layers, as existing protocols support this partitioning.

The middleware layer is further divided into two subsequent layers: Communication mechanisms are defined in the upper middleware layer. These mechanisms are directly used by application components to communicate with each other. The mechanisms are built upon an IPC protocol layer that provides messages to exchange data and that also introduce services to transform data according to the needs of each communication endpoint.

The positioning of middleware within the Open Systems Interconnection Reference Model reflect the higher level of abstraction that is offered. Middleware hides the primitive IPC mechanisms as provided by operating systems by supporting more sophisticated communication mechanisms that can be used by application components. These mechanisms alleviate the development of distributed applications by introducing transparencies and services such as transactions or security.

In this respect, Coulouris et.al. [40] define Middleware as follows:

> "The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages" [40, p.16].

This definition does not clearly characterize middleware, as it does not describe the way in which middleware overcomes heterogeneity and the different kinds of middleware available.

In this respect, Bernstein [23] proposes the following characteristics to describe middleware more precisely:

- Middleware must support a wide range of applications. Middleware is a standardized and general approach. An approach that aims at a specific scenario should not be considered middleware.
- Middleware should be available on a wide range of platforms and further allow for communication between components defined on these platforms.
- Middleware should support distributed communication.
- Middleware should support standard protocols, allowing the exchange of a middleware product.
- Middleware services should perform its services transparently and via standard APIs to allow for simplified development.

In summary, the middleware layer consists of two subsequent layers: A layer that provides basic services for interprocess communication as well as the necessary functionality to transform data for the communication endpoints and a layer that introduces the more sophisticated communication mechanisms of the middleware including services. The following two sections describe both layers, as they will provide the focus for conflict identification.

2.1.1 Inter-Process Communication

Inter-Process communication can be categorized by the number of hosts involved: communication is either local to a single host or distributed between different hosts. Classic interprocess mechanisms are shared memory, pipes, messages, and RPC. Thereby, shared memory forms an exception as it is based on data exchange on the heap. Therefore, it is a very efficient form of interprocess communication but is only applicable for communication on a single host. The other forms of communication can either be used for local or distributed communication.

[1] Precisely, according to Coulouris, they can also be implemented separately for each application. However, we do not consider this case in the following.

IPC mechanisms define protocols to exchange messages between communication endpoints. A respective message passing mechanism can be based on two operations: send and receive. Two or more processes initiate communication with only these two operations by adhering to certain restrictions and properties on these operations that are obliged by a particular IPC protocol. In general, protocols rely either on Request or Request/Reply semantics[2]. A request corresponds to a single message, whereas a request/reply protocol consists of two messages: A request is sent from process A and received by process B. B computes the request and sends a reply back to process A. This semantics corresponds in general to a remote method/function call.

Communication protocols as well as messages on which they are defined are normally provided by operating systems with network capabilities. Thereby, the exact primitives, the supported IPC mechanisms, and the available protocol stacks they provide are variable. Operating systems are free to choose the granularity and the form of these factors. This can lead to optimizations for particular communication mechanisms, as is often the case in experimental research systems. However, most common operating systems such as Windows NT or UNIX derivates provide mechanisms that use 'Sockets' as abstractions for communication endpoints and define IPC mechanisms between them. Sockets refer to an IP number and a respective port.

A remote communication differs from a local communication in several aspects. In particular, call semantics must be considered carefully, as in contrast with a local call several additional sources of errors are introduced. For example, an operation can fail because the remote node is offline or the remote node takes too much time to compute a request. Therefore, different warranties regarding distributed calls are introduced, which yield the following semantics:

Maybe semantics initiates an operation exactly once but do not provide any reliability warranties. It is therefore not guaranteed that the operation is successfully executed on the server or even reaches the server.

At-least-once semantics repeats an operation if no reply is received in a given time. This can be the case if the request is lost due to a network failure or that the server has not completed the computations of the request in the given time. Consequently, this semantics should only be used with idempotent operations.

At-most-once semantics repeats an operation until it is executed once. This semantics requires a complex design of the server if operations are not idempotent. A solution concerns the introduction of a history that caches already performed operations. For more details see, for example, Coulouris [40].

Exactly-once semantics executes a remote operation exactly once. This semantic is difficult to achieve and requires a sophisticated design of the involved processes.

Communication mechanisms can further be characterized by properties. The following properties are proposed by Coulouris [40]:

Synchronized Communication. Communication between participating processes can be characterized as synchronous or asynchronous. A synchronous communication requires the blocking of the message initiating process until a response of the addressed processes is received. Thereby, blocking can be classified into several distinct forms. Table 2.1provides an overview of different forms of blocking as defined by Tannenbaum [135].[3]

Message Destinations. Messages are normally targeted at only one receiver. In this case a communication requires an initial step that constitutes a connection between two endpoints. Alternatively, a message can be broadcast to a group of processes. This kind of communication does not require an established connection.

[2] Coulouris also introduces a Request/Reply/Acknowledge protocol that adds certain warranties on the reliability of operations.

[3] Tai and Rouvellou also describe a similar matrix in [134].

	Persistent	Transient
synchronous	(b) Process A sends a message and waits for an acknowledgment. The acknowledgment is given by the middleware. The called process B receives the message the next time it is started.	(d) The calling process A waits for a receipt that the message were received. The receipt is given from B, even if the receiving process B does not process the message immediately. (e) The calling process A waits for the delivery of the message, which means that the receiving process B has received the message and started processing of it. (f) The calling process A awaits the response for a message. The response includes the results of the message's computation.
asynchronous	(a) Process A sends a message and resumes its computations. The message is delivered by a middleware to the called process B, even if the process is not running at the moment.	(c) Process A sends a message to a receiving process B. B needs to be running in order to get the message.

Table 2.1: Six Forms of Message Passing Accoring to Tannenbaum [135]

Reliability and Ordering. Reliability describes whether a communication preserves data integrity and message ordering. A reliable communication provides guarantees about these issues whereas in the case of an unreliable communication it is not guaranteed that all messages are received. A protocol designed for unreliable communication is given with UDP.

It should be noticed that each property can be related to the basic elements of communication: synchronization corresponds to control, message destinations to communication endpoints, data integrity to data, and message ordering to control.

2.1.2 Middleware Communication Mechanisms and Services

The second middleware layer (Figure 2.2) defines sophisticated communication mechanisms for use with application components. The mechanisms are augmented with services such as transaction and security that provide valuable supplements of communication. The most common mechanisms are defined on procedure call semantics and message passing.

Remote Procedure Call

A remote procedure call can be characterized as a synchronous communication mechanism between a client and a server which is based on a Request/Reply protocol. Important properties of a procedure call include parameter passing and call semantics. Parameter passing can be described, similar to a local call, as by-value and by-reference passing. By-value passing describes the transmission of a data value to a callee. The process involves the marshaling and de-marshaling of the value. By-reference calls also concern the transmission of data structures. Contrary to local calls, a reference, which corresponds to a pointer to a local object, cannot be interpreted in a distributed environment. Therefore, the referenced structure must be copied to the remote location.

Remote Method Invocation

The advent of object-oriented programming languages also introduced object-oriented distributed communication. Thereby, communication depends on an object model with all its features from which the most important for distributed communication are object references and interfaces. Object references need to be unique in time and space to identify the correct object. This can be achieved, for example, by constructing an object reference from an Internet address, a timestamp, an object counter and an interface identifier.

Interfaces define the methods that can be called remotely. They provide the usual abstraction from the implementation. There are two approaches to define interfaces for a remote invocation. The first and more general approach uses an Interface Description Language (IDL) to define a remote interface. The advantage of this approach is language independence. CORBA and COM use IDLs to define interoperation between objects written in different languages. The second form is used by Java RMI. The interface is defined like a normal Java interface. This results in a more transparent integration of remote invocations.

In principal, a remote method invocation can be performed almost transparently. From a programmer's perspective it should be indistinguishable from a local call. However, additional failure semantics and the higher latency of remote calls should not be defined transparently. Java RMI resolves these requirements, for example, by the restriction that remote communication must be defined via a special remote interface. Furthermore, remote method declarations must contain remote exceptions that are used to express failure semantics in a distributed system.

Message Passing

Message passing cannot be characterized as easily as the former communication mechanisms because several variations are used in middleware systems. Direct message passing, message queues and publisher/subscriber form the most common mechanisms. Whereas direct message passing is often used in a single application, message passing and publisher/subscriber decouple applications in time and space. Both provide services to store messages in case a recipient is not available. Respective products are, for example, Message Brokers and Event Services.

Important characteristics of message passing mechanisms are the degree of coupling between processes, call semantics as introduced above, and the number of participants of a communication. Message passing mechanisms distinguish a variety of asynchronous and synchronous message passing semantics. Table 2.1 shows a respective overview as defined by Tannenbaum [135]. Contrary to procedure calls, message passing mechanisms support one-to-many communication. For example, in the publisher/subscriber model, several subscribers can register for particular events of a single publisher.

Classification of Middleware Communication Mechanisms

Middleware communication mechanisms can be classified by a number of properties, similar to the classification of communication in general. Thompson [137] provides a classification which originally was taken from a report of the Gartner group. Table 2.2 shows the slightly adapted classification consisting of properties describing the communication mechanism, the protocol, the number of participants, and the synchronicity of the communication. It is obvious that most mechanisms can be divided into two groups: asynchronous one-to-many communication and synchronous one-to-one communication.

2.2 Integration

Component integration itself refers to the process of composing two or more components to a more complex system. According to the IEEE glossary of software engineering terms [62], integration is defined as follows:

Communication Mechanism	Protocol	Participants	Synchronicity
Remote Procedure Calls	Request/Reply	1:1	synchronous
Remote Method Invocation	Request/Reply	1:1	synchronous
Direct Messaging	Request(Message)	1:1	asynchronous
Message Queues	Request	1:n	asynchronous
Publish/Subscribe	Request	1:n	asynchronous
Procedure Call	Request/Reply	1:1	synchronous
Database Gateways (e.g. JDBC)	Request/Reply	1:1	synchronous

Table 2.2: Properties of Middleware Communication Mechanisms (adapted from [137])

Definition 3 (Integration) *"[Integration is] the process of combining software components, hardware components, or both into an overall system" [62, p.41].*

The composed system forms another, more abstract kind of a component. This abstract component can be composed of a number of interoperating distributed (primitive) components that are coordinated in order to solve a complex problem. The abstract component can itself be further integrated resulting in more complex systems. Consequently, components form a hierarchy as proposed by several researchers such as Weber and Ehrig [144], Plasil [18], Kramer [85], etc.

The term interoperation refers to the exchange of data and control between components. A definition of the term interoperation, which emphasizes the exchange of information, is also given in the IEEE standard glossary.

> "The ability of two or more systems or components to exchange information and to use the information that has been exchanged" [62, p.42].

Another definition of interoperation, which is provided by Peter Wegner, also adds conflict resolution to the meaning of the term:

> "Interoperability is the ability of two or more entities to cooperate despite differences in language, interface, and execution platform" [145, p.1].

This definition of interoperability comes close to our understanding of the term. In our view, a proper component integration consists of two steps: in 'the' first step, a conflict analysis process identifies conflicts between two components, whereas in 'the' second step a connector is generated that compensates the conflicts to provide a seamless interoperation between components. Consequently, we define the term interoperability as the amalgamation of both former definitions:

Definition 4 (Interoperability) *Interoperability refers to the ability of two components to exchange information and to cooperate despite differences in language, interface, and execution platform.*[4]

Conflicts are incompatibilities in either the required and provided types, in the behavior of components or in the communication requirements. A conflict is defined as follows:

Definition 5 (Conflict) *A conflict is any mismatch between interfaces or behavior of two components or any mismatch between their domains that impedes a seamless interoperation between them.*

The definition of interoperability as defined by Wegner [145] emphasizes an important point of cooperation: Not only the component specifications need to correspond to each other, but

[4]The definition is based on [145] and [62].

also the technological context must match. The context[5] can be, for example, a middleware technology such as CORBA or an operating system such as Unix. As the entities that represent the context of a component can vary depending on the viewpoint of the observer, we use the more appropriate term of a 'domain'. A domain describes the relationship between components and their context or environment. The term is defined by the RM-ODP as follows:

Definition 6 (Domain) *"A set of objects, each of which is related by a characterizing relationship X to a controlling object" [67, p.9].*

Thereby X can be interpreted as a 'depends' or 'use' relationship between application components and their context, e.g. middleware such as CORBA or operating systems such as UNIX form a technical domain.

Interoperability between mismatched components can be realized by providing a connector that compensates the conflicts between components. A connector is a special kind of component that does not contain application logic but only provides transformation and coordination logic. The connector is often not directly visible to application developers.[6]

The RM-ODP uses the term 'interceptor' to refer to a connector that mediates conflicts between different domains. An interceptor emphasizes the relevance of the context of components - the domains - for composition.[7]

Definition 7 (Interceptor) *"[An Interceptor is] placed at a boundary between ... [two] domains. An ... interceptor*

- *performs checks to enforce or monitor policies on permitted interactions between basic engineering objects in different domains;*
- *performs transformations to mask differences in interpretation of data by basic engineering objects in different domains" [68, p.32].*

2.2.1 Compatibility and Substitutability

Establishing interoperability concerns the process of composing components to realize a seamless communication between them. The major question regarding this process pertains to the decision of whether two components are compatible or not.

We regard compatibility between two components. Both components require and provide services that are defined by interfaces. The interfaces consist of method declarations and behavioral specifications. Further, both components are placed in a technological context that describes their communication abilities.

We decide component compatibility by proving type, behavior, and property relationships. Type and behavior relationships are based on well-known theoretical approaches. Property relationships describe the required and provided abilities of the components regarding communication. They are described in a taxonomy that covers important aspects for communication. Contrary to type and behavior specification, communication properties do not provide a 'complete' description of a component. We propose that if a classification is based on carefully chosen properties, communication mechanisms can be differentiated and the root causes of incompatibility can be determined. This augments type and behavior specifications, especially if they are incompletely specified by providing an approximation of compatibility.

We believe that from a pragmatic perspective, a compatibility check of these relationships is sufficient to guarantee a technical interoperability of components. In this respect, we define compatibility as

[5] In this work, we perceive the context of a component as either the underlying middleware or operating system. We further use the term 'domain' instead of the term 'context'.

[6] The terms component and connector are discussed in Section 2.4.1.

[7] We use the term interceptor to emphasize this circumstance. Otherwise, we use the more general term of a connector.

Definition 8 (Compatibility) *Two components are compatible if they can interact with regard to type, behavior and property relationships.*

For component exchange, similar relationships can be used to verify the equivalence between the legacy and the substituent component. We refer to the term substitutability to check for component exchange:

Definition 9 (Substitutability) *Two components are substitutable if they can be exchanged without interfering with the functionality of particular components in any system configuration. Substitutability can be decided by proving equivalence relationships. These include at least subtyping of component interfaces and bisimulation of components' behavior.*

In the remainder of this section, we introduce a classification of relationships to determine compatibility and substitutability. In general, we distinguish two categories of relationships regarding syntactic and semantic interoperability between two components:

Type Interoperability defines relationships on the exposed and required interface types of components. Two relationships can be distinguished: type equality, which verifies the exchangeability of two components and subtype compatibility, which checks if two components can be composed.

Both relationships are determinable and are 'commonly' used in typed programming languages and in middleware systems.

Semantic Interoperability covers different aspects of component specification. Unfortunately, these aspects are not clearly denoted and consequently different terms are used for the same aspects throughout the community. In the following, we employ the terms and the categorization of Vallecillo et. al. [138]. They use the term 'Semantic Interoperability' as a top level concept to further distinguish into: behavioral semantics and protocol semantics.

Behavioral Semantics refers to the principle of 'design by contract' as introduced by Meyer [97] to check for compatibility. Consequently, the caller needs to establish the preconditions of the callee before a request and presumes valid post-conditions after a response.

Vallecillo further introduces the term behavioral subtyping to prove if components are exchangeable. Several authors propose their own definitions of behavioral subtyping to check for substitution.[8] According to Zaremski and Wing [152, p.14] most of the approaches are similar as they are based on

> "pre-/post-condition specifications (1) to describe the behavior of types and (2) to determine whether one type is a subtype of another."

Consequently, behavioral specifications are mainly based on the analysis of pre- and post conditions. These conditions can be annotated to components in several specification languages such as Z [129], OCL [107], OCLPrime [133], Larch [59].

A disadvantage of behavioral specifications relates to difficulties in proving matching specifications as *"behavior-preserving subtyping is in general undecidable"* [138, p.6].

Protocol semantics prescribes the order of operations within a communication. Formalisms to express protocols often rests on a transition system to prove the validity of a communication. Protocol semantics can also be specified in a number of formal languages. Examples are Process Algebras such as CSP [61], ACP [17], or FSP [86], Petri Nets [116], or Message Sequence Charts [64].

Protocol semantics allows checking several relationships: substitutability and compatibility of components as well as protocol properties. Substitutability can be decided by

[8] For an overview see [138] or [152].

checking for bisimulation between components. In literature, several slightly different definitions of bisimulation are proposed. An overview is given in Section 6.2. Compatibility can be decided by simulation relationships. Most research aims at proving process equivalence. However, simulation can be defined by substracting the relationship that establishes equivalent behavior between the callee to the caller from the respective bisimulation definitions. An overview of simulation is also provided in Section 6.2. Based on transition systems, properties such as deadlock or progress can also be checked.

Process algebra expressions and relationships are often checked by model checkers. For several of the languages above, tools were developed that provide the necessary functionality for proofs and property checks. We use these tools for compatibility and substitutability checks.

We call the former relationships formal, as we can prove their validity based on sophisticated concepts, languages and tools. In practice, they are not sufficient: Firstly, they do not allow proving the 'meaning' of operations when composing components of different sources and secondly, semantical specifications are complicated and therefore often omitted for 'common' application components. For both issues, approaches are proposed that aim to alleviate these problems:

(1) Ontologies can be used to provide meaning to operations of different sources. In a respective scenario, operations and components are augmented with meta information that state their purpose and their intended semantics. Consequently, similar components can be searched for in a repository and matches can be performed based on 'matching'. An ontology specifies a graph of related terms of a particular area. The meta information that is attached to a component needs to originate from the same ontology to perform semantic matches.

At present, research aims to augment component descriptions of web services with respective meta information. Several matching algorithms based on annotated information (see for example [111]) have been proposed to select similar components and to initiate respective compatibility matches. Some early ontologies can be used to specify meta information in certain fields.[9] However, a major problem of this approach concerns creating an ontology so that a 'common' agreement can be found. Consequently, at present useful ontologies are rare.

The framework proposed in this work can be easily extended to augment components with meta information that stems from a particular ontology as it is based on RDF. However, we do not exemplify a match on meta information that stems from a predefined ontology. Instead, we present almost the same concept for communication taxonomies introduced below.

(2) A description of the communication mechanism required by components to communicate can, in principle, replace the information provided by semantic relationships. Some approaches already exist that describe the communication as well as architectural requirements of components [44; 93; 123; 124]. However, these approaches are restricted to the architectural level and do not investigate the properties of existing middleware technologies.

This work expands existing approaches, in particular the approach of Mehta [93] to define communication mechanisms of middleware technologies. The resulting taxonomies can be used to reason the compatibility and substitutability of application components of the investigated middleware technologies. This can be used to first provide a quick estimation of the compatibility of two components, which can be used, for example, to decide between candidates for composition and second to partially substitute for semantic relationships if these are not defined for the components of interest. Although a taxonomy based reasoning cannot 'exactly' ensure the compatibility of components, it can describe the

[9] For a selection of ontologies refer to [9].

requirements for communication which arises from the underlying technologies and can therefore be used to estimate the costs of an integration. This work provides taxonomies that describe communication mechanisms of .Net and J2EE.

The framework principally supports all kinds of compatibility checks between components because of the structure of the chosen framework. However, in this work, we restrict ourselves to a configuration that proves type compatibility (subtype relationships), protocol compatibility (simulation relationships) and communication compatibility.

2.2.2 Characterization of Integration

We distinguish integration based on two criteria: the number of processes in the integrated system and the number of domains from which the involved components stem.

As a first criterion, we distinguish integration by the number of processes:

In-Process Integration describes a form of composition, in which the components are defined in the form of shared libraries. Procedure calls are the usual kind of communication in this scenario. Consequently many problems of a distributed communication such as transactions and security are simplified.

Inter-Process Integration. Components are composed via a connector that realizes a form of inter-process communication (IPC). Thereby, the connector mediates between components that are possibly physically distributed and written for different contexts (e.g. different middleware, programming languages, or operating systems).

Establishing a connection requires finding a communication mechanism that either can directly be handled by both components or that can be mediated via the connector.

We can further classify integration by the number of domains that are involved in the resulting system:

Single Domain Integration. Component composition takes place in a single context. Only a single technology is involved in a composition. Consequently, we can interpret a composition as an integration of application components within a single technology.

Multiple Domain Integration. In this case, an integration combines two or more technologies. Consequently, an integration depends on a communication mechanism that is understood and supported by both technologies.

Table 2.3 provides some examples of integrated components in the two dimensional space that is spanned by both classification dimensions.

2.2.3 Approaches for Connector Generation

This section provides an overview of general techniques for component integration. These techniques can be used to approach connector generation.

Integration approaches can be classified by two dimensions: the level of abstraction from a particular technology and the kinds of interoperability considered by an integration approach. In particular, most integration approaches either target at a specific integration scenario for particular technologies or they propose a general solution for a particular problem.

For the first class of integration approaches, software products are available that target at component integration between particular technologies. For example, the COM/CAS Bridge from Sun[10] provides a COM view of Enterprise JavaBeans running in an Application Server.

[10] The COM/CAS Bridge as well as the ActiveX-Bridge are no longer supported by Sun.

	intra-process	inter-process
single domain	library usage. For example, Java encapsulates libraries as Jar files.	application integration in the context of a single technology. This is the common case of component integration.
multiple domains	The Ontology-Based Domain Repository (ODIS) uses this approach to include the XSB knowledge base via an in-process integration. The ODIS is a Java component, whereas XSB is a C component that can be instantiated from a shared library. The two components can be combined because Java provides JNI that facilitates direct access to C functions.	An integration in the context of two different technologies. For example, integration of a COM component and a EJB component via a connector such as the CAS/Bridge or the ActiveX bridge designed by Sun.

Table 2.3: Categorization of Integration

Therefore, Windows clients can use Enterprise JavaBeans via a COM interface. Examples of other integration products are the Janeva [2], K2 [6], or JIntegra [4].

Abstract approaches cover different aspects of component interoperability: patterns mostly target at structural aspects of interoperability. Inveradi [65] provides an algorithm that generates a protocol for a connector that integrates two otherwise incompatible components. She further proves that the generated protocol is deadlock-free. Model correspondences [33] provide interoperability by defining mappings between database schemata and provide functions to map data types.

In the following, we provide an overview of some useful patterns for component integration and give an overview of the algorithm of Inverardi. Both aspects support a semi-automatic connector generation and can be used to resolve mismatches. We perceive a set of approaches that cover all aspects of interoperability as a starting point for the second step of integration: the generation of connectors that mediate existing conflicts between components.

Patterns

Adapter. An adapter aims at mismatched interfaces between two classes: the client and the adaptee. The adapter compensates the different interfaces of these classes by providing an additional layer of indirection between them. The kinds of conflicts addressed by the adapter pattern are type conflicts.

The pattern was proposed by Gamma et. al. [52] in the context of a single object-oriented technology. Both components are elements of the same technology and are executed in the same process. The pattern, however, can also be adapted to components and middleware. In this case, it addresses only components that are executed in the same middleware instance. It does not address problems that occur in the context of distribution or several middleware technologies.

Proxy. A proxy provides a kind of access control for a resource and handles all communication directed at the resource. The reasons for introducing a proxy as a surrogate are manifold. The reason which is relevant in the context of this work concerns location transparency. The resource can be a distributed object on another machine.

In the context of component integration, a proxy can handle remote services required by a local component. As the pattern does not deal directly with incompatible interfaces, it can be combined with the adapter pattern.

Broker. The broker pattern [32] addresses the problem of managing a remote client-server communication. The complexity of remote communication is almost completely hidden from application objects. It introduces a layer of indirection to organize the necessary managing tasks which handle issues such as remote object identifiers, transactions, security, location transparency, etc. The broker pattern is used in middleware systems such as J2EE, .Net, and CORBA.

The broker introduces a proxy and a skeleton to provide local surrogates for the remote object. Several brokers communicate via bridges that externalize object identifiers based on standard protocols such as IIOP.

The pattern provides a kind of standard architecture of a connector for handling communication between distributed and incompatible components. It can be augmented with several other patterns and idioms to handle certain mismatches between client and server.

Interceptor. An interceptor[11] pattern is a refinement of the proxy and the broker pattern. It mainly aims at transparently introducing additional services in a framework without changing the framework. The pattern is of special interest as it is used in middleware systems to handle client requests for component instances. The patterns introduce interceptors as well as dispatchers. A client can install custom interceptors in a framework instance. The framework calls the interceptors via a dispatcher object whenever it reaches a particular state. The dispatcher calls every installed interceptor object passing a special context object which encapsulates state information of the framework.

The pattern also handles the problem of mismatched services between the client and the server object.

Different integration approaches such as Keshav and Gamble [73] or Plasil and Bures [31] use these and additional patterns for connector specification and generation.

Protocol Integration

The algorithm of Inverardi [65] generates the protocol of a connector that provides behavioral interoperability between otherwise incompatible components. The algorithm has the following properties:

- It it based on an architectural style, called 'Connector Based Architecture', which is similar to the C2 style.
- It distinguishes between input and output actions, which are called requests and notifications.
- It uses communication channels to denote the link between the components and the connector that integrates them.

Regarding the integration of components from different technologies, the algorithm suffers two problems: First, methods are often named differently in components originating from different sources. Consequently, correspondences must be defined between method names. Second, the 'semantics' of methods often do not exactly match. Therefore, a complex matching between the operations is necessary, which is also not covered by the algorithm.

In the following, we give a short example of the algorithm based on the Mortgage Bank example introducted in Section 5.3.2. The example consists of two components, 'BLContractMgmt' and 'BLCalc', which have the following process algebra expressions annotated:

[11] The interceptor pattern was introduced by Douglas et. al. [119]

ContractMgmt:
R = (newContract→R1),
R1= (newContract→R1
|simulate→R1).

BLCalc:
Q = (getNewContract→Q1),
Q1= (getNewContract→Q1
|setRate→Q2),
Q2= (getNewContract→Q2
|setRate→Q2
|simulation→Q2).

As the Inverardi algorithm expects input and output actions, we split the action, which represents synchronous operations into an input and an output action. The input actions are annotated by α whereas the output actions are annotated as $\overline{\alpha}$. Both extended expressions are defined as follows:

ContractMgmt:
R = ($\overline{newContract}$→ newContract → R1),
R1= ($\overline{newContract}$→ newContract → R1
|$\overline{simulate}$→ simulate → R1).

BLCalc:
Q = (getNewContract→ $\overline{getNewContract}$ → Q1),
Q1= (getNewContract→ $\overline{getNewContract}$ → Q1
|setRate→ $\overline{setRate}$ → Q2),
Q2= (getNewContract→ $\overline{getNewContract}$ →Q2
|setRate→ $\overline{setRate}$ → Q2
|simulation→ $\overline{simulation}$ → Q2).

The first step of the algorithm reverses the action labels to represent both expressions from the viewpoint of a connector between both components. Inverardi calls the resulting transition system an *AS* graph.[12] In the second step, each action is exchanged by a sequence of an input action followed by an output action. The exact rule is defined as follows:

- Each input action α is exchanged by an input action α_c followed by an output action $\overline{\alpha_?}$. Thereby 'c' and '?' denote communication channels which are associated with components: 'c' refers to a particular communication channel, whereas '?' refers to any channel which provides action $\overline{\alpha}$. This information is used in the last step of the algorithm, which unifies action pairs of the form $\overline{\alpha_?} = \alpha_c$, where '?' is substituted by 'c'.
- Each output action $\overline{\alpha}$ is exchanged by an input action $\alpha_?$ and an output action $\overline{\alpha_c}$.

The resulting process descriptions are called EX graphs and are defined as follows for the example expressions:

[12] We omitted the presentation of this step for this example.

ContractMgmt:
$EXR = (\overline{newContract_1} \rightarrow \overline{newContract_?} \rightarrow newContract_? \rightarrow$
$\overline{newContract_1} \rightarrow EXR1),$
$EXR1 = (\overline{newContract}\rightarrow\overline{newContract_?} \rightarrow newContract_? \rightarrow$
$\overline{newContract_1} \rightarrow EXR1$
$\mid simulate_1 \rightarrow \overline{simulate_?} \rightarrow simulate_? \rightarrow \overline{simulate_1} \rightarrow EXR1).$

BLCalc:
$EXQ = (getNewContract_? \rightarrow \overline{getNewContract_2} \rightarrow$
$getNewContract_2 \rightarrow \overline{getNewContract_?} \rightarrow EXQ1),$
$EXQ1 = (getNewContract_? \rightarrow \overline{getNewContrac_2} \rightarrow$
$getNewContract_2 \rightarrow getNewContract_? \rightarrow EXQ1$
$\mid setRate_? \rightarrow \overline{setRate_2} \rightarrow$
$setRate_2 \rightarrow \overline{setRate_?} \rightarrow EXQ2),$
$EXQ2 = (getNewContract_? \rightarrow \overline{getNewContract_2} \rightarrow$
$getNewContract_2 \rightarrow getNewContract_? \rightarrow EXQ2$
$\mid setRate_? \rightarrow \overline{setRate_2} \rightarrow$
$setRate_2 \rightarrow \overline{setRate_?} \rightarrow EXQ2$
$\mid simulation_? \rightarrow \overline{simulation_?} \rightarrow$
$simulation_2 \rightarrow \overline{simulation_?} \rightarrow EXQ2).$

The last step of the algorithm generates a connector that mediates between both processes. The step consists of a unification of the same actions from different components. Thereby, in each step the action α_c is matched with $\alpha_?$ to yield a transition in the connector.

As mentioned above, the algorithm assumes the same action labels and does not cope with complex relationship between actions. However, both issues are needed to generate a connector for the example. The correspondences have to be defined by a developer. In the example, the following two correspondences are defined:

(1) $newContract \sim getNewContract$,

(2) $simulate \sim setRate \rightarrow simulation$.

The first correspondence fits easily in the unification algorithm proposed by Inverardi. Unfortunately the second correspondence is not as simple to integrate in the unification process because the algorithm is based on the principle that a connector firstly receives an operation from component A and forwards another operation to component B. Therefore all action are organized as sequences of input action that are followed by output actions and we cannot substitute the correspondence for each simulated action because the connector would not support an incoming simulated action from the 'ContractMgmt' component.

However, we can define another expression which represents the correspondence. This expression could, for example, represent a thread in the connector. For the example, the second correspondence can be defined as:

ContractMgmt:
$C_2 = (\overline{simulate} \rightarrow \overline{setRate} \rightarrow setRate \rightarrow$
$\overline{simulation} \rightarrow simulation \rightarrow \overline{simulate} \rightarrow C_2).$

// the process that represents an EX-Graph
$CREX = (simulate_? \rightarrow \overline{simulate_3} \rightarrow$
$setRate_3 \rightarrow \overline{setRate_?} \rightarrow$
$setRate_? \rightarrow \overline{setRate_3} \rightarrow$
$simulation_3 \rightarrow \overline{simulation_?} \rightarrow$
$simulation_? \rightarrow \overline{simulation_3} \rightarrow$
$simulate_3 \rightarrow \overline{simulate_?} \rightarrow CREX).$

The resulting connector definition can be generated by unification of the respective action pairs as follows:

$$
\begin{aligned}
&\underline{\text{Connector:}}\\
&K = (newContract_1 \rightarrow \overline{getNewContract_2} \rightarrow getNewContract_2 \rightarrow\\
&\qquad \overline{newContract_1} \rightarrow K1),\\
&K1 = (newContract_1 \rightarrow \overline{getNewContract_2} \rightarrow getNewContract_2 \rightarrow\\
&\qquad \overline{newContract_1} \rightarrow K1\\
&\qquad | \; simulate_1 \rightarrow \overline{simulate_3} \rightarrow setRate_3 \rightarrow\\
&\qquad \overline{setRate_2} \rightarrow setRate_2 \rightarrow \overline{setRate_3} \rightarrow\\
&\qquad simulation_3 \rightarrow \overline{simulation_2} \rightarrow simulation_2 \rightarrow\\
&\qquad \overline{simulation_3} \rightarrow simulate_3 \rightarrow \overline{simulate_1} \rightarrow K1).
\end{aligned}
$$

2.3 Scenarios for Integration

As the complexity of the integration scenario directly influences the intricacy of the analysis process, we define the simplest but still useful scenario possible: two components that are arranged in a classical client-server architecture. Figure 2.3 provides the conceptual model of a client-server integration. The model supports all forms of integration mentioned in Section 2.2.2.

Thereby, we do not restrict the forms of communication to a particular mechanism. Both components can rely on any form of communication supported by their context (see Figure 2.3). The context is either given by a middleware or an operating system that hosts the components.

To integrate components, a suitable third domain needs to be found which is compatible with both component domains. A connector mediates communication between the client and the server component on the basis of the third domain. Therefore, the connector must be virtually or physically part of all three domains.

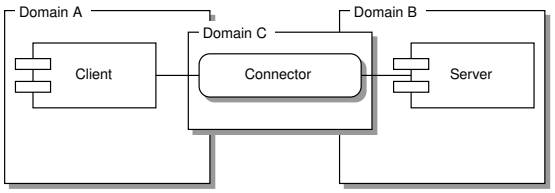

Figure 2.3: Overview on the Integration Scenario

We can further imagine that the server is an existing artifact, which is either a legacy component or a COTS, whereas the client can be either an artifact that needs to be composed with the server or it is the specification of a component defined in a software development process.

The framework requires a UML description of both components for the analysis process. In the following, we give an overview on the properties of both Legacy Systems and Commercials Off-the-Shelf and summarize techniques to create the UML specifications from these artifacts.

2.3.1 Legacy System Migration

There are several definitions of the term 'Legacy System' proposed in literature. For example Brodie&Stonebraker [29, p.3] define a legacy system as follows:

> "A legacy information system is any information system that significantly resists modification and evolution."

Another definition was proposed by Bennett [22, p.1]. He defines a legacy system as

> "large software systems that we don't know how to cope with but that are vital to our organization."

Both definitions do not directly describe which properties render a system as a 'legacy'. They also do not directly define the meaning of 'significantly resists modification' or 'to cope with'. However, summarizing their articles as well as descriptions from [27; 121] we can extract the following characteristics of a legacy system:

- A legacy system is a 'mission critical' system of a company,
- It causes high maintenance costs as it runs on obsolete hardware and is written in an early programming language or has no acceptable internal structure,
- It is difficult to create interfaces to communicate with external systems.

According to continuously changing business needs and the introduction of new technologies, the maintenance costs of each system tend to increase over time. Consequently, legacy systems need to be constantly maintained. In literature, several levels of maintenance are distinguished. According to Seacord [27; 121] maintenance can be classified into three categories: Maintenance, which refers to bug fixes and small functional improvements, modernization, which describes all kinds of system restructuring and component exchange, and system replacement.

Several methods were proposed to modernize (reengineer) a legacy system into a new system. These methods can mainly be classified into 'redevelopment' and incremental approaches: Redevelopment, also known as 'Big Bang' or 'Cold Turkey' approaches create a new system. Thereby, they completely exchange both systems. Redevelopment has a high risk of failure, as the behavior of both the new and old system need to be identical to the context in which they are executed. Incremental approach such as 'Little Chicken' [29], 'Butterfly' [27] or 'Risk-Managed Modernization (Horseshoe Model)' [121] exchange or reengineer systems in small steps, which result in lower risks for the stakeholders.

Modernization can be realized by using a number of techniques such as the ones described by Seacord et. al. [121, pp.10-17]:

Retargeting refers to an exchange of the hardware of a system. The approach aims at cost reduction as old hardware often implies expensive maintenance costs and at the same time limited capabilities in comparison with newer systems. The application is ported unchanged to the new hardware. The advantages depend on the enhanced capabilities of the new hardware.

Revamping exchanges the User Interface (UI) of an application. The advantage of the approach relates to newer front-ends that have enhanced capabilities in comparison with the original UI. Additionally, the modern UIs propose cheaper maintenance costs as they often use common approaches such as web browsers or standard APIs.

Screen scraping is a particular approach of revamping, which wraps the old functionality in modern UIs. Unfortunately, this does not change user guidance.

Integration of Commercials Off-the-Shelf replace system parts. Thereby, COTS are obtained from the market to improve quality and reliability. A limitation of COTS relates to their standardization, which becomes necessary to address a broader market. At it is not always possible to customize COTS for a company's needs, using them often requires partial changes of the architecture or of the existing system environment. Therefore, their usage benefits and their introduced costs need to be exactly calculated.

2.3.2 Commercials Off-The-Shelf

The idea of reuse involves composing components rather than reinventing and reimplementing existing artifacts. In this sense, components are often termed as COTS that can be obtained

from a component market, as was first proposed by McIlroy [91] in 1968. Unfortunately, there is not yet a common understanding of what exactly constitutes COTS. An overview of popular definitions is provided by the Data and Analysis Center for Software, which defines a classification matrix [101] for COTS. According to that classification, we can define a commercial off-the-shelf as

> "a reusable black-box unit with explicit context dependencies that originates from an external source and allows only little or no modification."

Unfortunately, COTS reuse is not as easy to be realized as reuse of ICs. Therefore, a marketplace generally does not exist yet. The Software Engineering Institute (SEI) points out some reasons for its non-existence [35]. It describes a marketplace as

> "characterized by a vast array of products and product claims, extreme quality and capability differences between products, and many product incompatibilities, even when they purport to adhere to the same standards."

Concerning this statement, there are two major problems of COTS integration: The first problem concerns selection and identification of COTS that offer required functionality and quality. The second problem concerns incompatibilities among current technologies that make it difficult to compose COTS.

Component Selection

How do we find a suitable component? This question is a prerequisite for integration. It involves identification of components that are likely to correspond with the stated requirements. Today, this is difficult to achieve because of several problems: First, most existing components are not registered in a repository. Second, there are only a few repositories available. Third, there is no standard format of component description. However, there are several approaches that aim to provide solutions to these problems. A promising approach is the Web Service [36] standard. It provides a format to describe services as well as a standard for repositories.

How do we select the best-suited COTS for integration? Component selection refers to the process of choosing the most adequate COTS for a stakeholder's requirements. These often refer to quality attributes such as performance, maintainability, conformance, etc. It is possible that several COTS are appropriate for system integration, but are differently well suited for different requirements. Today, there is practically no support to select COTS based on quality attributes.[13]

Component Integration

How do we integrate COTS and the software development process? COTS are existing artifacts, whereas software development is concerned with the design of a system. A design often represents a platform independent model of a system that is not bound to a specific technology. Alternatively, it aims for a particular technology, but is placed on an abstraction level that does not cover implementation details. COTS are specified in a technology-specific format. Thus, there is a mismatch in the abstraction levels between COTS and the objects regarded in software development.

How do we compare components? Comparison is an operation that identifies the similarities and differences between components. Comparison between COTS of different technologies proves difficult, as considerable differences exist in their specifications. Another problem concerns the degree of detail of those specifications. Components are rarely fully specified in current technologies. Some facts are stated implicitly or not at all.

[13] Refer, for example, to [142].

For both reasons, integration approaches normally only consider COTS based on the same technology. Integration of components that originate in heterogeneous systems is avoided, as it is a difficult and time-consuming task.

2.3.3 Analyzing Existing Software

Reconstruction of legacy systems is an important concern in system reengineering. It is also important for COTS that are not fully specified. Reconstruction of a system's architecture is a complex task which is outside the scope of this work. However, as it is a precondition for a UML specification of a system, we briefly summarize common analysis methods as defined by Seacord [121, pp.61-64] that aim at source code analysis:

Static Program Analysis includes techniques to generate reports of the structure of elements and the relationships between elements. Reports include for example function call hierarchies, type hierarchies and usage/dependency relationships. The information extracted via static program analysis is valuable to extract architectural descriptions of a system. Unfortunately, static analysis cannot cover all necessary aspects for architectural descriptions, as some issues of a program are only fixed at program execution. This includes late binding, function parameters as well as dynamic configuration of components, which are often used in middleware systems.

Dynamic Analysis generates reports of elements and their relationships, but from a dynamic (runtime) perspective. Therefore, techniques such as profiling are used to recover the necessary information. Profiling analyses the control flow of a running system. It provides information regarding the order of operation calls.

Slicing. Program slicing works on a component's source code. It investigates programs by marking or extracting statements that directly concern the properties of interest.

Redocumentation is a manual approach to code analysis. A developer analyses the functionality of a program and creates appropriate documentation. However, the amount of code that a developer can handle is limited. According to Seacord [121] a developer can handle code up to a size of 50,000 lines.

Refactoring is a popular technique to improve code quality and functionality by methodically modifying the internal structure of code. Fowler [49] provides a catalog of refactoring techniques - similar to the pattern catalog of Gamma et. al. - that describes a number of useful code transformations.

The presented methods aim at an analysis of a system's source code. They provide statistics, reports, or descriptions on the source. An architectural reconstruction, however, requires the creation of higher level abstractions from these basic results. This mainly includes extraction of architectural views, which describes the system from different perspectives such as configuration of the system's components or the used component types and their dependencies. As described in [121] the extraction of architectural views is a highly iterative process that involves interaction of the analysts and the stakeholder.

For a more complete description of system renovation and transformation, Seacord et. al. proposed the Horseshoe model [121]. The model distinguishes three main processes: reconstruction, transformation and refinement. All processes are required for legacy system migration. Reconstruction aims at system analysis and is therefore a prerequisite for component analysis. The process itself is outside the scope of this work. The transformation process covers program and structure transformation and is supported by the framework. The refinement process is partially supported. The framework supports parameterized model transformation,whereas code generation is unsupported.

2.4 Foundations for Component Specification

This section gives an overview of software architecture as well as on type and behavior descriptions. Software architecture provides the basis for a canonical component description, whereas types and behavior provide the basis for the 'traditional' conflict analysis. Types are used in most middleware technologies. In contrast, behavior descriptions are often not supported by these technologies as they are more difficult to handle. However, for both techniques a considerable amount of knowledge exists, so that they provide the basis for deciding structural and semantical interoperability. In the next chapter, we augment both concepts with reasoning on communication properties.

2.4.1 Software Architecture

The term 'Software Architecture' is used in several fields of software engineering with slightly different meanings. This led to numerous definitions of this term (see [3]). In the context of this paper, we define Software Architecture as proposed by Bass et. al.:

Definition 10 (Software Architecture) *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [20, p.21].*

This definition precisely defines the basic elements of software architecture: components, their relationships, systems of these elements as well as properties. On the other hand, it is an abstract definition as it does not exactly state the nature of components and their relationships. Thus, we are free to interpret these terms, which we will do in the following subsections.

Software architecture focuses on a high-level viewpoint of software systems. This captures early design decisions, but leaves implementation details undefined. This, in turn, provides several advantages: a high-level description can be easily understood, it can be used in negotiations with stakeholders, it documents early design decisions, it can be reused like a design pattern, etc.

A general problem of software architecture is linked to the informal description of systems as imprecise box and line diagrams. These frequently used representations often provide no semantics of the meanings of the constructs used. This impedes a system's analysis, rendering the proposed advantages useless. To enable formal analysis, Architectural Description Languages (ADL) as semi formal languages were proposed. Besides ADLs, other efforts for formalization and standardization such as the IEEE 1471-2000 [63] or proposals for documentations such as [38] exist, but are not in the focus of this work.

Software architecture also aims to classify systems based on certain characteristics. This is done by defining a set of (functional) properties such as synchronicity of communication flows, topology of a system, continuity of data flows, binding time of components, delivery policy of messages, modes of data transfer, transactional secured interactions and many more. For a more complete overview refer to [44; 95; 123]. A set of these properties restricted to particular values constitutes an architectural style:

Definition 11 (Architectural Style) *"An architectural style is a description of component types and a pattern of their runtime control and/or data transfer. A style can be thought of as a set of constraints on the component types and their pattern of interaction" [19, p.25].*

In this definition the term 'type' is interpreted as a set of property/value pairs independent from a component's interface. Similar to types, properties constrain architectural elements. Thus, they need to be taken into account during any composition effort. A conflict analysis therefore requires a classification of relevant properties and property values. We describe this issue in Chapter 4.

In the following, we introduce the standard elements that are of interest for the framework. Thereby, we first present commonly used definitions of each term and discuss these in the light of component composition and conflict identification.

Component

Components are generally perceived as elements that perform the actual calculations in a system, whereas connectors specify the communication that takes place between components. A common definition that emphasizes the functional viewpoint of a component was proposed by Shaw [124, p.165]:

> "Component are the loci of computation and state."

Unfortunately, this very abstract definition does not mention any technology dependencies, nor does it clarify the constituting parts of a component. Moreover, in the context of component integration, computation and state as mentioned in this definition are not directly of interest. Both are hidden in the component body (black box view) and are interpreted no further. Solely the interconnection points of a component as well as its associated properties are of interest.

A better suited definition of the term 'component' focuses on the actual configuration of components at runtime. Here, the term 'component' denotes binary artifacts, i.e. executables that are instantiated on a particular node. Thus, each component represents an instance such as a web browser running on a computer. A system configuration then consists of several related and communicating component instances. The OMG proposes this kind of view in UML 1.x. They define

> "A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. [...]
>
> A component may be implemented by one or more artifacts, such as binary, executable, or script files" [108, p.36].

Software architecture also chooses this view if particular properties of runtime systems such as security or performance are of interest. Here, a runtime system describes the layout of participating components, the allowed types of components and their properties of interest. Furthermore, software architecture investigates components in a module view to describe the structure of a component type as well as in an allocation view to describe the physical instantiation of a system [38].

The former definition indicates that the interaction points are determined by interfaces. However, it does not define the exact requirements of these interfaces. A more exact definition, in this respect, was proposed by Szyperski [132, p.34]:

> "A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. "

This definition interprets interconnection points as contracts and 'context dependencies'. Contracts are attached to interface definitions and place additional constraints on the usage of a component. They are stated between components on the same level of abstraction. We interpret 'context dependencies' as implicit relations to components located at another level of abstraction. For example, a context dependency relates application components to components such as containers, which host these components, required databases, or the underlying OS.

A properly defined component composition, however, requires resolving these 'context dependencies'. Every interaction needs to be explicitly expressed via interfaces. A communication between two Enterprise JavaBeans, for example, needs to take into account the underlying

container. Consequently, the container needs to be modeled as a 'canonic' component[14], too. Furthermore, the container cannot be instantiated without an underlying operating system and so on. For a more detailed discussion refer to Section 1.2.

OMG's UML 2.0 proposal more precisely describes 'context dependencies'. It also distinguishes between a component type and a component instance:

> "A component represents a modular part of a system that encapsulates its contents and whose manifestations is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics)" [104, p.110].

This definition is closely related to our definition but does not distinguish different abstraction layers as used for model-driven development. In the light of the above remarks, we define a component as follows:

Definition 12 (Component) *A platform independent (canonic) component (PIC) is an abstract entity of a particular type that consists of a body and 'requires' and 'provides' interfaces, which are defined as contracts and properties associated with these parts. Properties define the context of a component as defined in Definition 6. A component is independent as it has no hidden dependencies besides the ones stated by the interfaces.*

A platform specific component (PSC) has the same structure as a PIC. Different from a PIC, it is specified in terms of its associated platform: A PSC's interfaces are defined in the type system of the platform and its properties - the context of the component - must correspond to the abilities and requirements of its platform (domain).

'Provides' interfaces determine the exposed services of a component. 'Requires' interfaces identify demands of a component that are used in the body to realize a component's services. Thereby, the body represents a black box. Properties describe the context of a component. In this work, a component's context describes the communication mechanism[15] the component depends on. Interactions between components are defined exclusively via connectors, which represent the 'relationships' within a system configuration.

Component types are defined in the metamodel of the Ontology-Based Framework for component analysis and transformation, which is defined in Section 5.1. Component types describes the vocabulary from which system configurations consisting of instances from these types can be built. A system configuration describes the physical layout and the properties of a system.

Connector

The word 'connector' originates in the Latin word 'connectere' [Merriam-Webster], which basically means 'to bind'. Following that definition, the term refers to a relation between components. In software architecture, that relation is represented by a first-class architectural element, the connector. A connector manages the control flow as well as the data flow between components. Thereby, it takes into account all dependencies that are relevant for an interaction. In other words, communication is exclusively conducted via connectors.

A popular definition of the term 'connector' was proposed by Shaw [124, p.165]:

> "Connectors are the loci of relations among components."

This definition expresses the above-mentioned role of a connector. However, it leaves many unanswered questions:

[14] At the same time the container is a connector between these Enterprise JavaBeans.
[15] Communication mechanisms are described by a taxonomy, which is defined in Chapter 4.

- What is the internal structure of a connector, if any?
- Which parts of a connector are mandatory?
- Which properties are relevant to classify communication?
- How are these properties attached to connectors?
- How do these properties characterize an architectural style?
- Which kinds of connectors exist in software systems?

We do not want to discuss all of these questions in this section. However, we want to point out two different viewpoints of how connectors are perceived: from software architecture and from technology specific settings.

Software Architecture View. Software architecture focuses on the properties of connectors, which are used for analysis purposes. Thus, at an architectural level, a connector is treated as a structured element that describes a link between components. This link is associated with particular property values. These determine which kind of communication can take place between components in a system bound by a particular connector. Properties are classified into taxonomies such as the one by Medvidovic and Mehta [93–95], who try to identify existing connector types. Furthermore, connectors need to explicitly provide and require interfaces, in order to associate types and behavior. This information is needed for a formal description of a system. It is, however, of no importance how a connector is realized in a particular technology.

Besides simply providing communication resources, a connector is also able to handle several extra tasks. It can be used to modify the behavior as well as the data structure of passed information. It also allows adding services such as transaction, security, or replication. These tasks can be provided transparently.

Connectors are often used to overcome mismatches between components. In this regard, several patterns were developed in order to describe the role of connectors in a composition on the architectural level [73]. These patterns include proxies, bridges, mediators, links, etc.

Middleware View. Middleware aims for simplified development of large complex systems. Therefore, each middleware technology provides a high-level framework as well as a number of services that can be used to easily access the underlying communication and interaction mechanism. At a minimum, middleware supports a particular application domain and connectors that reflect the needs of that domain. These connectors are well integrated into the technologies and can be handled almost transparently. For example, a complex connector such as a remote procedure call can almost be treated like a simple local call. A developer does not need to write complex code to use remote calls. Thus, connectors are often not explicitly handled in technology dependent component descriptions, but hidden within middleware. In summary, middleware focuses only on a small subset of connectors, which is often not explicitly modeled but can be customized to a certain degree.

Several middleware systems allow customizing the behavior of connections between components. This mainly includes, specifying additional services such as logging and load balancing. Some technologies provide a kind of 'interceptor' to attach arbitrary user defined services. As more concrete examples, the JBoss Server [1] supports a call-back interface to intercept method calls to Enterprise JavaBeans. The BEA WebLogic Server [21] provides a call-back interface in order to specify customized load balancing. Microsoft offers integration techniques, so-called hooks, which easily and transparently allow addition of interception code into existing software [71; 80]. However, customization is restricted to prefabricated connector types.

Definition 13 (Connector) *A connector at a platform-independent level is a first-class entity that links components based on their provided and required interfaces. It consists of interfaces*

that exactly match the component interfaces, a body that establishes the communication, and associated properties that describe the kind of communication that the connector facilitates. A platform specific connector links platform specific components.

Interface

As discussed above, interfaces define the interaction points of a component with its environment. Accordingly, they are often defined as entities

> "... for specifying a service. An interface gives a name to a collection of operations that work together to carry out some logically interesting behavior of a system or a part of a system" [76, p.2].

However, this definition does not mention which functionality an interface exactly has to describe. As mentioned previously, Szyperski exactly defines the role of an interface as a kind of contract between a 'supplier' and a 'consumer'. He defines a contract as

> "... a specification attached to an interface that mutually binds the clients and providers of that interface" [132, p.370].

We take up his position and interpret an interface similar to an 'Abstract Data Type' (ADT)[16]. An ADT consists of a set of operations, invariants and pre- and post-conditions. Operations consist of a number of arguments, a return type and a set of possible exceptions. Arguments consist of a name, a type, a direction, and an optional default value. Constraints define states that have to be valid immediately before or after the end of the execution of an operation. These constraints can be expressed in different languages such as Z [28] or Larch [59].

Almost all current middleware technologies use or support interfaces for component specifications. They either offer an Interface Description Language (IDL) or define interfaces directly in an associated programming language. IDLs are trimmed according to the requirements of each middleware. However, most IDLs offer elements such as operations or attributes that we perceive as 'canonic'elements of every technology.

Unfortunately, middleware technologies often only specify structural aspects - types - of interfaces. They leave out behavior and properties. Thus almost all interface specifications are incomplete as they cannot be used to identify behavioral conflicts or mismatched assumptions on the environment.

Ports and Roles

Regarding system configurations, interfaces are not directly associated with components and connectors. In fact, components are related via ports with interfaces, whereas connectors are related via roles with interfaces. The main reason behind this indirection concerns multiplicities of interfaces. Ports and roles allow defining how many interfaces can be bound by a component. Another reason concerns binding of behavioral specifications. In principle, a component or a connector can simultaneously execute a protocol in each port or role. Additionally, ports can be used to describe communication requirements of connectors and components. In this respect, a connector can be described with different communication requirements on each port. This is the basis to describe connections between different middleware technologies.

In the framework, ports and port types are used to define behavior (protocols) as well as properties (see Sections 5.1.3 and 4). We do not distinguish roles and ports in this work because of the relativity of components and connectors.

[16] Contrary to an ADT, an interface provides no implementation.

System

A system describes a graph of interconnected components and connectors. These elements are bound via ports and roles. A composition requires a match between ports and roles and therefore between the interfaces associated with these elements

2.4.2 A Type System for Components

Type systems are well known in the context of programming languages, where they are used to statically ensure the correctness of statements and whole programs. A typed language provides considerable advantages compared to untyped languages, as it can be automatically checked whether a statement such as *'if x+1 then 'a' else 1'* is correct or not. This is achieved by algorithms, which exploit a predefined typing relation to deduce the type for a statement. In this respect, it is important to notice that types can only be used to prove the absence of certain kinds of 'bad behavior'. They cannot ensure 'correctness'.

In our context, we use types to check if two components can be composed or not. More precisely, we are interested in the question of whether there are type conflicts between components selected for composition. In the following, we will trace back this question to the determination of whether the interfaces of both components are in a subtype relationship. According to our definition of a component, this is sufficient to guarantee structural compliance.

In the remainder of this section, we will briefly define the terms 'type' and 'type system'. Based on these, we will explain the properties of a subtype relationship between types. Furthermore, we will investigate the problem of mapping type systems of several middleware systems into a single system.

Type Systems

In order to clarify the term 'type system', we first precisely define the meaning of a type. In the following, we will perceive a type as a set of values, where each value needs to satisfy particular properties. For example, a 16 bit unsigned integer type is a set of natural numbers that is constrained by a maximal (65535) boundary and a minimal (0) boundary. In the following, we define a type by a quotation from Carelli/Wegner:

Definition 14 (Type) *"There is a universe V of all values, containing simple values like integers, data structures like pairs, records and variants, and functions. ... A type is a set of elements of V" [34, p.14].*

They use this definition to introduce type systems. Please note that this is a simplified view on types. Cardelli/Wegner describe a far more complex theory of types, which is based on 'Ideals', polymorphic types, etc. However, we do not address these advanced properties as they are not required to describe most object-oriented middleware and therefore choose to use the above definition.

Based on this definition, a type system consists of a set of types and a number of rules that express relationships between types. Each type of a type system is either a basic type, which has no internal structure (as far as the type system is concerned) or a composite type, such as a record or a list. A type system assigns a single type to each well-formed term if a derivation tree exists, or fails if there is no deduction for a term. In this respect, we define a type system according to Pierce as follows:

Definition 15 (Type System) *"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute" [112, p.1].*

As an example of a simple type system, we present the three typing rules of the pure simply typed λ-calculus as defined by Pierce [112]:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \tag{2.1}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \tag{2.2}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \tag{2.3}$$

In the simply typed λ-calculus a single function type exists $T ::= T \rightarrow T$. The syntax of the λ-calculus is described by only three kinds of valid terms:

(1) Variables: x,

(2) Abstractions: $\lambda x : T.t$, where $x : T$ represent a typed variable,

(3) Applications: $t\ t$, the application of two terms.

The 'typing context (Γ)' as defined in the typing rules from Pierce, is basically a set of mappings from free variable names to their associated types. Thus, the rule '$\Gamma \vdash t : T$' states that term t has type T, if all free variables in t are substituted by the Types assumed under the actual typing context Γ. This technique of annotations is used in explicitly typed languages.

Subtype Relationship

A type system deduces the type of a term according to a set of given inference rules. However, these rules often are stated too strictly. For example, the following expression is not well typed (without subtyping),

```
void foo() {
 long x;
 int  y = 1;

 x = y;
}
```

because the types 'long' and 'int' are not identical. Nevertheless, it is obvious that it is safe to assign the value of y to x.

In order to construct more flexible type systems, which allow type deductions of 'safe' terms, the subtype relationship was introduced. Informally, a subtype corresponds to a subset relationship. Thus, we can substitute the value of y for x because the set of integers is a subset of the set of long integers.

We define a subtype as follows:

Definition 16 (Subtype) *A subtype is a binary relationship between types, written $S \preceq T$ that indicates that S can be substituted for T. It consists of a set of direct relations between basic types and inference rules for complex types. A subtype relationship is reflexive and transitive.*

According to this definition, separate inference rules need to be specified for each complex type. Additionally, there are two basic rules: a reflexive and a transitive rule. The first rule states that every type is a subtype of its self:

$$T \preceq T$$

The second defines a transitive relationship between types: if S is a subtype of U and U is a subtype of T then S is also a subtype of T.

$$\frac{S \preceq U \qquad U \preceq T}{S \preceq T}$$

Both rules are standard assumptions used for subtyping and were taken from Pierce [112]. Pierce additionally introduces rules for common complex data types such as records, tuples, variants, lists, references, etc. In Section 5.1.2, we use his definitions for functions and records. These rules constitute the foundation for the subtype relationships of components, connectors, and interfaces.

A subtype relationship can be defined with several semantics and can have more sophisticated properties as indicated here. A more complete introduction is given, for example, by [34; 112].

Type Conversion Between Different Type Systems

A major problem for the framework concerns different type systems of middleware technologies. These differ in properties of basic types as well as in supported complex types. For example: a Java 'int' is defined as a 32 bit type whereas a C 'int' can also be defined as a 16 bit type. Furthermore, C supports pointers whereas Java does not, etc.

In general, there are two approaches to map types between different technologies. We can provide one-to-one mapping rules that map between every technology or we can introduce a single type system that participates in each mapping. We refer to the latter approach as a 'star' schema, because a single type system is in the center of each mapping. We use this approach and provide a 'minimal' type system for Ontology-Based Framework for component analysis and transformation (OBF), which we use for conflict analysis and model transformation. It consists of only these types that are needed to decide for conflicts between components. The other alternative would introduce additional costs as there are more connections to maintain.

Furthermore, the proposed approach fits well into a model-driven development that supports abstractions and refinements between PIM and PSM level entities (see Figure 3.1). A PIM is related to several PSMs. Correspondingly, a platform independent component model is related to several platform specific component models.

2.4.3 Formal Languages for Behavior Specification

According to Lamsweerde [141, p.149], a formal specification

> "is the expression in some formal language and at some level of abstraction, of a collection of properties some system should satisfy."

This definition neither prescribes the kind of system that is being specified nor the kinds of properties of interest. This leaves room to interpret the meaning of a system: it can range from a single component to a complex configuration of components within a system and its environment. Properties of interest can be summarized as functional properties and non-functional properties such as security and performance.

According to Wing [148], a formal specification language can be informally defined as a language that

> "provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule, defining which objects satisfy each specification. A specification is [then] a sentence written in terms of the elements of the syntactic domain" [148, p.10].

	Property-based	Model-based
Sequential behavior	Larch, OBJ, Anna	VDM,Z
Concurrent behavior	LTA+	CCS,CSP,FSP, Petri Nets

Table 2.4: Specification Language categorization based on distinctions between property-based and model-based language paradigms and the kinds of system being specified

Specification languages can be characterized by two dimensions: their semantic domain and their specification paradigm. In the context of component-based systems, the semantic domain can be subdivided into sequential behavioral specifications and concurrent behavioral specifications. The former are used to describe the requirements and obligations of component services (their provided functionality), whereas the latter are used to describe the synchronization between components.

Formal languages can further be characterized by their specification paradigm: the way they state system assertions. Specification languages are often distinguished into model-based and property-based languages (see Wing [148]). The former construct a model of the system usually in terms of mathematical structures such as sets and functions. This kind of specification is often based on states. It constrains possible states as well as their transitions. Property-based specification languages define assertions with logical propositions (sets of equations), usually based on an axiom system.

However, other characterizations were proposed claiming to give a more comprehensive classification. For example, Lamsweerde [141] proposes such an alternative classification. Lamsweerde distinguishes specification languages into the following categories:

History-based specifications, which assert the state of a system based on their past, current and future states,

State-based specifications, which constrain behavior with pre- and post-conditions,

Transition-based specifications, which are based on state machines,

Functional specifications, which express system properties mainly with some kind of functions (Lamsweerde distinguishes algebraic and higher-order functions), and

Operational specifications, which express behavioral properties for a number of concurrent processes.

With regard to our objectives of component integration, we summarize popular specification languages according to the characteristics relevant for the framework. Table 2.4 classifies specification languages in two dimensions: sequential/concurrent behavioral specifications and model-based/property-based approaches.

As mentioned before, the framework provides features to support behavioral specifications of both sequential and concurrent behavior allowing conflict analysis. However, there are several problems concerning this objective:

(1) A wide range of specification languages exist that can be used to define the behavior of components. Each of these languages defines its own syntax and reasoning support. How can these languages be supported in the framework?

(2) Another problem concerns conflict identification. How can conflicts be identified if the involved components were specified with different specification languages? In contrast to the approach in the structural model in which we map different type systems into a unique (canonic) type system, it seems to be impossible to provide simple mappings for each of the mentioned specification languages into a canonic representation. This gives rise to the question whether it is impossible to provide an automated checking for behavioral equivalence of components specified in different languages.

In response to the first question, we equipped the framework with elements to handle sequential and concurrent behavioral specifications. As we perceive the framework as a customizable tool, we do not enforce a fixed specification language. It is up to the user to instantiate the framework with appropriate specification languages and to provide mappings between different languages. Thus, it is possible to use any of the languages mentioned above if the constraints can be ascribed to components and connectors using the given framework classes.

Regarding the second question, we do not believe that an ultimate solution exists. However, we designed the framework to at least partially solve this challenge for concurrent specifications. As can be seen from Table 2.4, most popular specification languages for concurrent behavior are model-based approaches. More precisely, they are often based on automata theory. If these automata are interpreted as process graphs, they can be checked for simulation and bisimulation equivalence. These equivalence relations compare the behavior of concurrent executing processes.

Part II

Framework for Component Conflict Analysis

Chapter 3

Approach for Conflict Analysis

This chapter describes our approach to conflict analysis of software components, which always compares two components for compatibility. Section 3.1 starts with a definition of requirements for a conflict integration framework and presents the chosen technologies that satisfy these requirements. Section 3.2 provides an overview of the framework's architecture that fuses on the presented technologies and meets the objectives stated in the introduction. Based on the framework architecture and the technologies presented in Section 3.1, Section 3.3 introduces two processes for component analysis, and transformations are presented. These describe the major activities supported by the framework.

3.1 Technologies for the Conflict Analysis Framework

In order to achieve our objective of component analysis, we impose the following five requirements that must be provided by a conflict analysis framework:

- Definition of a canonical component model that allows describing important aspects of components of different technologies. The model constitutes the basis for conflict analysis.
- Creation of an abstraction function that translates platform specific component descriptions into the canonical component model.
- Proposition of conflict analysis functionality that identifies the most relevant conflicts regarding an integration of two components.
- Creation of refinement functionality to create platform specfic component and connector specifications from an abstract model.
- Integration in a UML-based software development process.

As mentioned before, each requirement alone can be satisfied by using a standard technology or language. For example, UML is the de facto modeling language, whereas ADLs provide canonic representations of components and connectors. However, neither UML nor ADLs directly support reasoning, which instead is provided for by deductive programming languages. Furthermore, ADLs are often defined in proprietary formats and are not supported by a wide range of tools. Model-Driven Development (MDD) supports models on several abstraction layers. It also supports model transformations. However, the MDA, which is the standard MDD architecture, currently lacks a model transformation language. Furthermore, it is not compatible with ADLs. Nevertheless, we believe that combinations of these technologies are able to fulfil all these prerequisites together. We propose the following technology combination:

(1) UML as a front-end modeling language for the end-user.

(2) MDA as an environment that handles models at several abstraction layers and that provides transformations between these layers.

(3) RDF/RDFS to express canonical components and particular technologies.

(4) TRIPLE to provide model transformation based on RDF and reasoning support.

(5) Feature Models to represent communication properties of technolgies and to realized parameterized transformations.

The surprising element is the decision to use RDF as the main component specification language. RDF is a standardized language that can be used in several application areas. It is supported by a growing number of tools. Furthermore, TRIPLE supports reasoning and transformation on RDF.

In the following, we give an overview of the technologies and languages chosen for the conflict analysis framework.

3.1.1 Architecture Description Languages

Software architecture aims at describing the structure and behavior of software systems on an abstract level. Thereby, the details and source code of a system are suppressed and only the 'gross aspects' consisting of components and connectors and their configuration are described.

ADLs formalize architectural descriptions by providing syntax and semantics of the parts and relationships of a system. The descriptions can be used to analyse systems on an architectural level, before an implementation is accomplished. This provides advantages regarding a cost reduction in software development. In particular, an ADL can be used to verify component composition, to provide hierarchical descriptions of a system from different viewpoints, and to analyse non-functional properties and semantics.

According to Medvidovic [92] an ADL should consist of three parts: explicit descriptions of components, connectors, and their configurations in systems. He further defines the aspects that should be described for each part, which contain for components and connectors among other things definitions for syntax, semantics (behavior) and constraints. Medvidovic further notes that an ADL is only useful in conjunction with tools that provide analysis capabilities.

In recent years, several Architecture Description Languages (ADLs) [55; 83–85] were developed to describe the structure and behavior of architectural elements for specialized application domains or for particular analysis requirements. A classification of these languages can be found in [92].

However, there is no standard of architectural description. As a result, a number of different concepts and language constructs were proposed. Unfortunately, most of these ADLs are incompatible. To cover this problem, the research community proposed two ADLs (ACME, xADL) that both provide unified concepts, language elements and frameworks for interchange of ADL descriptions. They divide available elements into two sets: standard elements and 'enhanced' elements. Standard elements are components, connectors, systems, ports (interaction points of a component), roles (interaction points of a connector), hierarchical compositions, and mappings of internal and external interfaces in composite components. Enhanced elements are used to analyse a particular problem and differ from ADL to ADL. Therefore, they are expressed by property sets and can only be interpreted by tools supporting these elements. Both ADLs define the least common denominator of architectural elements, including canonical components. Consequently, this framework is fundamentally based on the descriptions of these two ADLs.

Standard elements represent types of a 'canonic' type system. This enables dependency and usage analysis, analysis of structural compliance of elements to an architectural style and conflict identification of mismatched interfaces. The last point is important for component composition. Components need compatible interfaces to be composed. In particular, a subtype relationship is required between provides and requires interfaces of two components to be composed (see Section 2.4.2).

3.1.2 Model-Driven Development

Component analysis requires a uniform representation of components. Components of different technologies cannot be compared directly, because of inherent differences in their specifications, such as different type systems, etc. Therefore, a concept is needed that describes a principal method to abstract components from platform dependent artifacts into canonical representations.

One existing approach that describes component abstractions is given by Model-Driven Development [96]. Model-Driven Development proposes to model systems at a higher level of abstraction - independent of a particular technology - and to automatically transform a design into code. This kind of development promises faster time-to-marked and cost reduction because specifications on a more abstract level simplify development.

In our case, we are interested in component abstraction. We start with platform specific components and abstract them into a platform independent representation. Thereby, we assume that each component specification not only consists of structural definitions but includes pre- and post conditions, behavior descriptions, and a state model. In the following, we use the terminology from the Model Driven Architecture (MDA) [98], which we perceive as a particular manifestation of Model-Driven Development, to describe component representations on different levels and to describe transformations between these presentations.

The Model Driven Architecture is proposed by the OMG. It targets fully automated component generation and distinguishes two kinds of models: platform independent models (PIM) and platform specific models (PSM). We refer to a PSM if it is based on a particular form of technology such as Enterprise JavaBeans, JavaBeans, Jini, etc. A PSM is normally described in a modeling language such as UML and corresponds to an implementation of the system. For example, the OMG provides several UML profiles (PSM) that describe a platform such as Enterprise JavaBeans or CORBA in UML [109]. These profiles also define mappings in order to automatically generate source code. Contrary to these models, platform independent models (PIM) can be defined without reference to a platform, and therefore without describing a particular form of technology. These kinds of models are usually specified using a modeling language without using platform specific elements, e.g. platform specific types. Figure 3.1 shows the core concept of Model-Driven Development. It distinguishes the different kinds of models as well as model transformations between them. In general, a PIM can result in several PSMs. In particular, Figure 3.1 presents two mappings between a platform independent model and the Enterprise JavaBeans technology, respectively the .Net Platform.

Unfortunately, the proposed advantages of Model-Driven Development cannot yet be realized with MDA because automatic model transformations are required to gain an advantage towards traditional source code development. Currently, the MDA lacks a transformation language to perform the necessary mappings [57]. Therefore, the OMG issued a Request for Proposal for Queries/Views/Transformations (QVT) [106].

Several proposals for model transformation have been recently published in response to that RFP. These proposals can be classified regarding several categories such as how they define transformation rules or rule application strategies.[1]

OBF uses TRIPLE as a transformation language. According to [42], a TRIPLE-based transformation is a declarative relational model-to-model approach.

3.1.3 Feature Models

Feature models play an important role in the area of domain analysis. Introduced in FODA [70] they serve as a description of the features of domain entities using and-or-trees enhanced with some useful elements to express variability.

[1] Czarnecki and Helsen [42] provide a classification of model transformation approaches.

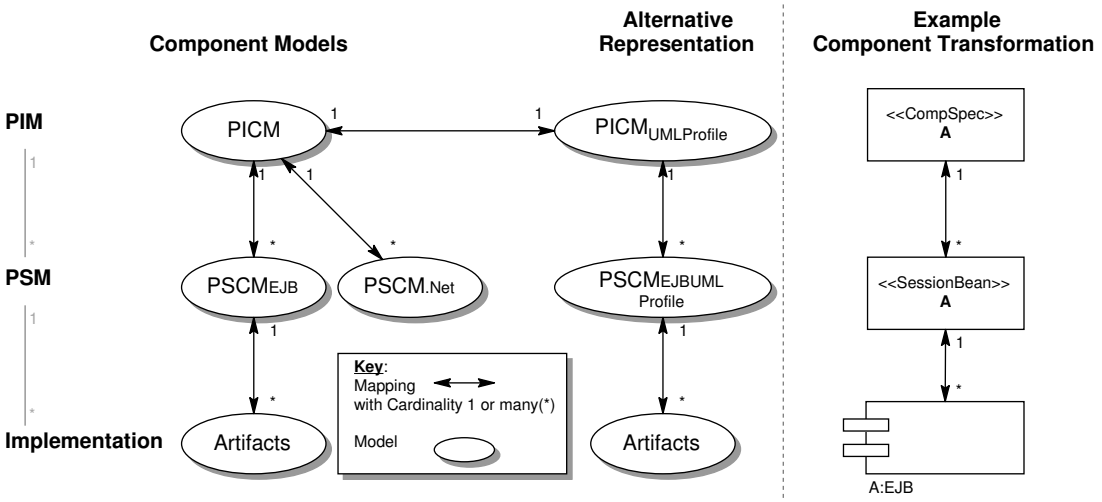

Figure 3.1: Component-Based Models and Their Abstraction Levels

The root of a feature model is called a concept.[2] It describes the 'object' of interest regarding particular stakeholders. Features characterize the relevant properties of the concept:

> "Features allow us to express the commonalities and differences between concept instances. They are fundamental to formulating concise descriptions of concepts with large degrees of variations among their instances" (Czarnecki & Eisenecker [41, p.83]).

Thereby, they span a design space of possible instances. For example, our focus concerns the description of communication - the concept - between several middleware technologies. The subsequent features characterize communication variability by describing possible kinds of communication. A feature instance is a feature model that does not contain any variability elements. For all variability points in the original model, a choice has been made.

In the following, we do not use the original definition of feature models as introduced in FODA, but refer to the representation as proposed by Czarnecki and Eisenecker [70]. Accordingly, a feature model mainly consists of a diagram, and for each included feature of a description, a rationale, an example, a number of stakeholders that the feature is of interest, and a number of constraints.[3]

The diagram consists of several node types that differ in the degree of variability that they allow. Figure 3.2 shows each of the node types, which are described as follows:

Mandatory Features. Mandatory features have to be included in a feature instance iff their parent node is included in the instance. Mandatory features describe the invariant parts of a concept or a parent feature. In a diagram they are represented by a filled circle.

Optional Features. Optional features may be included in a feature instance iff their parent is included in the instance. They are represented by a blank circle.

Alternative Features. An alternative describes a set of features. If the parent feature is included in a feature instance, exactly one feature of the set also needs to be included in the instance. A group of alternative features is represented by a blank arc that spans a set of mandatory features. In the following, we often refer to alternative features as 'xor' features.

[2] We regard concepts and features as synonyms. A concept is the root of a feature diagram. It defines the context of analysis. However, the context is relative. If we aggregate feature models, the former context nodes become normal features in a larger feature model.

[3] It furthermore contains some other aspects that are of no interest in the context of this work. The interested reader may refer to Czarnecki and Eisenecker, [70], Chapter 4.

Or Features. Or features also form a group. If the parent feature is included in a feature instance, one or more of the features in the set also needs to be included in the instance. The group is represented by a filled arc that spans the set of mandatory features.

Optional Alternative Features. This node type is similar to alternative features. In contrast to an alternative feature set, the included features are all defined optional. Consequently, if the parent feature is included in a feature instance, at most one feature can be selected for the instance.

Figure 4.1 shows an example of a feature model, which describes the communication features of a SessionBean.

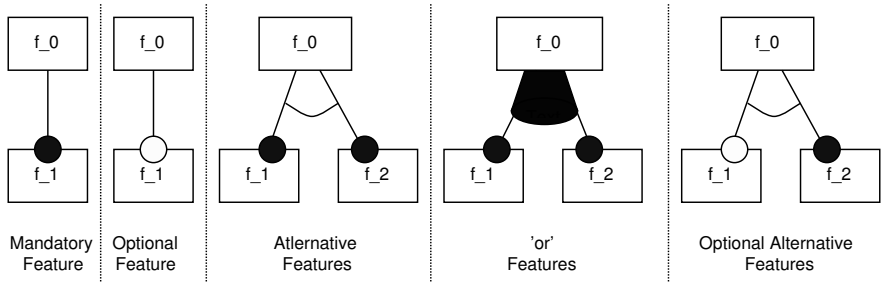

Figure 3.2: Feature Model Node Types

3.1.4 Resource Description Framework

Resource Description Framework (RDF) is a standardized language of the W3C [149]. The goal of RDF is to express information (metadata) about resources on the World Wide Web. Thereby, a resource can refer to individuals, things, properties, or property values. Each resource is identified by a Uniform Resource Identificator (URI). RDF can express properties of resources as well as relations between resources.

Information about resources expressed by statements that consist of a subject, a predicate and an object. The subject is a resource, for which information is attached by predicates. Predicates refer to objects, which are either resources or constant values (literals). A RDF model is consequently constituted by a set of those statements.

In order to exchange information, RDF models should conform to a schema that was defined in RDF Schema (RDFS) [150]. RDFS schema can be compared to a type system in object-oriented programming languages. It defines classes and properties of the classes for a domain of interest. Based on a RDFS instance, applications are able to exchange RDF models. An important difference between a type system and RDFS, however, consists of the treatment of properties. In RDFS, properties are defined independent of a specific class. Another important aspect of RDF concerns data types. RDF and RDFS do not define specific data types. Instead, they define data types as special URIs that refer to a specific type system. As an example, data types are often defined by referring to the types of XML Schema [26].

In this work, we use RDFS to define models for component types, behavior, and properties. The structural model is based on ADL elements, which are important for conflict analysis. Furthermore, several platform specific models that describe components of several middleware technologies are also defined in RDFS. These models are used for model transformations.

3.1.5 TRIPLE

TRIPLE was proposed by Decker et. al. in [126]. TRIPLE is a language designed to provide a reasoning service for the semantic web. TRIPLE facts are very similar to RDF statements.

TRIPLE is based on F-Logic [74], which supports object-oriented features and distinguishes between instance data and schema information (types/classes). TRIPLE states facts as tuples (S,P,O,C): S for subject, the entity to be described; P is a predicate that states the relation of interest; O stands for an Object, which is either a Literal or another tuple; C describes the context within which the tuple is valid. The 'context' is a new construct that allows specifying views of an object in different contexts. This feature is extremely helpful because it divides fact bases into chunks that can be used as separate units.

An RDF statement can be formalized in TRIPLE as

```
subject[predicate->object]@context.
```

Constraints for building such statements are formulated with the special schema-vocabulary RDFS, which essentially enables the definition of binary relation signatures [150]. For example the statements

```
class1 [ rdfs:subClassOf -> class2 ].
prop   [ rdfs:domain -> class2; rdfs:range -> class1 ].
```

express the following constraints: The statement $s[prop \rightarrow o]$ is valid if s is an instance of *class1* or *class2* and o is an instance of *class1*. Therefore

```
x [ rdf:type -> class1; prop -> y [ rdf:type -> class1 ]].
```

is a valid statement with regard to the schema statements above.

For the sake of readability, TRIPLE has been extended syntactically in [24]: Similar to F-Logic the form $p \rightarrow \{A, B\}$ stands for $\{p \rightarrow A;\ p \rightarrow B\}$.

A TRIPLE-mapping is defined through a parameterized context. For example, the clause

```
forall X  @inv(X) {
   forall U,V  V[requiredBy->U] <- U[requires->V]@X.
}
```

simply defines a mapping 'inv' which filters and inverts the association 'requires'. The source model of the mapping is the set of all statements in context X. For each instantiation of the rule body with a statement of the source model a statement according to the rule head will be produced. The target model contains these statements which are in the context $inv(X)$. For example, the query

```
forall s,p,o s[p->o]@inv(ctxExample).
```

transforms the source model with context 'ctxExample' to the target model with context 'inv(ctxExample)'.

3.2 Architecture of the Conflict Analysis Framework

The architecture of the framework for 'Component Conflict Analysis' (Figure 3.3) is divided up into five logical parts: part one consists of a UML modeling tool that is able to import and export XMI.[4] Part two is responsible for transformations between models formalized in XML Metadata Interchange (XMI) and RDF. This transformation is realized as a service of

[4] XML Metadata Interchange (XMI)

the Evolution and Validation Environment (EVE)[5] [131]. We created the Analytical-Data-on-Artifacts-and-Models repository[6] (part three) to store all kinds of component specifications regardless of whether they are formalized as XMI, RDF, artifacts or whether they represent PIM or PSM specifications [118]. Component analysis and transformations are realized by the Ontology-Based Framework for component analysis and transformation (OBF)[7] (part four) [77]. Ontology-Based Framework for component analysis and transformation (OBF) uses external tools (part five) to check for special consistency relationships such as subtypes, simulation, and bisimulation.

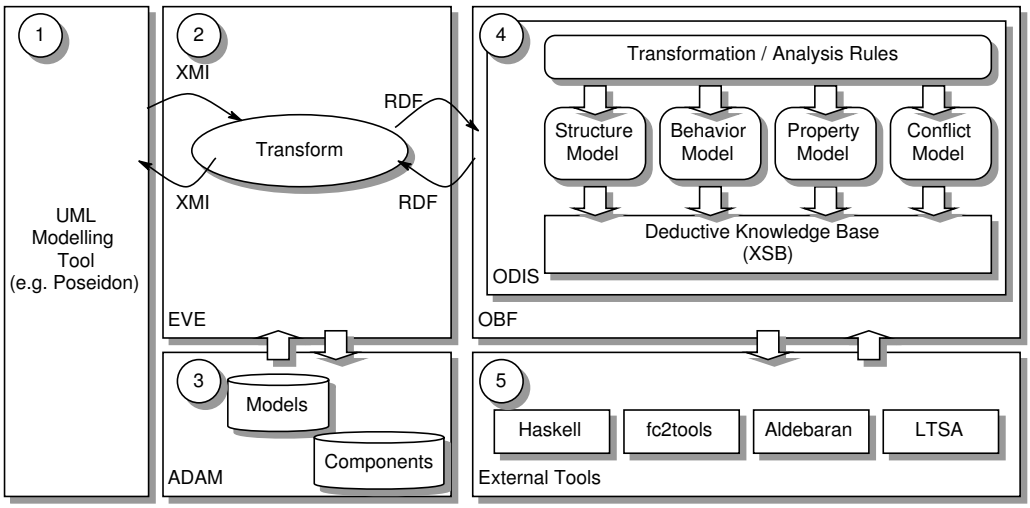

Figure 3.3: Architecture of the Framework for Component Conflict Analysis

3.2.1 UML Modeling Tools

UML modeling tools constitute the front end of the framework. Thereby, developers can use the UML tool of their choice and operate the framework almost transparently. However, as mentioned above a major limitation in the choice of the tool concerns XMI support. UML tools should provide a standardized XMI import and export or appropriate filters need to be specified that adjust the respective XMI streams.[8]

Component analysis requires the specification of components consistent to one of the provided UML profiles of the framework.[9] The profiles define a UML representation of the framework's models. The representation is formalized as a one-to-one mapping between the framework's elements and UML model elements. Most of the framework's models are represented based on UML class diagrams, where the standard elements such as the UML 'class' element are annotated by stereotypes and tagged values. Therefore, most UML tools should be able to handle these specifications.

[5]EVE is based on an idea of my colleage Jörn Guy Süss. Together, we implemented the framework in a student project. On basis of this implementation, I implemented the transformation service, which I needed for the conversation into the OBF framework.

[6]The architecture of ADAM was developed by Felix Schmid as part of his master thesis. ADAM was implemented in a student project.

[7]The OBF framework mainly consists of the core models needed for conflict analysis as well as of rules describing model transformations and conflict analysis. Its execution environment is the ODIS tool, developed by the Fraunhofer ISST.

[8]At present, EVE either works with standardized XMI as exported by Poseidon or supports Rational Rose by providing filters that transform the proprietary format of Rose.

[9]A profile to describe platform independent components can be found in appendix B. A profile for specifying Enterprise JavaBeans can be found in [146].

3.2.2 The Evolution and Validation Environment

The Evolution and Validation Environment (EVE) [131] allows executing arbitrary services on UML models. Thereby, the architecture of EVE is centered around serialized and standardized models, creating a tool independent environment. EVE is based on a Meta-Object Facility (MOF) repository and uses XMI to operate on models. In the context of this thesis, we define a transformation service that converts models between XMI and RDF representations.

Figure 3.4 shows the basic concepts of EVE: Models specified in an arbitrary modeling tool are extracted. Services are applied on these models in a sequential order. The results can either be displayed or handed back to the modeling tool environment. Alternatively, a service can create artifacts such as source code. EVE consists of the following basic components:

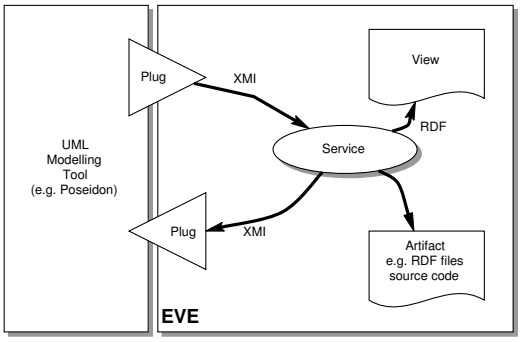

Figure 3.4: Basic Concepts of the Evolution and Validation Environment

Models. The EVE framework is able to handle model instances that are based on a MOF compliant metamodel, as it is based on the Netbeans MDR metadata repository.

Plugs. Modeling tools handle and store model instances in different proprietary formats. In order to use these model instances, they have to be translated into proper XMI.[10] Plugs perform this translation and therefore provide the connection between modeling tools and the framework.

Services. Services define functionality that can be applied to model instances. Typical examples of services are model validation, model evolution and model transformation.

Views. Views are components for immediate visual presentation of models and system or service feedback. A view can be a HTML page, a RDF feedback, or an error report. Views are also used to monitor the status of the EVE platform, as users must be able to determine which services are available.

Artifacts. The term artifact refers to all data that does *not* represent a (UML) model instance. Services may produce or consume arbitrary artifacts such as source code, reports, etc. However, artifacts cannot be passed around in the primary framework. If services intend to cooperate on the content of the artifact, then the artifact must be *modeled*, i.e. expressed as a model instance and embedded into the model that is passed around. Otherwise the services must define a private, proprietary communication channel.

We constructed the EVE framework on the basis of two architectural styles: The 'pipe-and-filter' style [32; 124] was used to provide sequential operations on services. A strict 'layered systems' style was applied to arrange the invocation capabilities of the framework (local access, LAN access, WAN access) in different layers.

[10] Most modeling tools support an XMI dialect that more or less complies with the XMI Specification. Thus, model instances have to be cleansed in order to become usable.

We used the pipe-and-filter style to gain a transactionally secured composition of services. Services represent particular functionality that can be composed in subsequent service calls. Figure 3.5 shows an example instantiation of the EVE framework expressed in a pipe-and-filter style. In the figure, two functionally independent services are composed. The transformation is started by a plug component. A plug is an abstraction for model input or output: E.g. a command line operation or plug-in for a tool such as Rational Rose or ArgoUML. It supplies a model instance to the first or retrieves one from the last service. Each service performs a particular operation on its source model and outputs a modified target model as well as feedback and optional artifacts. Feedback refers to details of the services operation, e.g. error messages or derivation traces. Artifacts are objects like generated code, etc. Each operation of the framework ends either in a viewer or in a plug component.

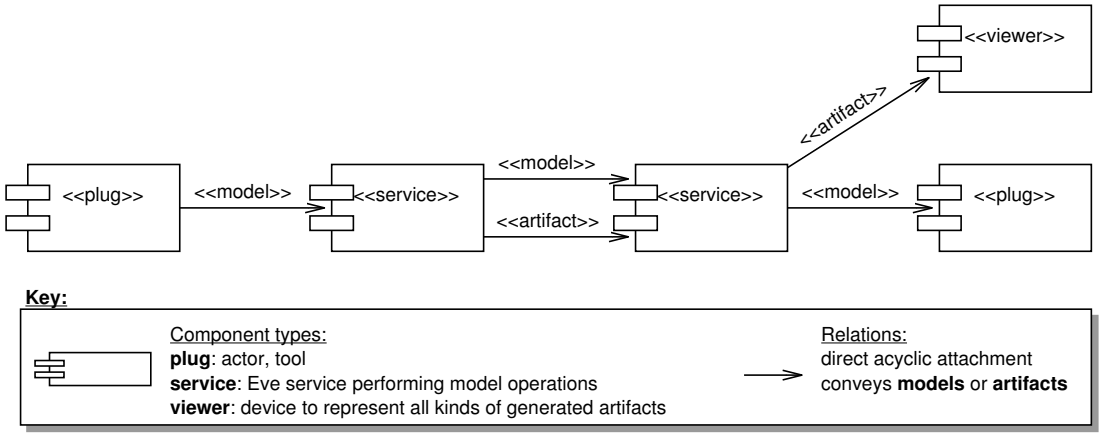

Figure 3.5: Main Architecture of EVE

EVE has been developed over the last two years at the TU Berlin and is mainly part of the dissertation of Jörn Guy Süß.

3.2.3 Analytical-Data-on-Artifacts-and-Models

The Analytical-Data-on-Artifacts-and-Models (ADAM) repository [118] represents a generic facility to store resources of arbitrary format and to handle meta information on these resources. In the context of this work, ADAM is customized to handle component descriptions specified in XMI and RDF, component transformations specified in TRIPLE, communication taxonomies and component binaries. ADAM provides RDF Schema (RDFS) to handle relationships between instances of these resources.

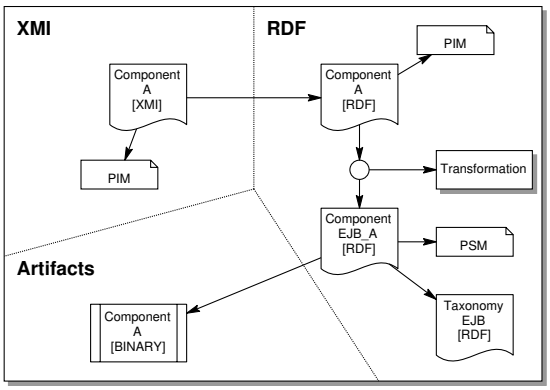

Figure 3.6: The Analytical Data on Artifacts and Models (ADAM) Repository

Figure 3.6 provides an overview of ADAM. The figure shows a single component A that is stored in several representations: It is stored as a PIM both in an RDF and an XMI representation, and also as a specialized EJB PSM component. The relationship between the PIM and the EJB PSM component is described by a transformation. The EJB PSM component is associated with a corresponding communication taxonomy that describes the communication requirements of A. Furthermore, A is stored as a binary artifact in the repository. The relationships between these component elements are managed by ADAM via a customized RDFS, which describes the components on different abstraction layers as well as their relationships. Besides the EJB technology, the ADAM repository can be customized to supports additional component models.

ADAM is implemented as a Java component. It is based on Jena, the Semantic Web Framework for Java [89], and can be customized for the requirements in different application areas. The repository provides an open design which allows saving resources in different kinds of storage systems. At present, it uses the file system and the Tamino XML database to store models and artifacts. The metadata is stored in MySQL.

ADAM was developed in the scope of a student project at the TU Berlin. It was designed within a master thesis [118].

3.2.4 Ontology-Based Framework and External Tools

The Ontology-Based Framework (OBF) for component conflict analysis consists of several RDF schemata describing structure, behavior, communication properties, and conflicts as well as of TRIPLE rules for component transformation and conflict analysis. OBF is implemented on top of the Ontology-Based Domain Repository (ODIS) [24]. ODIS provides an environment for the interpretation of TRIPLE rules and uses XSB, a deductive knowledge basis for rule evaluation. OBF extends ODIS by adding several specialized tools for component conflict analysis.

At present, it uses a prototype implementation, which is written in Haskell to check for type conflicts. Further, it uses the existing model checkers Aldebaran and fc2tools to check for simulation and bisimulation conflicts and LTSA to identify deadlocks and progress violations. OBF models are explained in Section 5.1, whereas conflict analysis is explained in Chapter 6. We decided to use external tools instead of an implementation directly based on TRIPLE, as these tools are already functional or are simpler to implement.

3.3 Processes for Conflict Analysis and Model Transformation

This section gives an overview of the process of conflict analysis as well as of the associated process of component transformation. The second process is a necessary side effect of conflict analysis.

3.3.1 Conflict Analysis Process

The conflict analysis process that is shown in figure 3.7 can be divided up into three parts: The specification part covers the creation of component descriptions, the transformation part transposes these descriptions into a canonical representation and the analysis part identifies mismatches that hinder integration.

Specification

If a developer creates a new component, he is free to describe the structure and behavior as well as to select appropriate communication properties and the technology of this component. However, often binary artifacts are to be integrated. In this case, it can be extremely difficult

to obtain the information necessary to create a UML specification. The first activity of figure 3.7 covers the creation of UML specifications from COTS and legacy systems. This usually requires the application of reengineering techniques such as static or dynamic code analysis to obtain information necessary to create component descriptions. Often tools are used to reengineer artifacts in order to create a documentation and an architectural description. There are a number of techniques, some of which are described in Section 2.3.3, that can be used for reverse engineering.

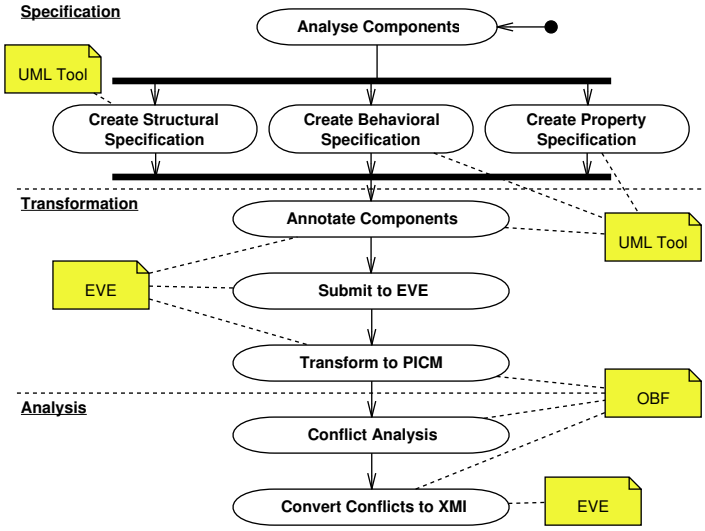

Figure 3.7: Conflict Analysis Process

Most of these analysis techniques are complex and their applicability depends on the particular situation. Therefore, we do not investigate these activities in this work. We assume that reengineering activities have already been applied to extract the information necessary to gain a more or less complete specification of an artifact. We further assume that the structural specification, e.g. the interfaces and types, are completely specified, whereas the behavior and the property descriptions may be incomplete.

Transformation

If all required components are specified in UML, a developer can choose two components for conflict analysis. He can further annotate information that is required for component analysis. Useful annotations span among other directives such as property restrictions, which prescribe communication and the component technology.

Conflict analysis is based on a platform independent canonical component model. This model is represented by an RDFS component model, which we call the Platform Independent Component Model (PICM). Therefore, the transformation activities involve the reinterpretations of components specified in UML in terms of the PICM metamodel.

The transformation involves several steps, depending on the exact format of the UML component descriptions. The steps are described in the transformation process (Section 3.3.2).

Analysis

Conflict analysis evaluates several essential criteria for component integration. Among others, the following criteria are checked: type conformance, behavioral compatibility, existence or absence of certain protocol properties, and compatibility of communication requirements.

The framework checks the actual specifications of two components for conflicts. We do not impose any order or a sequence of the conflict categories in this process. A developer can

analyse the conflict categories he is most interested in. However, we suggest to start either with a structural analysis or with an analysis of the communication properties of components. Checking of communication properties is especially useful, if the amount of integration work between components of different technologies should be estimated, whereas type analysis is most useful, if two components are already selected for integration. Identified mismatches between certain parts and features of these components result in conflicts which are represented as RDF statements. These results are presented to a developer, who then can decide how to proceed.

3.3.2 Model Transformation Process

The analysis process includes an activity for component abstraction. Figure 3.8 shows this activity as part of a general transformation service, which covers not only component abstraction but also parameterized refinements. Thus, component descriptions can either be abstracted or refined to more general or more specific descriptions. Consequently, the framework partly presents a solution to the OMG's Request for Proposal for Query/View/Transformations (QVT).

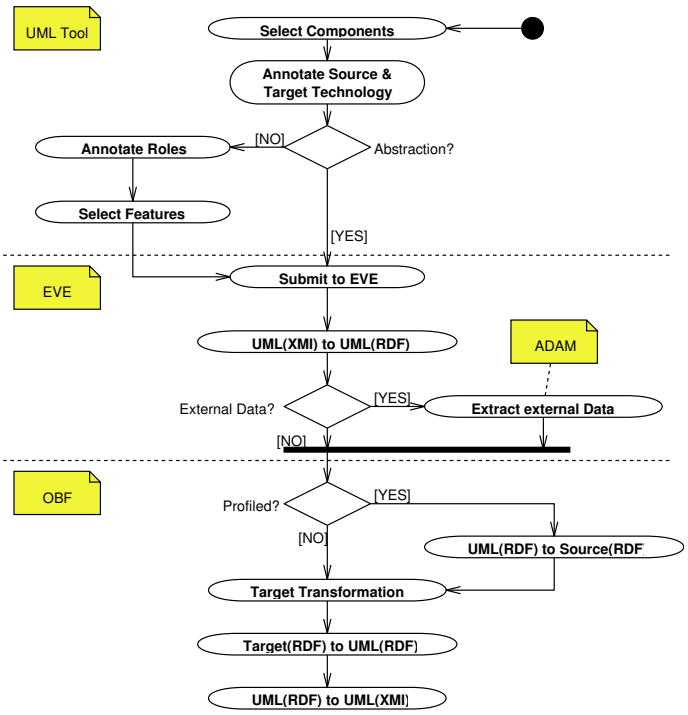

Figure 3.8: Transformation Process

The first activity of transformation involves the selection of UML component descriptions of interest. The descriptions must be annotated with information specifying the intended source and target technologies. This determines the direction of the transformation, which is either an abstraction or a refinement. For a refinement additional information for a transformation can be given: If a transformation is targeted at several components that are composed in a system, the roles of the components in that system can be specified. The roles are used to trigger specific mappings of the components under consideration of user requirements. User requirements are specified by special features that are selected from an overall feature model. This results in a parameterized and therefore flexible mapping of components. We have proposed this kind of mapping in [25].[11]

We propose two approaches to select components for transformation and analysis: First, they can be connected by dependency relationships which are annotated with a note, which gives the

[11] We give an overview of parameterized transformations in Section 8.1.

framework a command to transform or compare these components. This method of encoding commands directly in UML models and is completely tool independent. We proposed this approach in [79]. The approach requires that annotated UML models be exported manually. Alternatively, commands can be specified by issuing commands on a plug which has been integrated into a UML modeling tool. This approach requires a kind of Plug-In support in the modeling tool (see [131]).

The next activity serializes the components as XMI streams and transmits them to our XMI to RDF transformation service. The service has been embedded in the Evolution and Validation Environment. The service extracts the components as well as their attached annotations and generates a RDF representation.[12]

At this point, the components are encoded as RDF statements relative to a UML metamodel. Thereby, the RDF representation is an instance of a RDFS model, which conforms to the UML metamodel that has been used by the UML tool in which the original UML component description has been given. The framework contains several RDFS models that cover the major versions of UML 1.x.

The next step of the transformation identifies external references in the attached information of the components and tries to include external references into the RDF model. At present, this activity concerns communication properties that can be referenced in the UML description. Information concerning communication properties is stored in the ADAM repository. The activity copies the respective definitions into the RDF model.

The UML(RDF) representation is treated differently, regarding the targeted transformation operation. An abstraction transforms platform specific models into a platform independent representation. At present, the only choice of abstraction results in the PICM component representation, which is used for component analysis. A refinement operation transforms the source model under consideration of specified features into the target technology. Both operations depend on the existence of appropriate transformation rules.

If the UML components are specified according to a framework's profile, the former step is preceded by a transformation of the component's RDF(UML) representation to a representation corresponding to the metamodel, which is described by the profile. Thus, the components are reinterpreted (transformed) in terms of their metamodel. However, this operation depends on the conformance of the UML components to a profile of the framework.

After a successful transformation the process is reversed: the components are reinterpreted in terms of UML and are handed back to the UML tool.

3.4 Related Work

In the following, we will describe related work that aims at component integration. We will focus the discussion on approaches which target integration from a software architecture's perspective and from a middleware context. Furthermore, we will look at approaches targeting the model-driven perspective of integration. We will not discuss conflict identification. This issue will be discussed in Section 4.4.

3.4.1 Architectural Frameworks

A yet unresolved problem of Software Architecture concerns component composition. A composition requires checking for compliance of structural and behavioral specifications as well as of architectural properties.

In recent years, several Architecture Description Languages (ADLs) [55; 83–85] were proposed to handle the structure and behavior of architectural elements. They define type systems to

[12] For a detailed description of this process see Section 5.2.2.

handle structural aspects of composition and use formalisms such as Process Algebra [61; 86] to identify and overcome interaction mismatches.

However, despite these formalisms, 'architectural mismatches' [53] still exist. According to Garlan, they are caused by divergent architectural properties. The software architecture community developed several frameworks to cope with component integration. The tools can be distinguished into two categories: Tools that only specify component compositions and component interactions and tools that also enforce the architecture in an implementation.

AcmeStudio is an architectural modeling environment which is based on the Eclipse Platform. AcmeStudio is based on Acme [54; 55]. AcmeStudio supports modeling with architectural styles (Acme Families). Furthermore, it allows the definition of new styles and is able to generate Acme descriptions from graphical representations. Acme is a second generation ADL. It supports seven basic architectural elements: components, connectors, ports, roles, systems, representations and rep-maps. Acme associates property definition with component descriptions to support for a translation of definitions specified in other ADLs.

Archstudio from the University of California, Irvine, is another architectural framework. It is based on the C2 architectural style [136]. C2 uses an asynchronous communication mechanism, which is especially useful for development of GUI software. Archstudio is able to generate Java programs. It thereby provides a link between architectural specifications and a particular implementation technology.

ArchJava [10] also links abstract Architecture Description Languages and frameworks for the design and integration of middleware systems. Contrary to most ADLs it includes new language constructs into the Java Language. Therefore, the architecture and the implementation of a system coincide with each other. ArchJava aims at ensuring the communication integrity of an architecture: Communication between components is only allowed via explicit defined connections. Recently ArchJava was integrated with AcmeStudio.

Similar to ArchJava are approaches such as ComponentJ [122] or ACOEL [130], which also introduce components into the Java language or other mainstream languages. Both approaches introduce type systems and subtyping for components. Further approaches in this area are, for example, Gestalt [120] or Jiazzi [90].

3.4.2 Technology-Related Frameworks for Integration

Approaches such as SOFA/DCUP [113], Fractal [30], or OSGi [5] define component models and standard services on a platform independent level and provide language mappings for several technologies. This concept is similar to the middleware concept. However, the approaches provide advanced concepts in comparison to currently used middleware such as CORBA, COM etc. Components from different technologies are integrated by providing plugs that map these components into the component model.Currently, all approaches provide language bindings for at last the Java programming language. From the three approaches, the OSGi Service Platform is developed by an industrial consortium which includes companies such as Nokia, Motorola or BMW. The highlights of this platform include a standardized specification, flexible application deployment and respective life-cycle management, which also includes remote management, and a number of implemented service components from which can be chosen.

The Vienna Component Framework (VCF) [103] provides a meta-component model to describe components of different middleware technologies. This metamodel supports the addition of concrete middleware technologies via a standardized interface. A technology can be added to the framework by implementing the interface. The framework supports integration of JavaBeans, Enterprise JavaBeans, CORBA and COM components. Contrary to our framework, the VCF does not guide component integration by providing conflict identification. It further does not support a model-centered development process.

UniFrame [115] is similar framework, which targets at composing distributed 'heterogeneous' components. In our terminology, it generates an additional domain which uses adapters to attach components of other technologies. Contrary to ArchJava, UniFrame does not support

component generation. It provides a simple meta-data based model to describe components, their functionality, and their service attributes. Uniframe does also not support conflict identification. Instead, connector generation is based on the Prototype System Description Language, which needs to be created manually. An interesting aspect of UniFrame is related to system composition, which is based on Quality of Service attributes and calculations thereof. Each component is required to provide information about Quality of Service attributes. The framework calculates the optimal system from these attributes. Uniframe also provides a service discovery service [127], which is used to discover optimal components from user queries. This approach is similar to service discovery in the area of web services, but is based on the unified meta-component model from UniFrame.

Other approaches aim at connector generation in the context of different technologies and different communication mechanisms. Bures and Plasil [31] provide an approach for connector generation that takes four communication styles into account and creates a connector by assembling several units, from which each unit provides a distinct functionality such as logging, marshaling, encryption, etc. The approach further takes into account non-functional properties which are considered in connector generation. A prototype is available for the SOFA/DCUP component model [113]. Another approach from Spitznagel and Garlan [128] focuses on connector generation based on protocol transformations. Although, this work is better placed in the area of process algebra research, it aims at protocol generation to connect COTS products. The paper uses FSP to specify the protocols and uses an ADL similar to Wright [11].

As mentioned before industrial solutions do not support a wide range of technologies but focus on the integration of two particular technologies. Examples of products are Janeva [2], K2 [6], or JIntegra [4].

3.4.3 Model-Driven Development

The framework is also related to the principles of Model-Driven Development [96]. It is concerned with the transformation of components between different levels of abstraction. In particular, it can be seen as an application of the Model Driven Architecture [99] as defined by the OMG. Model transformations are defined in TRIPLE. According to the classification of model transformation approach provided by Czarnecki and Helsen [42] TRIPLE is a declarative relational model-to-model approach. The framework supports several platform specific RDF schemata that allow transformation of components specified relative to one of these schemata into the platform independent schemata of OBF.

Chapter 4

Communication Taxonomy

In this section, we propose to augment structural and behavioral component specifications with an additional form of specification that describes the communication requirements of components. This form of specification consists of a property-based description of communication mechanisms of interest. As explained in Section 1.2, application components use one of the provided communication mechanisms of their context for communication. A comparison of the communication mechanisms required by components identifies mismatched properties that hinder interoperation.

Communication mechanisms correspond to architectural connectors as they implement communication between components. In this respect, a property-based description of communication identifies key constraints of connectors. These constraints partly correspond to type and behavior descriptions of components. However, the description is incomplete with regard to a protocol specification and a structural specification as only key properties are included. These properties consist of general restrictions that are implied by the underlying middleware.

Property descriptions of several communication mechanisms are composed into a taxonomy, which describes key constraints of the mechanisms. We call this taxonomy a 'communication taxonomy', because it contains properties that describe the communicationm mechanism used. These properties need to be compatible in order to allow 'communication' between two components.

A communication taxonomy can further be used to support model refinement in MDA. It can be used as a parameter to provide flexible mappings that are adapted regarding user requirements.

Section 4.1 starts with a motivation, which explains why comparison of communication properties is important. Section 4.2 introduces communication properties and the communication taxonomy. We further explain our approach of defining a communication taxonomy based on feature models (Section 4.3) and conclude the chapter with related work.

4.1 Motivation

We propose to augment component specifications with property-based specifications that describe the abilities and requirements of a component regarding communication. Such a specification provides an answer to two common interdependent problems: incomplete component specifications and as a special case implicit assumptions of communication properties:

Incomplete Specifications. Incomplete specifications concern the degree of detail of component specifications. Components are rarely fully specified in current technologies. Some facts are stated implicitly or not at all. For example, a major problem of present middleware technologies is that they ignore the need for a behavioral specification of components. Most popular technologies do not associate this kind of information with their component

definitions. Thus, it is unlikely that there will be a considerable amount of component reuse based on these technologies, at least not without a time-consuming and cumbersome integration process.

Implicit Assumptions. Another problem is implicit assumptions of component developers. Within one technology, aspects regarding communication are often implicitly defined by the structural and behavioral parts of a component specification but not named explicitly. This is sufficient because in the context of one technology communication must be based on predefined mechanisms which the technology transparently takes care of.

However, when dealing with more than one technology, with different communication paradigms, or with adaptable communication mechanisms, it is likely that the requirements of components regarding communication will vary. This is especially a problem if the technology of the component is unknown to the integrator. For example, components of a distributed system often require an initialization of their communication. This often involves contacting a name service. However, if the required initialization procedure is not explicitly specified and the technology is unknown or not standardized, an integrator needs to analyse a component to find out whether it needs initialization and to identify the required order of procedure calls.

Regarding our discussion on the communication model (section 1.2), each component must be seen in a technological context. The context provides abilities to a component but also imposes requirements that must be obeyed. Features describe the abilities and requirements given to a component from its context. Thus, even without regarding the interfaces of a component, certain assumptions on its communication abilities can be deduced solely by knowing its technology context. For example, a Java[1] component (a JavaBean), which uses no additional services such as distributed communication, normally communicates via synchronous procedure calls.[2] It further cannot deal with default parameters nor handle pointers, as these features are not defined in Java.

Inherited features are relevant in conflict analysis. A comparison of a Java SessionBean and a C++ DCOM component will eventually result in mismatches, even if the canonical representation of these components match in their structural and behavioral specifications, as both 'technologies' provide different features. For example, a mismatch between a Java and a C++ component will result because of different mechanisms for name resolution. Java uses, for example, a hierarchical naming scheme whereas DCOM uses an attribute-based naming scheme. A connector must mediate between both concepts.

An important property of a component's feature description relates to the inherent variability of some features. An EJB component, for example, can principally be based on asynchronous communication (MessageBean) or on synchronous communication (SessionBean and EntityBean). Furthermore, transaction handling can be enabled or disabled for an EJB by choosing one of several transaction attributes in the deployment descriptor. Consequently, the knowledge that a component is contained in the EJB technology does not fix all exposed features. Certain features are still variable.

Our goal is to explicitly describe properties required for component composition in order to reason about the compatibility of components. We therefore propose a taxonomy to describe the communication mechanisms of several technologies. We believe that a communication taxonomy allows a quick estimation of the compatibility of two components without relying on a costly analysis from scratch. The taxonomy should also include technology-related metadata. Technology-related information such as the language in which a component is written, platform availability, or resulting costs provides additional information regarding the complexity of a connector.

We argue to express the abilities of a component via features. Features are members of a general communication taxonomy. They provide an overview of the abilities of components in the

[1] We define the abilities of Java as defined by the Java Language Specification [69] including the definitions of the classes packaged in 'java.lang.'.

[2] Asynchronous communication is defined in JMS.

context of certain technologies, which cannot be directly gained from investigating the interfaces of a component. A consistent taxonomy can be defined containing all relevant properties for the technology of interest. From this taxonomy specialized instances can be extracted that describe concrete communication mechanisms of the technologies. The taxonomy can be reused for the analysis of components of the particular technology. Considering the above discussion, we impose two requirements on a communication taxonomy:

(1) A communication taxonomy must include relevant features of components that can cause integration conflicts. The features should describe the abilities and requirements implied by the surrounding technologies from which a component depends.

(2) A taxonomy must be able to express variability, as certain features can be customized.

4.2 Classification of Communication Mechanisms

Each communication mechanism establishes a particular kind of message exchange between components by determining the exact conditions for a valid interaction in terms of message exchange, protocols, data format, services, etc. Thereby, we interpret each mechanism as an instance of the communication model as defined in section 1.2. As our communication model defines three key elements: endpoints, control and data, each mechanism must include exact specifications regarding these elements. In this work, we aim for a high-level classification of communication mechanisms used in .Net and J2EE. We base the classification on a number of key properties that characterize the constraints of each mechanism regarding the key elements of the communication model.

4.2.1 Property-Based Classification

As mentioned above, a property-based classification of communication mechanisms comprises a high-level characterization of certain key constraints of each mechanism. Therefore, the classification is not as accurate as a type specification or a behavioral protocol specification. Contrary to these specifications, it cannot be used in an interpretation that generates a 'complete' representation of a mechanism in terms of a more specific model (e.g. source code). The classification describes only important aspects and can therefore at most be used to generate partial models (e.g. code templates). However, a 'complete' interpretation is not our intended goal. We propose that if a classification is based on carefully chosen properties, mechanisms can be differentiated and the root causes of incompatibility can be determined. This augments type and behavior specifications, especially if they are incompletely specified, by providing an approximation of compatibility.

Regarding these statements, the question arises: Which properties are needed for a useful classification? Unfortunately, this question cannot be exactly answered as we cannot generally state a criterion that determines whether the correct properties are chosen which describe the key constraints of the mechanisms at hand and whether the mechanisms are described in sufficient detail. The level of detail that appropriately describes mechanisms depends on the task at hand. A simple selection of a communication category (e.g. messages or procedure calls) can be based on a small set of properties, whereas a conflict analysis should include a large set of properties. According to these reasons, we can only indicate criteria for significant properties. In general, properties should be related to the key elements of the communication model and further form a single taxonomy which consistently describes the mechanisms of interest. Only a single taxonomy allows a meaningful comparison of explored mechanisms regardless of the level of detail chosen. Properties should be selected if they are relevant for a communication mechanism of one technology of interest. This can result in properties defined on a platform specific level. In the context of this work, we restrict properties by choosing two middleware technologies: J2EE and .Net.

Coordination	Persistency			
	transient		persistent	
	Communication			
	connection-less	connection-oriented	connection-less	connection-oriented
synchronous		RPC, RMI, Java RMI, .Net Remoting, COM		
asynchronous	Messages			Message Queues

Table 4.1: A Classification of Communication Mechanisms

A communication taxonomy should at least contain two independent dimensions that are chosen from the three elements of the communication model. As an example, we define a simple taxonomy consisting of three dimensions, from which each corresponds to one of the three elements of the model and define two values for each property:

Communication can be related to the communication endpoints. A communication can either be *connection-less* or *connection-oriented*.

Persistency describes the storage of data for a deferred communication. Data exchange between two endpoints can either be *transient* or *persistent*.

Coordination describes an aspect of message exchange. Message exchange between the endpoints can either be *synchronous* or *asynchronous*.

Using these dimensions, Table 4.1 shows the classification of common communication mechanisms, including mechanisms used by the middleware technologies of interest. However, the expressiveness of this simple classification is limited as it can only distinguish between major categories.

Consequently, the taxonomy has to be refined to express the differences between mechanisms such as Java RMI and .Net Remoting more clearly. This can be done by adding appropriate properties to the basic classification. Candidate properties for a refinement are for example, parameter passing mode, error handling, cardinality, etc.

Besides properties that directly describe the key elements of the communication model, another category of properties that can be commonly found in middleware needs to be described. Middleware technologies often aggregate services such as transactions, security, monitoring, cryptography with communication mechanisms to further simplify application development.

We can further distinguish properties in two categories: Explicit visible properties and transparent properties. A transparent property is not directly visible in an analysis of a component specified in a technology. At least eight transparencies [66] are defined in order to simplify development and to provide other advantages. In the case that a communication mechanism implements a transparency, some properties become transparent with regard to the specification and the source code of a component. The identification of the exact values of these properties requires the analysis of the specification or the source code of the mechanism.

On the contrary, explicit properties can be identified from the specification and the source code of components. They either relate to the programming language and describe the properties of a language regarding communication or they directly relate to a special communication mechanism. In particular, all customizable features are allocated in this category. Besides compatibility checks, customizable features are also useful for parameterized model transformations, as they can be used to generate special platform specific component descriptions regarding the variants chosen.

4.2.2 Definition of a Communication Taxonomy

The assembly of an adequate communication taxonomy is a cumbersome and time-consuming task. Fortunately, a number of existing approaches [44; 123] can be used as a foundation, from which a taxonomy for the technologies of interest can be developed. In our opinion the most useful and advanced taxonomy was proposed by Mehta and Medvidovic [93; 94]. Their taxonomy defines eight connector types from which the procedure call connector is shown in Figure 4.7. Each connector[3] is characterized by a number of properties and allows for a fine grained differentiation between similar mechanisms.

Unfortunately, Mehta's taxonomy cannot directly be used for comparison of middleware communication mechanisms for two reasons:

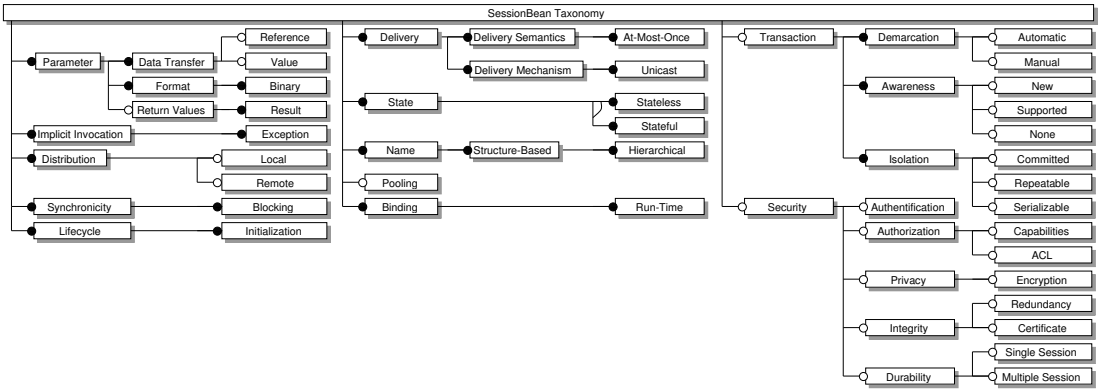

Figure 4.1: Example of a Feature Model (FM) describing the Communication Properties of SessionBeans.

- The taxonomy is defined on a platform independent level. Consequently, it does not include some properties that we need to distinguish connectors of middleware technologies.
- It does not directly handle customization of connectors. Middleware technologies often provide communication mechanisms that can be customized by a developer to a certain degree. For example, Enterprise JavaBeans can be configured to handle transactions. Therefore one of the six attributes defined in the EJB specification has to be selected for each method of an Enterprise JavaBean. Therefore, we need to describe the exact relationships between properties offered by a connector and used by a particular component.
- The taxonomy includes properties that describe the interns of connectors. As we annotate the properties to component's and connector's ports, this information is irrelevant for conflict identification.

Consequently, we adapted Mehta's taxonomy by introducing technology specific properties and by introducing a mechanism to handle variability of properties. Precisely, we modified the original taxonomy in the following way:

- We selected only properties that provide additional information about communication mechanisms. Thereby, the properties must provide information that is not directly visible in the type and behavior descriptions of components. The information[4] must provide knowledge of technology specifics and the underlying architectural styles that are relevant for connector generation.

[3] A connector in the taxonomy corresponds to our notion of a communication mechanism.

[4] The information can be expressed by type and behavior descriptions but the necessary specification is often missing or not necessary in the context of a single technology.

- We omitted properties that are not relevant in the context of our proposed communication model. This particularly includes properties that deal with hardware issues, n-ary relationships, or that describe properties for adapter functionality.
- Platform specific properties that describe communication in Java, J2EE and .Net were analysed [51; 81] and added to the taxonomy. Figure 4.1 shows for example a part of a feature model consisting of communication properties relevant for EJB Session Beans, whereas Figure 4.2 shows a feature model describing Java procedure calls.
- Mandatory and optional properties are distinguished. Mandatory properties refer to structures and behavior that must be present, whereas for optional properties there is a choice for implementation. Optional properties are interpreted differently regarding the type of element they are attached to (see Section 4.3.2).
- The original Connector Taxonomy of Medvidovic/Mehta consists of eight connector types, which as they claim, are sufficient to express most of the connectors which can be found in present systems. These connector types were removed from the taxonomy. Instead, properties are directly annotated to connector types[5] ('ConnectorType') from the defined platform independent ADL (see Section 4.3.1).
- Each property is precisely defined by a description.
- Properties with equal names are renamed to resolve ambiguities in handling component descriptions.

A second taxonomy covers the aforementioned technological properties such as platforms (OSs), programming languages, etc. As we have not found any existing taxonomy that covers these properties, we have defined them from scratch. This taxonomy provides additional information for a quick estimation of compatibility. The taxonomy is described in [102] and used for component search, in the ADAM repository [118]. The approach can be compared to similar approaches in the area of the semantic web for semantic matching[6] such as the one proposed by Paolucci [111]. However, the taxonomy is mostly limited to describing implementation aspects and the purpose of components.

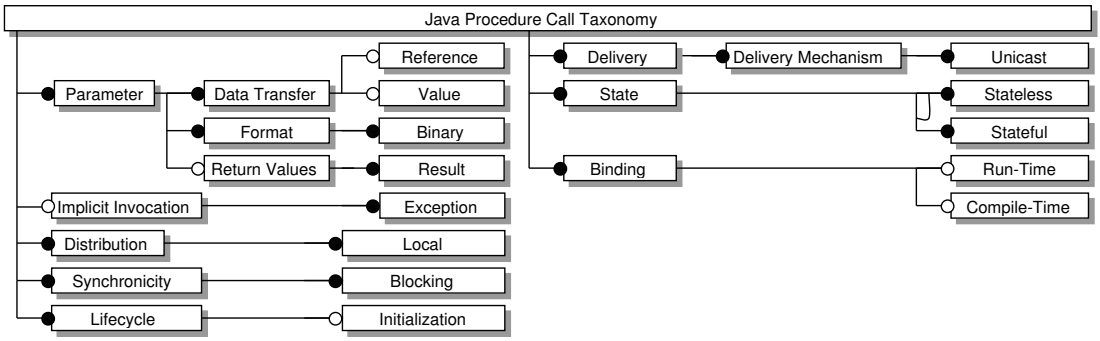

Figure 4.2: Example of a Feature Model (FM) describing a Java Procedure Call: filled circles represent mandatory features, empty circles represent optional features.

The communication taxonomy forms a consistent set of terms that describe the technologies of interest with the same properties and on the same level of abstraction. This results in a meaningful comparison of components written in these technologies. As we are mainly interested in Middleware technologies, we omit several properties and dimensions from the original taxonomy, which are of no interest for the chosen mechanisms.

[5] Properties are more precisely annotated to the ports of connectors and components.
[6] Matching based on ontologies.

Communication Taxonomy

In the following, we describe properties of the overall taxonomy augmented with platform specific properties. Figure 4.3 provides an overview of the feature model of the taxonomy. In the general taxonomy, most properties are optional properties that can be specialized for a particular communication mechanism. The taxonomy is composed based on the works of Liao [81] and Gädicke [51], who researched J2EE, Jini and .Net technologies. We use a simple table consisting of two columns: property name and description. We mark top level properties with bold font and properties that represent values with italic font. We further introduce a special section in the description called 'Interpretation', which explaines the meaning of a property in the taxonomy.

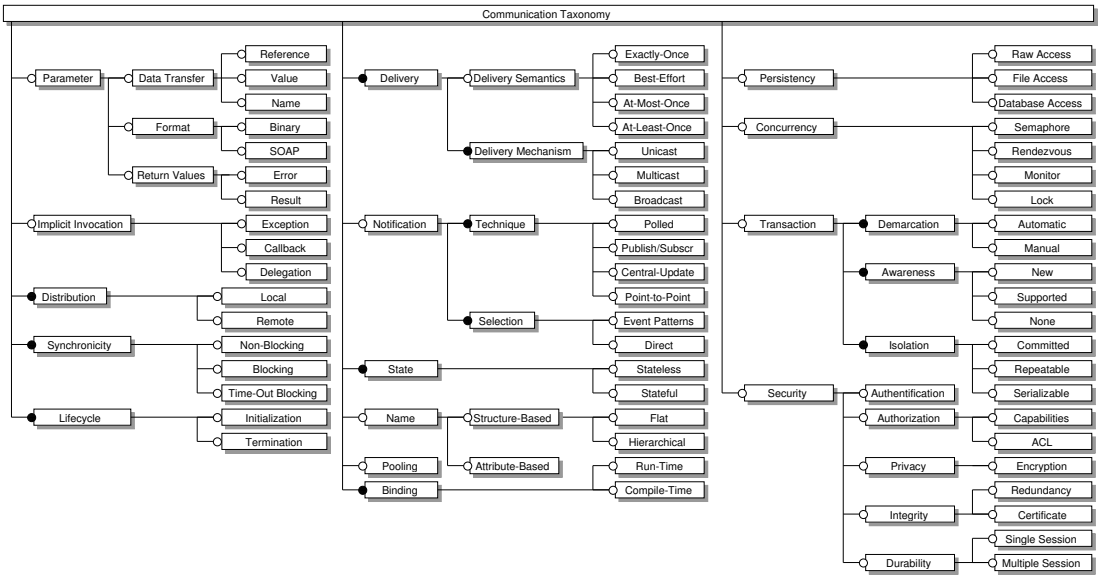

Figure 4.3: The Communication Taxonomy

Property Name	Description
Parameter	Parameters denote the arguments that can be passed along with a call to another routine, process or component. They describe the abilities of a language or technology to handle arguments, data transfer, exceptions, data types, etc. Parameters are visible in the interface declarations of components and connectors.

Data Transfer	Data transfer denotes the way data is passed to the receiving routine. In general, there are two calling conventions: call by value and call by reference. A subroutine call that uses a 'by value' call first evaluates the argument and passes the value to the receiving routine. A call 'by reference' submits the address of the argument (indirect addressing) to the callee. Consequently, the callee can modify the original argument. In addition to these main variants, some derivations are defined. For example, call by reference can be distinguished into two cases: automatic and manual dereferencing of the pointer. For example, C++ provides both modes by specifying a pointer (X *) or a reference (X &). A reference automatically dereferences the argument. Call by name is another calling convention proposed by some programming languages such as ALGOL. It differs from the 'normal' cases as it passes the argument unevaluated. Each time the callee references an argument provided 'by-name' the argument is re-evaluated. The classification of Java regarding data transfer yields an interesting case: Can Java handle by-reference parameter passing? Java distinguishes two kinds of types: primitive types and reference types (see [69] Chapter 4). For each method invocation, a new activation frame is created for the callee and the parameters are copied by value to this frame. The callee works on the copies of the 'original' values. However, if the callee accesses a variable of a reference type, it can 'globaly modify' the referenced object. This effect can be compared with that of reference types in C++. A C++ reference type encapsulates a pointer that is automatically dereferenced, if accessed. In summary, Java provides no concept to handle pointers explicitly. This is a simplification to allow writing programs more easily. Behind the scenes, however, Java handles references. As this can effect globally defined objects, we consider Java to also have 'by-reference' data transfer, even if the official documentation only mentions parameter passing by value. *Interpretation:* The parameter describes the supported modes of data transfer that must be compensated by a connector. Thereby, reference parameters often require more complex type mappings and bookkeeping capabilities included into a connector. Konstantas [75] describes these issues in detail. The property is mandatory if a technology supports method arguments in general. The parameter modes reflect the actual usage of parameters in a component. For the taxonomy of the mechanism, they must be declared optional if they are supported.
Reference	Parameter passing by-reference.
Value	Parameter passing by-value.
Name	Parameter passing by-name.
Format	The data format in which data is exchanged. The property has to be delcared mandatory if parameters are supported by a mechanism. For example, .Net supports binary and SOAP data passing. *Interpretation:* Different data formats must be handled and transformed by a connector.
Binary	Data is communicated in a binary format.
SOAP	Data is communicated in a XML-based data format.

Return Values	The property describes if the result of a procedure call contains error information, normal result values, or does not support return parameters. Some technologies such as COM use this mechanism to signal errors. The mechanism is often used if a technology does not endue an exception mechanism. The encoding of errors can often be identified by special return types (e.g. HRESULT in COM). It is also important as many newer technologies rely on an exception mechanism to signal errors which can lead to conflicts. *Interpretation:* This property generally describes if a mechanism uses Return values to indicate errors or not. It is mandatory if a mechanism supports return parameters.
Error	Procedure call returns by convention an error status.
Result	Return values of a procedure call imply no special meaning.
Implicit Invocation	Implicit invocation refers firstly to an event mechanism for which components register and are called back by an event provider and secondly to an exception mechanism that is implicitly invoked via a method, component, or the runtime system in case of an error. Both aspects involve the transfer of the control flow. The first idea of an event mechanism is described by Shaw and Garlan [124]. Exception mechanisms are often explained in programming handbooks. *Interpretation:* The property describes the possible effects of a call. It should be declared mandatory if one of the following effects can occur. The subproperties are selected if a communication mechanism explicitly supports callbacks, exceptions, or delegation by defining appropriate constructs and the mechanism must be used or is used by a particular component.
Exception	An exception is triggered by an unforseen error. Exceptions can be classified in two categories: exceptions that can be handled by an application and exceptions that result in an immediate termination of program execution. In the taxonomy, a component annotated with this property can make use of exception functionality. For example, Java uses exceptions for remote method invocations, whereas COM does not use exceptions but uses special error conditions (HRESULT).
Callback	A callback mechanism describes the general principle of registering a function to be called by an external resource. However, there are several slightly different application areas in which the term is used, resulting in a wide variety of definitions. Szyperski [132, p.48] defines a callback as follows: *"A callback is a procedure that is passed to a library at one point; the callback is said to be registered with the library. At some later point, the callback is then called by the library."* According to this definition, a callback can be described as an external call of part of the application program. Callbacks are often used in GUI programming. They also often refer to an asynchronous event-mechanism. A callback is important for the description of communication as it reverses the control flow.

Delegation	Delegation is a concept to 'reuse' functionality of other objects. A receiving object does not handle the request completely by itself but 'forwards' the call to an inner object. The difference between delegation and normal message passing is the passing of the receiving object's reference to the delegate. Therefore, the delegate can refer to the receiving object explicitly. In this situation, the receiving object is refered to as the outer object, whereas the delegate is refered to the inner object. Delegation is in this sense similar to inheritance. The outer object corresponds to the subclass, the inner object to the super class. Thus, the call graphs of two objects being in an inheritance relationship are similar to that of two objects - with the same functionality - being in a delegation relationship. However, delegation requires a 'planned' interface, it is not possible if the object to be delegated is not designed for such a relationship. For a more complete discussion refer to Szyperski [132] or to Gamma et. al. [52], who uses delegation in several design patterns. For the taxonomy, delegation is interesting as it can provide a coupling of components similar to an inheritance relationship. However, it cannot be easily recognized by analysis tools.
Distribution	Distribution describes whether a communication mechanism aims at supporting communication within a single process or between several processes. This is important because the meaning of parameter passing modes (value, reference) depends on whether the communication of components is local or distributed. Some communication mechanism such as EJBs support both local and distributed communication. A communication between two or more components can either be local or remote. A local communication happens in the same context, e.g. a process or a VM. Local communication does not use the mechanisms of distributed communication. This results in a more efficient form of communication. *Interpretation:* Parameters are handled differently regarding the distribution of components. If a mechanism supports both local distribution (single process communication) and remote distribution, the parameter passing models have different meanings. The property have to be declared mandatory. The subsequent properties has to be declared either mandatory, if they are supported by a mechanism, or unsupported.
Local Distribution	The communication partner is expected in the same context.
Remote Distribution	The communication is distributed and crosses process boundaries.

Synchronicity	A fundamental characteristic of communication concerns the behavior of a caller between the issue of a request and the reply by the callee. The two major choices for the caller are blocking or non-blocking. If a caller is blocked while waiting for the result, it does no computations until the result arrives. In the other case, the caller resumes computations and either does not expect a result or receives the result by a special operation. Blocking can be further distinguished by the amount of time that a caller awaits results. A time-out blocking returns after a certain amount of time and the call is considered as failed. *Interpretation:* Synchronicity is mandatory for each mechanism. At-least one subproperty must be supported by each communication mechanism.
Non-Blocking	A caller resumes work while its request is pending.
Blocking	A caller is frozen, while its request is pending.
Time-Out Blocking	A caller gets blocked only for a certain amount of time.
Delivery	Delivery directly refers to the call semantics of interprocess communication as introduced in Section 2.1.1. It describes the guarantees about a message delivery. It further describes whether communication is point-to-point or point-to-many. *Interpretation:* Delivery is mandatory for each communication mechanism. Furthermore, Delivery Mechanism is mandatory and at least one delivery mechanism has to be supported.
Delivery Semantics	The call semantics of interprocess communication. *Interpretation:* Delivery Semantics is mandatory for mechanisms that aim at distributed communication.
Best effort	There are many definitions of this term. According to Emmerich [46, p.121], best-effort calls *"do not give any assurance about the execution of the request."* Plasil [114] further characterizes best-effort as a one-way asynchronous call.
Exactly-Once	A remote call is executed exactly once.
At-Most-Once	A remote call is executed at-most-once.
At-Least-Once	A remote call is execute one or more times.
Delivery Mechanism	The property describes the available mechanisms to communicate messages in a network. In general, three mechanisms can be distinguished: Unicast, Multicast and Broadcast.
Unicast	Unicast refers to a point-to-point communication between two network nodes.
Multicast	Multicast refers to an one-to-many communication. A node sends a message to a group of other nodes. This form of communication is often supported by the underlying network. The IP protocol supports multicast groups with class 'D' addresses, for example.
Broadcast	Broadcast refers to a one-to-all communication. A message is sent to every node in a network. As mentioned by Tannenbaum, broadcasts can lead to inefficient communication in large networks. Again, the IP protocol supports broadcasts: Relative to the subnet mask, the broadcast address is determined by all bits of the host's part of an address set to one.
Notification	Refers to the method of event delivery. It distinguishes two sub categories: delivery technique and selection of important events/messages. *Interpretation:* Notification is used to describe special event mechanisms. It can be freely chosen for the description of a communication mechanism.

Technique	A message can be exchanged in a number of differnt ways. On the one end, an interested component can poll messages from a source, or on the contrary, events can be distributed via a central update that is uniformly delivered to all components. In between, the publisher/subscriber technique can be used, in which events are only delivered to subscribed (interested) parties. *Interpretation:* We define technique as a mandatory feature, as we interpret the property as based on a standard notification technique.
Polled	A client polls an event source (busy waiting).
Publish/Subscribe	An event source delivers message to all registered parties.
Central Update	Events are distributed to all parties.
Point-To-Point	Events are communicated between exactly two components.
Selection	This property determines if event reception can be constrained by a client or not.
Event Patterns	A receiver can select events by specifying some kind of filters.
Direct	A receiver receives all events from a chanel and has to provide a mechanism to handle the events.
State	The property describes whether communication is able to express a conversation between components, which is characterized by handling of a state. *Interpretation:* The property is mandatory for each communication mechanism. It describes whether a conversational state can be maintained between communication partners or not.
Stateless	A communication only transfers data between components, without being able to create a conversation between the communication endpoints.
Stateful	A communication can be controlled by a state describing relevant properties of a communication. In this case, a state can be changed in the callee and be accessed or modified in subsequent communication calls.
Naming	Mehta considers two subproperties of naming: structure-based naming and attribute-based naming. Unfortunately, he does not exactly describe the intended meaning of these properties, so we interpret them as defined by Coulouris [40]: The property structure-based naming refers to the general structure of name, which can be either flat or hierarchical. The property attribute-based naming refers to the fact that the names also include attributes. *Interpretation:* Naming is required if the communication partners are not statically linked to each other.
Structure-Based Naming	Structure-based describes names in a flat or in a hierarchical namespace. A flat namespace uses simple identifiers to map names to objects and is a finite space (provided that the identifier of the name is of fixed size). On the contrary, a hierarchical namespace is able to generate an infinite number of names. Examples of services with structured namespaces are Domain Name Service (DNS) and most file systems.
Hierarchical	Hierarchical composeable names.
Flat	A flat namespace.
Attribute-Based Naming	A discovery service stores not only names but also attributes to these names. Consequently, we can also search for names with a number of known attributes. This property can be augmented with particular properties of the technologies of interest.

Lifecycle	This property determines special operations that initialize and terminate data access. These operations need to be executed prior to normal business operations. *Interpretation:* The property must be selected if a special operation is required.
Initialization	A data source must be initialized before the content can be accessed.
Termination	A termination operation must be issued to terminate an interaction between components. Otherwise, consequences such as data integrity violations can occur.
Persistent	This property describes that a component's data is stored on a non-volatile medium. Contrary to the former property, which describes transient data handling, this property is relevant for communication and classification of components, as the stored data can often be accessed independently from a particular component. Furthermore, it often refers to special APIs such as JDBC or ODBC that must be used to access particular components. *Interpretation:* The property should be selected if persistency is required or provided by a mechanism.
Raw access	Raw data refers to 'uninterpreted' or proprietary data, that is not encoded using a standard format. Raw data access refers to propertary protocols and APIs that have to be used for data access. Components that provide a proprietary format are often more difficult to modify and to exchange than compoents that use standardized APIs.
File Access	Data is stored in the file system. The format of the data follows a standard format such as XML or RDF.
Database Access	Data of a component is accessed by a database API such as ODBC. Components that provide this property are often database systems.
Concurrency	The property describes whether a component needs to be synchronized with other components on critical resources, or if a component is itself a critical resource. The subproperties describe the exact kind of concurrency needed for communication. As a component can be executed in different processes or threads, this property provides information about the supported synchronization mechanism. *Interpretation:* Concurrency should be selected, if any form of concurrency is associated with one of the communication partners. It is not really relevant for the communication of two components but necessary to describe simultanuous interaction with other components. If a synchronization mechanism has to be used for communication, the property must be selected. A potential connector has to mediate concurrent access.
Semaphore	A semaphore is a concept to synchronize concurrent processes. The concept was originally developed by Dijkstra.
Rendezvous	A rendezvous is a synchronization mechanism used by Ada [8]. It is also a system call in the Plan 9 operating system [7].
Monitor	Monitors restrict access to a critical data structure to a single thread: *"A monitor encapsulates data, which can only be observed and modified by monitor access procedures. Only a single access procedure may be active at a time"* [86, p.79]. As semaphores, monitors were introduced early in computer science (see, for example, Hoare [60]).
Lock	A lock is similar to a monitor as it is used to provide mutually exclusive access to an object or another resource.

Transaction	Transactions are operations that form a logical unit. The operations are normally applied on a database system and transform a system from one consistent state to another consistent state. A transaction therefore ensures the integrity of the database. Usually transactions enforce several properties: atomicity, consistency, integrity and durability. Contrary to Mehta's taxonomy, we use other properties. Nesting is unimportant as we associate the taxonomy with ports. Thus, each transaction can be expressed as a flat transaction via a connector.
Demarcation	The property defines whether a transaction is autmatically initiated by a component or if it has to be manually declared. *Interpretation:* Demacartion is mandatory. A connector has to eventually mediate between manually and automatically initiated transactions.
Automatic	A transaction is automatically started.
Manually	A transaction needs to be started manually.
Awareness	Transaction awareness describes if a component relies on transactions. The subproperties of 'Awareness' describe the exact form of a component's support for transactions. In middleware systems the granularity of this property is often determined by operations of a component. Thus, each operation can require another transaction awareness property. We support method granularity with feature model attributes. Unfortunately, the awareness attributes of middleware technologies cannot directly be used because they are asymetric and overlapping. For example, we cannot distinguish between supported, required, and mandatory awareness as defined in the EJB specification, because if the callee delcares awareness as mandatory but the caller declares awareness as required, we get a mismatch 'false positive', because required implies mandatory awareness. Therefore, we restrict ourselves to three attributes: new, supported, and none. *Interpretation:* Awareness has to be delcared mandatory if transactions have been selected. At least on subproperty of awareness must be supported by each communication mechanism.
New	A call to a method will start a new transaction.
Supported	A method supports transactions. The transaction has to be already started.
None	The method does not support transactions.
Isolation	Isolation is one of the ACID properties. It describes the degree of isolation between transactions that are concurrently executed against a 'component', e.g. a database system. Completely isolated transactions are called 'serializable' transactions. This form of isolation does not result in 'anomalities'. However, serialized transactions are inefficient, therefore other forms of isolation were introduced. These levels usually cover three anomalities: dirty read, non-repeatable read and phantom read. *Interpretation:* Isolation is mandatory if transactions have been selected.
Committed	This isolation level does only forbid dirty-read. A dirty-read lets a transaction query data that is not committed by another transaction.
Repeatable	This level forbids dirty-read and non-repeatable reads. A non-repeatable read allows that committed modifications of other transactions become visible in a transaction. The same query therefore can result in different result sets.

Serializable	This level describes serializable transactions. No anomality occurs in this transaction level.
Security	This property represents the root property for security-related properties. *Interpretation:* Security is an optional service, which can be supported or required. Each subproperty of security is also optional and depends on the actual communication mechanism.
Authentification	Authentification refers to the identification of a user by the system: *'Authentication is the mechanism by which callers and service providers prove to one another that they are acting on behalf of specific users or systems'* [125].
Authorization	Authorization checks if a particular resource can be accessed by a user: *"Authorization mechanisms limit interactions with resources to collections of users or systems for the purpose of enforcing integrity, confidentiality, or availability constraints"* [125, p.293]. Authorization is usually enforced with Access Control Lists (ACL). However, a second concept, Capability Lists, exists that resolves authorization more efficiently. For a differentiation between the two see Miller, Ye and Shapiro [100].
Capabilities	Authorization via capability lists.
Access Control Lists	Authorization via ACLs.
Privacy	The property describes how information is secured.
Encryption	Communicated data is encrypted and third parties cannot read the exchanged information. An encryption is implemented by an appropriate cypher algorithm.
Integrity	Integrity concerns the correctness of submitted data: *"Integrity mechanisms ensure that communication between entities is not being tampered with by another party, especially one that can intercept and modify their communications"* [125, p.304].
Redundancy	We define 'Redundancy' as a form hash code that is attached to a message and ensures the correctness of this message. This form of protection is, for exmaple, used in J2EE and in IPSec.
Certificates	A certificate ensures that the data sent by a person/component belongs to the right identity. Certificates are often created by a Certificate Authority (CA).
Durability	This property restrictes the validity of security settings to either a single session or to multiple sessions. This property determines if authorization is required before each communication.
Single session	Security settings are valid for a single communication session.
Multi session	Security settings are valid for several communication sessions.
Pooling	This property describes the ability of a component to store connections and objects in a pool. This provides a faster access to these objects or connections. In general, this property is not necessary to describe communication as it does not influence the behavior but only the performance of a communication. *Interpretation:* This property is an optional service that can be used by a connector to optimize communication.
Binding	This property describes the time of allocation of a component. *Interpretation:* The property is mandatory for each communication mechanism.

Compile-Time	Two components are statically linked. This is a typical scenario for a local composition of components. Often one component is a shared library.
Run-time	Two components are bound at run-time. This is the normal scenario for distributed communication.

Taxonomy of Technological Aspects

The following table describes technology-related properties of the second taxonomy. This taxonomy is not directly used for conflict analysis but provides information about technology, language, and specification level of a component. The taxonomy was developed by Neubus [102] to support component transformation and mainly to allow searching for components based on metadata. Neubus implemented a matching algorithm for component search in the ADAM repository. We do not further describe this taxonomy as it is not used for conflict analysis.

Property Name	Description
Name	Name of the component.
Function	A textual description of the functionality of a component.
Level	The abstraction level of the component.
PIM	Platform independent component description.
PSM	Platform specific component description.
Artifact	The component is only available as an artifact.
Keywords	A collection of keywords that describe the functionality of the component.
Language	The language in which the component is written.
OS	The Operating System on which the component can be executed.
Memory consumption	The amount of storage that is required by a component.
Storage consumption	The amount of memory needed by a component.
Context	The context needed by the component for execution. The context can refer to an operating system, middleware or kind of virtual machine.
Dependency	A collection of dependencies of a component. These can either refer to other components or to hardware.

4.3 Applications of the Communication Taxonomy

We express the communication taxonomy with a Feature Model. As mentioned in Section 3.1.3, feature models originate in the area of product lines and domain analysis and serve as a description of variable and fixed features of domain entities using and-or-trees. Feature models are adequate to model a communication taxonomy for the following reasons:

- They distinguish between optional and mandatory features. This differentiation is required for conflict analysis to deal with 'unknown' values. This issues is explained in Section 6.3.
- At the same time feature models can be used for parameterized model transformations. They can be used to generate platform specific component and connector descriptions from a platform independent representation. Each feature triggers the generation of particular structure and behavior. Section 4.3.3 introduces how feature models can be used for parameterized model transformation.

In the following, we discuss how communication properties are to be attached to components and connectors. We further give an overview of conflict analysis and model transformation. Both issues are more completely discussed in chapters 6 and 8.

4.3.1 Feature Annotations

We propose to annotate properties to components and connectors to describe their requirements and abilities regarding communication. However, properties cannot be associated directly to these elements but must be related to their exposed interfaces.

For example, imagine a configuration as shown in Figure 4.4. The components C_1, C_2, and C_3 are directly annotated with features[7] expressing their abilities and requirements regarding communication.

The connectors K_1 and K_2 have to mediate the differences between the communication features exposed by the three components. Thus, they need to comply with two sets of communication features imposed by the linked components. Consequently, we cannot attach the properties directly, but must consider the relationships of the connectors K_1 and K_2 to their linked components. The same holds true for components, because of the relativity of components and connectors.

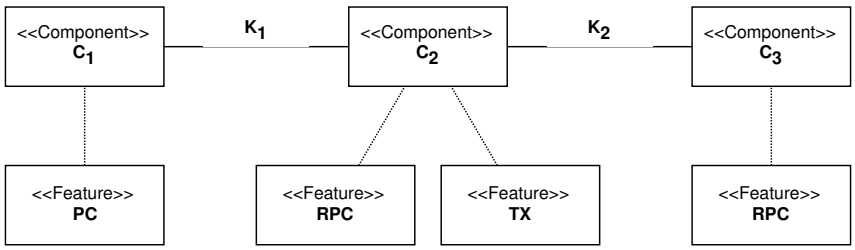

Figure 4.4: Direct Property Annotations to Components

Therefore, we bind communication features to the ports of components and connectors. Consequently, a component/connector can expose different communication requirements on each interface. Evolving our example from Figure 4.4 to an extended version, shown in Figure 4.5, K_2 and C_2 handle two feature models for their connections.

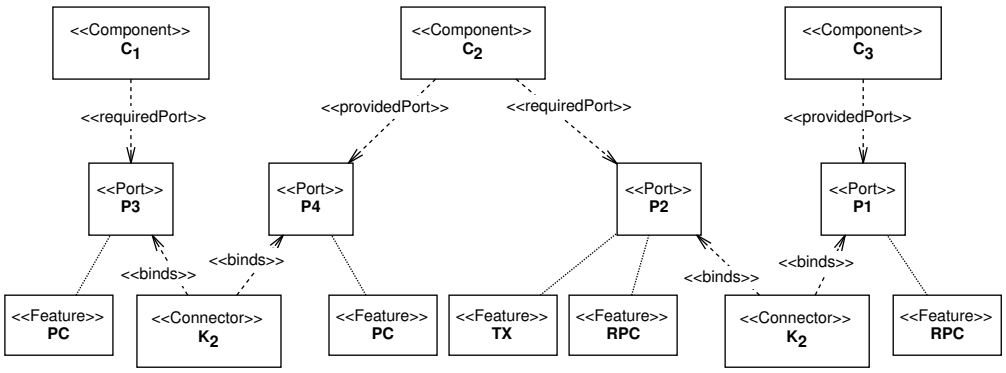

Figure 4.5: Property Annotations to Component Ports

4.3.2 Conflict Analysis

Concrete communication mechanisms are described via a subset of the properties included in the communication taxonomy. A comparison of two mechanisms checks if the properties of both mechanisms correspond to each other. Thereby, two interleaving aspects must be considered: the role of the element, for which the instance is defined and the variability of the features in the concrete taxonomies, which describe a particular communication mechanism.

[7] They are annotated with features describing a communication mechanism. The features are likely to contain variable elements that allow customization of certain aspects of the mechanism such as support for transactions.

We distinguish two kinds of elements: components and connectors. Properties attached to these elements are interpreted differently regarding the element type. Properties linked to a component describe the requirements of a component regarding communication. As explained in Section 1.2 components are defined in a context. The context provides communication mechanisms that are to be used by a component. For variable properties of the communication mechanism, a component needs to specify if a property is used or not. On the contrary, properties attached to a connector refer to the characteristics of a communication mechanism itself and therefore can include variable properties.

Considering these statements, a comparison operation needs to take into account the variability of properties[8] and the types of the elements to be compared. In Section 6.3, we therefore define compatibility matrices for the comparison of two components, two connectors, and between a component and a connector.

4.3.3 Model Transformation

The Model Driven Architecture (MDA) relies on the transformations between different model representations and therefore addresses the transformation of platform independent models (PIM) into platform specific models (PSM) and vice versa. A transformation defines a relationship between models on different abstraction levels that gets a source model and generates a target model. Figure 4.6(a) shows a transformation relationship between a PIM and several PSMs. Often this relationship is implemented as a function that takes a source model and generates a target model. The function triggers a set of fixed transformation rules that always produce the same derivation. For example, a set type is always converted into a vector for a translation of a UML model into Java source code.

As we have shown in [25] these transformations can also be represented by a single parameterized transformation. A parameterization describes relevant variability points for a transformation. It describes for example that a 'set type' can be represented in different languages such as Java and in this language that the set can be transformed into particular concepts, e.g. a Collection class or a Vector class. The selection of a particular mapping is encoded in the parameters, which are chosen by a developer. Parameterized transformation can also be based on more complex issues. In [25] we demonstrated a transformation of Enterprise JavaBeans based on optimization issues. The choices of a developer result in transformations based on J2EE patterns (see Section 8.1).

A feature instance (a feature model without variability points) triggers a particular mapping from a PIM into a PSM. Figure 4.6(b) shows a mapping by a single function t that is parameterized by feature instances fi_1, fi_2, ... fi_n.

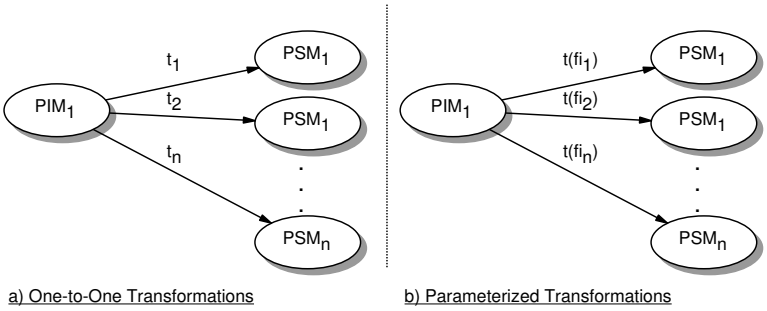

Figure 4.6: Representation of Parameterized Mappings

In our context, the communication taxonomy describes all relevant properties regarding communication. A selection of particular features results in a particular mapping into a target

[8] A component also can expose variable properties, if actual property values are 'unknown' and the underlying communication mechanism allows customization.

model, for example into a model with stubs and skeletons and a protocol description specifying synchronous communication.

4.3.4 Discussion

As mentioned above, we cannot generally state a criteria that determines if the correct properties where chosen. Accordingly, we cannot be sure that the taxonomy covers each relevant conflict in the actual form with the properties defined above. This is by no means our intention. We rather claim that a communication taxonomy is valuable because it supports a quick analysis of the communication capabilities of two components. Developers should be able to estimate the necessary integration effort to connect to components. The advantage of the approach relates to the concept of property-based conflict analysis in the context of our communication model. Furthermore, the taxonomy can be augmented and adapted to further technologies. Any missing property can be added to the taxonomy to complete conflict analysis. The downside of the approach concerns the effort to create a taxonomy instance for a concrete technology. However, a taxonomy has to be developed only once.

Another problem of the approach concerns the requirements that only symmetric and non-overlapping properties can be chosen for the taxonomy. For example, the EJB technology defines six transaction awareness attributes, including 'requires' and 'mandatory'. The properties overlap as both require a transaction for their operation. However, a 'requires' method of a callee component will create a transactional context if none is specified by a caller, whereas a 'mandatory' method will throw an exception. The problem refers to the specification of the attributes in the communication taxonomy. The conflict analysis algorithm will result in a conflict if a 'requires' method is annotated with 'requires' and a 'provides' method is annotated with 'mandatory'.

This issue conflicts with our goal of model transformation. A transformation which is based on the properties of the communication taxonomy will be less effective if we cannot include every property of a technology in the taxonomy.

A further problem of classifying communication mechanisms relates to supported but optional features that differ from implementation to implementation. The EJB standard for example defines several transaction features and security features as optional. A vendor can choose if he supports these features and in which way he supports them. We model these features as optional properties of a technology. A component that requires or provides these special optional features has to be executed on a suitable application server.

4.4 Related Work

In the following, we briefly describe related approaches that aim at a property-based characterization of components and connectors: We consider Architectural Styles, Mehta's Connector Taxonomy as well as related approaches that also aim at conflict identification or at property-based reasoning.

4.4.1 Architectural Styles

Architectural Styles were one of the early approaches in software architecture to classify systems. A style specifies the parts of a system as well as properties that need to be satisfied in a system configuration. Shaw and Garlan define an architectural style as follows:

> "An architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined" [124, p.20].

One possible application area of styles is the classification of systems regarding the composition of their constituting parts. Shaw [123; 124] provides such a classification. A style is represented

by a set of values that describe the kinds of components and connectors, the control and data flow as well as their interactions in a system. Table 4.4 shows a part of that classification and is directly taken from [123; 124]. Another application area infers resulting properties of a particular style. A style, for example, is indicative of such aspects as system reconfiguration, component exchange, and component adaptation.

In general, an architectural style takes a kind of 'macro' view of a system. It describes 'coarse grained' properties that must hold for a whole system. These are helpful if we investigate the system as a whole. However, only a few of these properties are relevant for deciding compatibility of single components. Unfortunately, the classification provided by Architectural Styles is not useful for analysing middleware technologies such as CORBA, J2EE, etc. These systems show almost no differences in the classification, as middleware systems aim for similar goals and are designed with similar architecture in mind.

4.4.2 Connector Taxonomy

Medvidovic/Mehta [93–95] propose a sophisticated taxonomy to describe communication properties of connectors. Part of this taxonomy, rendered by the ODIS repository [24], is shown in Figure 4.7. It describes properties for a procedure call. This taxonomy consists of eight connector types, each of which is described by several dimensions (complex properties) that consist of subdimensions and values. Each connector can provide four kinds of service[9]:

Communication: It enables interaction between components,

Coordination: It controls the communication between components,

Conversion: It modifies passed information,

Facilitation: It allows adding extra services.

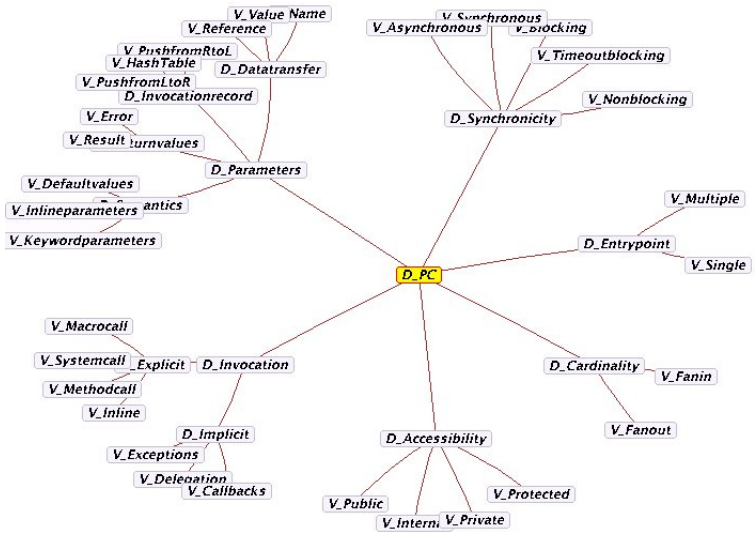

Figure 4.7: Procedure Call: Starting from the Connector Type Procedure Call, including Dimensions and Values as defined by Mehta [93]

Connectors in programming languages and middleware technologies can be described by selecting particular properties and extending the taxonomy.

[9]Services are more precisely described in Section 2.4.1.

Style	Constituent parts		Control Issues			Data Issues				Control/data Interaction	
	Components	Connectors	Topo-logy	Synch-ronicity	Binding Time	Topo-logy	Contin-uity	Mode	Binding Time	Isomorphic Shapes	Flow direction
Naive Client/-Server	programs	procedure calls / RPC	star	synch	w,c,r	star	spor, lvol	passed	w,c,r	yes	opposite
Distributed Objects	managers	remote procedure calls	arb	ls/par, synch	w,c,r	arb	spor, lvol	passed	w,c,r	no	n/a
Process-based Client/Server	processes	request/reply messages	star	synch	w,c,r	star	spor, lvol	passed	w,c,r	yes	opposite
Event Systems	processes	implicit invocation	arb	asynch, opp	i,r	arb	spor, lvol	bdcast	i,r	no	n/a
Data Centered Client/Server	managers, computations	transactions opns with history	star	asynch	w,c,r	star	spor, lvol	passed	w,c,r	yes	opposite
Key											
Topology	hier(hierarchical), arb (arbitrary), star, linear (one-way), fixed (determined by style)										
Synchronicity	seq(sequential, one thread of control), ls/par(lockstep parallel), synch (synchronous), asynch(asynchronous), opp (opportunistic)										
Binding Time	w(write-time), c(compile-time), i(invocation-time), r(runtime)										
Continuity	spor(sporadic), cont(continuous), hvol(high-volume), lvol(low-volume)										
Mode	shared, passed, bdcast(broadcast), mcast(multicast), ci/co(copy-in/copy-out)										

Table 4.4: Example of Architectural Styles as defined by Allan and Shaw [123].

The taxonomy is platform independent as it does not contain platform specific values and dimensions. Therefore, it needs to be extended to introduce technology dependent values as well as new services (Jini Leases [14], Tupelspaces [50; 56]).

The analysis of middleware technologies in the context of Mehta's connector taxonomy also reveals several problems: The terms used in the taxonomy are not explicitly defined. For a number of terms, a lot of different definitions are available. Some terms are ambiguous as different interpretations in the taxonomy can be chosen. For example the property 'Exceptions' can refer to a method that throws an exception or one that is activated because of an exception. The taxonomy describes connector types as part of the taxonomy. We feel that these types should not be included in the taxonomy because connector types are the entities that are described by properties, but they themselves are not properties.

4.4.3 Other Approaches

Other approaches aim to automatically discover mismatches based on conflicting characteristics. Today, several approaches use taxonomies [44; 93; 95; 123] to categorize architectural properties and further aim to automatically discover mismatches based on conflicting characteristics [11; 43; 72; 94; 151]. Most of these approaches concentrate on architectural mismatches and do not handle structural and behavioral specifications.

Davis et. al. [44] propose an interesting approach to deal with unknown architectural properties. They define twelve properties on three levels of abstraction and define rules between them. Based on the rules, it becomes possible to deduce unknown system properties from known properties. However, their properties are too abstract to be used for an exact estimation of communication properties of a 'system'. Furthermore, their rules are difficult to implement in terms of a deductive programming language.

Pahl [110] also considers behavior. He encodes behavior directly in description logic. This results in a more elegant language, but also does not allow a customization of specification languages.

However, most of these approaches concentrate on architectural mismatches. They do not handle technologies directly nor do they identify conflicts in structural and behavioral specifications. They can be classified into approaches using only a structure such as a table to describe properties [11; 94] and in approaches which additionally provide reasoning support [43; 44; 72]. Approaches providing reasoning support often only support a subset of properties available in the former category.

We propose to combine structural, behavioral and property specifications with a single ontology-based framework. We check structural and behavioral specifications with the help of these tools.[10] Therefore, it becomes possible for a developer to customize the framework with different kinds of formalisms. This is in particular interesting as a wide range of incompatible approaches for behavioral specifications exist that differ in their languages and in their analysis capabilities. Thus, specification languages such as Z [129], OCL [107; 133; 143], CSP [61], ACP [17], etc. can be integrated in the system, if appropriate checker tools are available as external tools. Furthermore, different type systems can also be defined for use in the framework. However, possible type systems are restricted to features of object-oriented systems, which are found in contemporary middleware systems.

[10] As we base compatibility checks on model checkers (see Section 3.2.4), we are restricted by the abilities of these tools to solve the given protocols.

Chapter 5

Conflict Analysis Framework

The objective of this thesis concerns component conflict analysis in the context of a UML based software development process. In Section 3.2 we provided an overview of a framework's architecture for conflict analysis and component transformation. The central component of this architecture regarding conflict analysis and model transformation is the Ontology-Based Framework for component analysis and transformation (OBF) component. OBF provides the necessary platform independent component models to facilitate both analysis and transformation.

Section 5.1 describes the main models of OBF that we use for component descriptions, conflict analysis and model transformation. Section 5.2 relates these models to a UML-based notation, which uses profiles to create compatible component descriptions. Section 5.2.2 describes the mapping between XMI and RDF that we defined for component exchange between the front-end and the framework. Section 5.3 exemplifies specification of platform independent components in UML. The last section discusses related work.

5.1 The Ontology-Based Framework

The framework provides models that cover the previously mentioned aspects of component specification. The models are formulated in RDFS [150]. Unfortunately, RDFS only supports a single level of metamodels. The representation of the structural model, however, requires two levels: descriptions of types and instances. Therefore, both of these levels have been folded into one model.

5.1.1 Structural Model

The structural model is divided into two parts: a vocabulary part (shown in Figure 5.1 and a configuration part (shown in Figure 5.2). The vocabulary describes element types, on which type checks are performed. It defines the set of valid elements to be used in the design of a system. The main elements are component types, connector types, interface types, and port types.

A system configuration consists of component and connector instances as well as of bindings between them. The instances can be derived from previously defined component and connector types. A configuration composes components and connectors via the 'binds' relationship. Furthermore, it provides ports to specify the multiplicity of interfaces.

Vocabulary Part. The elements of the structural model are combined by several relationships. Two of these relationships indirectly describe the dependencies between interface types and component respective connector types. Provide relations identify the services that are

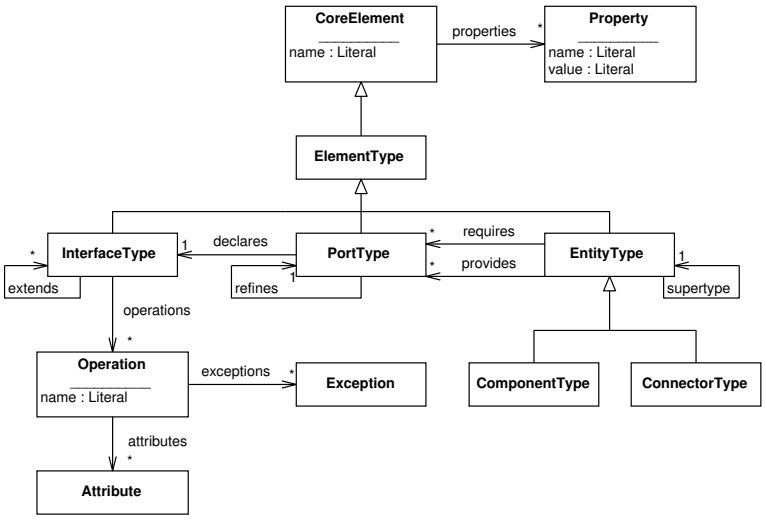

Figure 5.1: Vocabulary Part of the Structural Model

provided by a component type or a connector type. Thus, they indicate that the providing entity implements these services. Requires relations describe the required functionality of a component type or a connector type. Thus, an entity is dependent on services (entities) that offer particular interface types. In a system, these entities have to be connected with a realizing entity in order to perform their services. Port types are located between entity types and interface types. Port types are used to specify constraints (protocols and properties) relative to an interface type.

Component and connector types can participate in a *subtype* relationship (single inheritance). The core model defines the *supertype*[1] relationship in order to indicate inheritance. On the contrary, interface types use multiple inheritance. Multiple inheritance is expressed by the *extends* relationship.

Types can be instantiated by the *instanceof* relationship. We enforce that every instance of a component, connector, port, or interface has a particular type.

Despite the presence of the supertype relationship for component types and connector types, the component model used in this framework is not hierarchical as it can be found for example in Fractal [30]. A flat component model is sufficient to analyse conflicts between two components. The supertype relationship is therefore only used to copy already defined interface specifications to a new component.

However, as the framework uses a model-driven development approach, we can use a model transformation to map the flat component model used by the framework into a hierarchical component model.

Configuration Part. A system configuration describes a set of components and connectors that are linked via ports. We do not explicitly distinguish between ports and roles. We assume that a connector's role is identical to the required or provided port of a component. Therefore, we omit 'roles' in the following and describe configurations solely based on ports. The class 'Port' defines connection endpoints for these relationships. Each 'port' represents exactly one interface. In order to describe a composition we need an additional relation. This relation defines a connection between a particular component and a connector. It is specified as a *bind* relation.

The actual version of the framework considers binary connectors only. Thus, a connector binds exactly two components, or more precisely, it binds exactly one port of each component.

[1]The supertype relationship is the inverse of the subtype relationship.

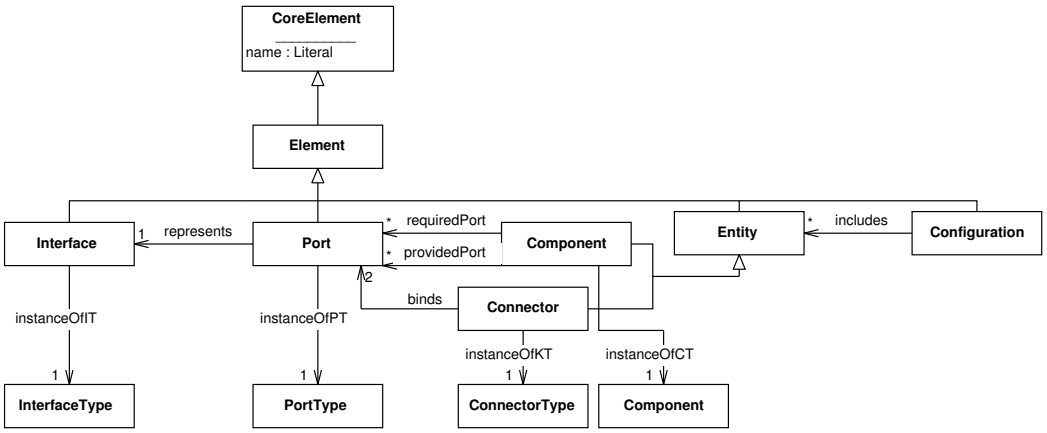

Figure 5.2: Configuration Part of the Structural Model

This approach is adequate for conflict analysis, where only a single connection between two components is considered.[2] As a connector refers directly to the components' ports, the exact linkage between the connector and two components can be calculated and connector ports need not be specified explicitly.

The configuration part, also defines the 'includes' relationship that describes a runtime configuration consisting of components that are linked via connectors.

5.1.2 Type Model

The type system of the framework is shown in Figure 5.3. It mainly consists of an abstract class 'Type' that is the root for basic types as well as for composite types. Basic types represent the primitive types used in middleware technologies. Composite types describe more complex structures such as records, functions, pairs, lists, etc.

In this work, we only provide a minimal type system. It consists of elements that are needed in order to define component substitution and component composition. Therefore, we only provide some general primitive types. This approach brings a certain degree of flexibility. The framework can be adapted for use with different technologies. A user can define a type system that is perfectly adapted to a particular situation.

The other types of the 'minimal' type system are operations, records, lists and element types. An element type specializes to component, connector, and interface types.[3]

Typing Rules

Figure 5.3 directly defines the typing rules of the framework's minimal type system. However, we have to define some additional constraints:

Operations. Operations are defined as usual: They can have several arguments, a return value and can point to several exceptions.

Exceptions. Exceptions are subtypes of records. They have no additional properties. We introduce exceptions to distinguish them from normal records in the graphical user interface of

[2]Conflict analysis identifies conflicts between two port types of two component types. Therefore port types have relations to interface types, protocol expressions and communication features.

[3]The definition of element type as a complex type would allow using component types, connector types, port types, and interface types in the definition of attributes, results from operations, and list types. We disallow these relationships as they do not correspond to the type systems of the focused object-oriented middleware.

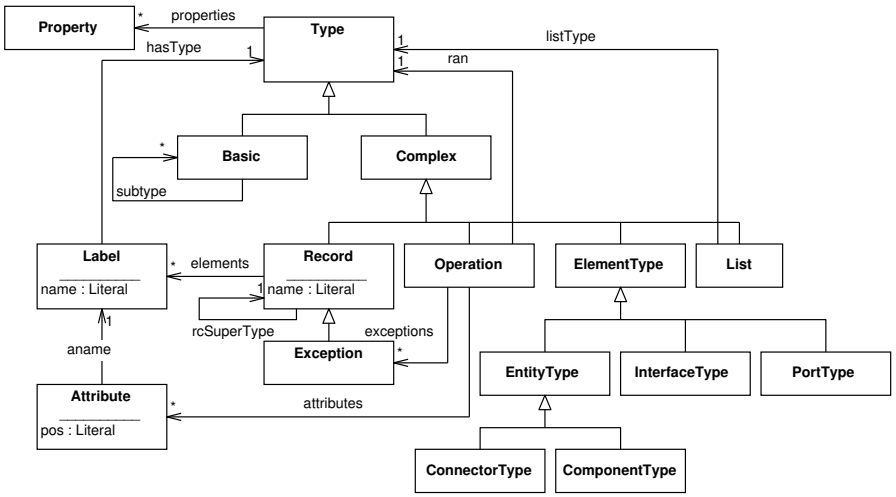

Figure 5.3: The Type Model of the Framework

the ODIS tool[4]

Records. Records are labeled sets of types. In this paper, we compare records by the names of their elements. We do not impose constraints on the ordering of their elements.

Interface. Interfaces often consist of operations and attributes. We opted to only allow operations and no attributes as attributes can be described by get and set operations. Furthermore, attributes are not directly supported by communication mechanisms such as RMI, but are emulated by implicitly defined get and set methods.

Components & Connectors. Connectors and Components both consist of interfaces only. We propose that a component must have at least one interface. A connector must offer one 'provides' and one 'requires' interfaces.

Element Types. We disallow using element types as attribute types, return values and list types. The resulting types are not supported by the type systems of J2EE and .Net and furthermore are not common in object-oriented languages.

5.1.3 Behavioral Model

Figure 5.4 shows the behavioral model of the framework. It mainly covers protocol specifications that are associated with components or connectors and constraints to express pre- and post-conditions as well as invariants. Constraints are handled by the class 'Constraint'. This class consists of three features: An 'expression' that is stated in a particular specification 'language' and the 'kind' of the constraint: invariant, pre or post-condition. Constraints can be specified for each core element.

Protocol specifications are handled by the class 'protocolExpression'. Each connector and component type can have at most one protocol specification. As in the case of constraints, there are no languages predefined. Thus, a user can customize the framework to handle protocol specifications specified in languages such as CCS, CSP, FSP, etc. Furthermore, the framework accepts a corresponding transition system for that expression. It consists of states and actions between these states. Each component or connector points transitively via a 'protocolExpression' to the

[4]In the implementation of the subtype algorithm exceptions are not distinguished from records.

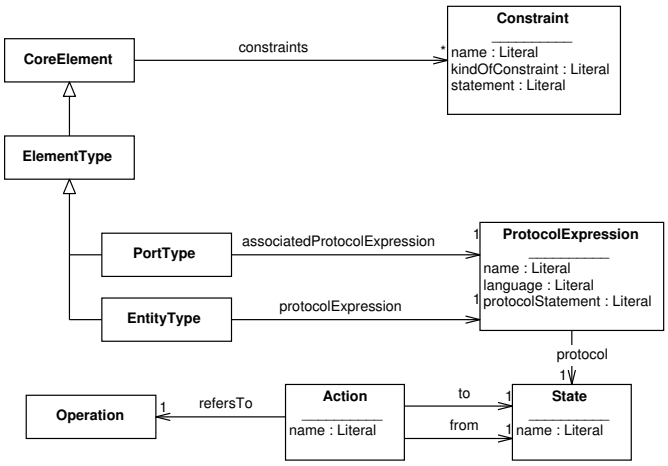

Figure 5.4: The Behavioral Model of the Framework

starting state of such a transition system. Furthermore, each action refers to an operation of the described entity. Default predicates for conflict identification are based on these transition systems.

Besides protocols that are associated with components or connectors types, protocols also can be attached to port types via the 'associatedProtocolExpression' relationship. Protocols attached to port types are used to analyse component composition.

5.1.4 Property Model

Our property model shown in Figure 5.5 defines a structure to create hierarchically connected properties. It is defined as a feature model, described in FODA [70]. Each property is described as a feature that can either be optional or mandatory. A feature can contain several sub-features. Sub-features can be grouped by two operators, 'xor' and 'or', to describe possible feature combinations. Furthermore, each feature can be associated with attributes (Feature-Attributes) to state additional requirements. 'EntityTypes' can be associated with 'Features' by two relationships: one to describe communication properties (comProps) and one to state technology-related properties (techProps).

The model also defines relationships between features: An 'excludes' relationship excludes one feature (codomain of 'excludes'), if another feature (domain) has been selected. On the contrary, the 'requires' relationship enforces the selection of a feature (codomain) if a particular feature (domain) has been selected.

5.1.5 Conflict Model

The conflict model, shown in Figure 5.6, covers all kinds of conflicts that are of interest for component integration. Each conflict description handles a particular problem class that can occur in the three models described above. Conflicts are generated by framework predicates when requested by a developer. In this case they describe potential conflicts between two components.

In particular the conflict model covers

Integrity & Type Conflicts: The former result from model instances that violate the integrity rules defined for the core models, whereas type conflicts correspond to mismatched interface types of components selected for composition. Integrity conflicts[5] are always

[5]Integrity constraints are not described in this work. They can be found in the technical report of the framework [77].

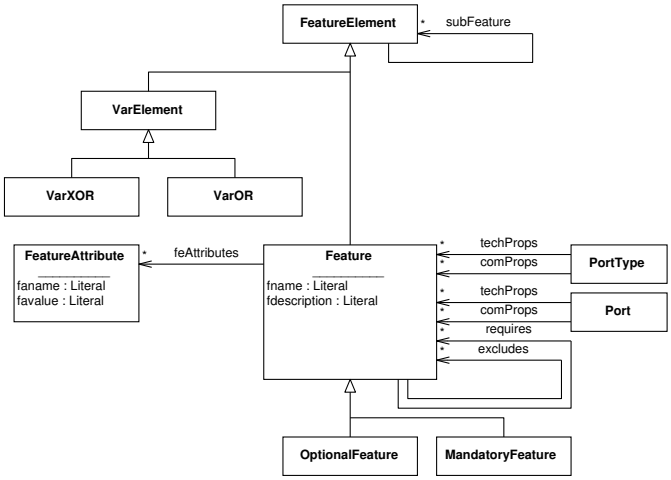

Figure 5.5: The Property Model of the Framework

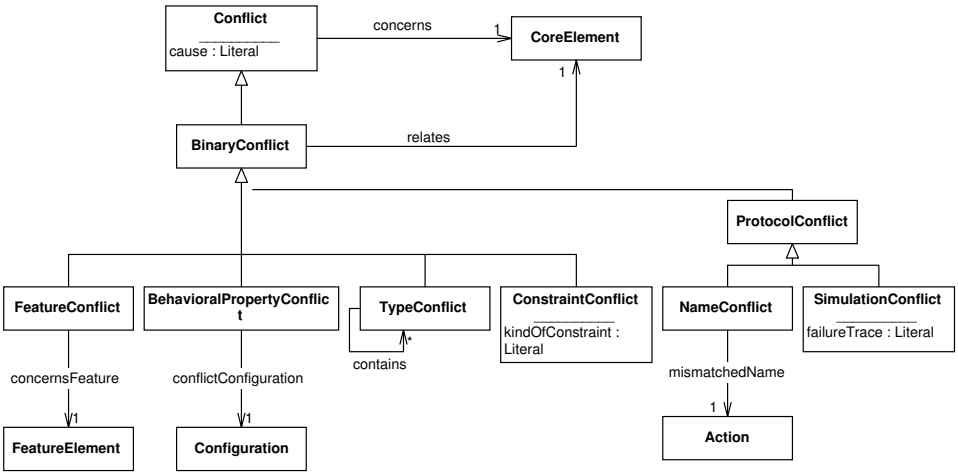

Figure 5.6: The Conflict Model of the Framework

caused by instances of model elements that violate constraints of the framework and therefore cannot be processed any further nor checked for compositional conflicts.

Type conflicts result from mismatched subtype relationships between two component types. A 'TypeConflict' is defined as a kind of 'BinaryConflict' between two component types. It is further specified by a pointer to the interface type for which no subtype relationship can be calculated.

Behavior Conflicts: In general, three kinds of behavioral conflicts are covered: constraint violations, protocol violations, and property violations (e.g. deadlocks). Protocol violations are either name conflicts, simulation or bisimulation conflicts. The former occurs if the actions of one process do not have a counterpart in the compared process and no correspondence exists that identifies different labeled actions. A prerequisite for simulation and bisimulation are identical action labels of the compared process graphs. Simulation and bisimulation conflicts are proven based on the chosen equivalence relationships.[6] As we aim to check these relationships with external tools, such as Aldebaran, FC2Tools, LTSA, the set of algorithms may be customized. However, for each algorithm a check predicate as well as a conflict generation predicate has to be defined.

Constraint violations refer to one or more constraints that are violated within the comparison of two components. This kind of conflict occurs if the pre- and post-conditions of two components are incompatible.

Besides checking for equivalence relationships between components and/or connectors, we can also verify certain properties of a composed system, consisting of a connector and several components. System properties that can be checked include progress properties as well as deadlocks. In terms of process algebra, these properties can be tested by existing tools such as LTSA [82]. These tools allow testing properties based on parallelly-composed process expressions (bound to components and a connector). A property conflict consists of a field stating the kind of property violation, a reference to a 'Configuration' and pointers to the elements (components and connectors) of a parallelly-composed system.

Property Conflicts: Property conflicts are described by the ProperyConflict class. Property conflicts refine binary conflicts by a new relationship pointing to the property (FeatureElement) that caused the conflict.

5.2 UML Representation of the Framework's Components

In order to integrate the conflict analysis framework in common software development processes that use UML for the design stage, a mapping between the framework models and UML needs to be defined. A prerequisite for the mapping concerns the identification of the framework's components in UML. We solve this requirement with UML profiles. These mainly use stereotyped UML classes to represent the framework's entities (on the platform independent level and on the platform specific level) in a one-to-one relationship. The mapping itself consists of two parts: a transformation of XMI into RDF based on the UML metamodel and the reinterpretation of component instances, specified relative to the UML metamodel against the framework's metamodels.

In this section, we describe how the framework components are principally represented in UML based on the defined PICM UML profile. The examples provided in Section 5.3 augment this introduction by a complete overview of UML representations of all main models defined by OBF. Furthermore, this section describes the mappings necessary to exchange component descriptions between UML models and the framework.

[6] Section 6.2 introduces several relationships such as strong bisimulation or branching bisimulation that can be used for conflict analysis provided that the used model checker supports the relationships.

5.2.1 Representation of the Framework's Components in UML

Each framework entity such as a component type or a connector type is represented by a single UML 'ModelElement' that is annotated with a stereotype that is named like the corresponding framework element. Some elements require a more complex treatment. For example, protocol expressions are represented as tagged values. The UML Profile that describes the OBF core models uses UML class diagrams and UML state diagrams to define the framework's elements (see Appendix B). It maps UML ModelElements and the elements of the framework one-to-one. For class diagrams, the UML model elements 'Class' and 'Dependency' are used as a basis for the mapping. For state diagrams, the UML elements 'State' and 'Transition' are used.

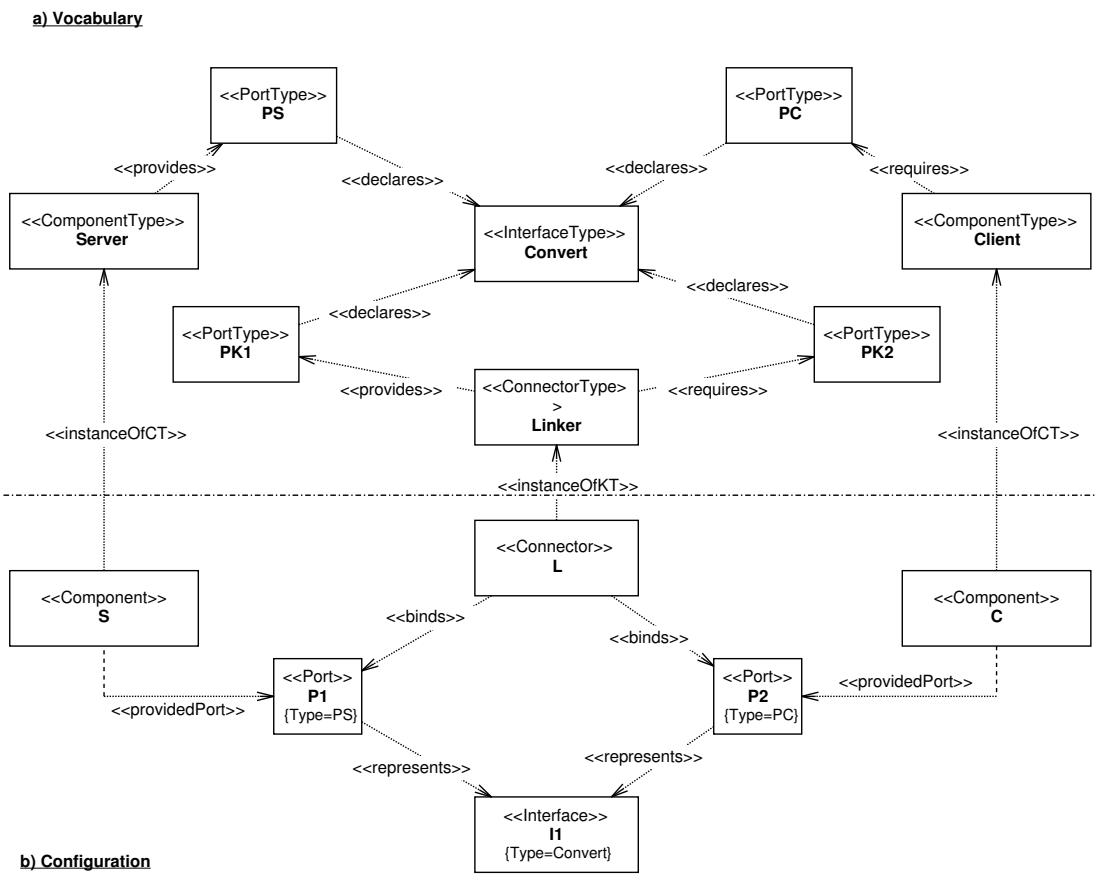

Figure 5.7: An Example Vocabulary and Configuration in UML

Figure 5.7 provides an example of the UML representation of the framework's platform independent components. Figure 5.7a describes two component types ('Server' and 'Client'), which provide and require respectively the interface type 'Convert'. This interface is linked to the components via two port types. Furthermore, a connector type 'Linker' requires and provides the 'Convert' interface type. This connector type can therefore be used to describe interactions between 'Server' and 'Client'. A framework representation of this type description can be found in Figure 5.8. As can be seen, both representations correspond to each other.

Figure 5.7(b) shows a detailed description of a configuration. It exposes a very detailed view of a configuration. However, we often do not want to describe a binding on this detailed level. Therefore, a shortcut notation is introduced. In UML, we allow describing a binding by a simple dependency between both components. The dependency has to be stereotyped with a prefix (C ::) followed by the names of the connector type and the interface type. The shortcut notation can only be used if the required and provided interfaces of both components declare the same interface type and if the ports to be bound can be identified unambiguously. The

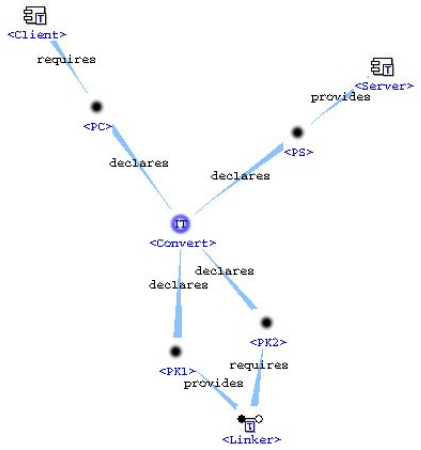

Figure 5.8: Example Vocabulary Rendered by ODIS

dependency has to point to the component that provides the interface. Figure 5.9 shows the shortcut notation for a part of example 5.7.

Figure 5.9: The Shortcut Representation of a Binding

5.2.2 Transformation between UML and RDF

The transformation of a UML model, which is specified according to one of the supplied UML profiles, is realized in two steps: In the first step a generic UML-to-RDF transformation is applied, which results in the representation of UML models as RDF instances of the UML metamodel. This step is a generic transformation of a UML model into an RDF representation. Thereby, both versions of the model are instances of a UML metamodel, which is expressed in MOF and in RDFS. The second step converts models that are defined relative to a UML metamodel into a representation which is an instance of one of the OBF metamodels.

Creating an RDF Representation of a UML model

The first step of model transformation converts UML models, which are encoded as XMI data streams, into a RDF representation. The resulting RDF model is consistent to a RDFS model, which represents the UML metamodel of the UML source model in RDFS. Figure 5.10 denotes this transformation as a '1:1 mapping' between a UML metamodel and the corresponding RDFS model on the MOF M2 layer. Based on these two metamodels, arbitrary UML models (M1 layer) can be translated into RDF.

Metamodel Transformation

The second step requires a model transformation between two different metamodels. The transformation converts models that are instances of a UML 1.4 metamodel into models relative to one of the available PICM or PSCM metamodels of OBF. This step only covers UML models that are specified according to one of the provided UML profiles. As the profiles provide an almost one-to-one representation of the framework's metamodels in UML, transformation rules orientate on stereotypes and tagged values to perform the mapping.

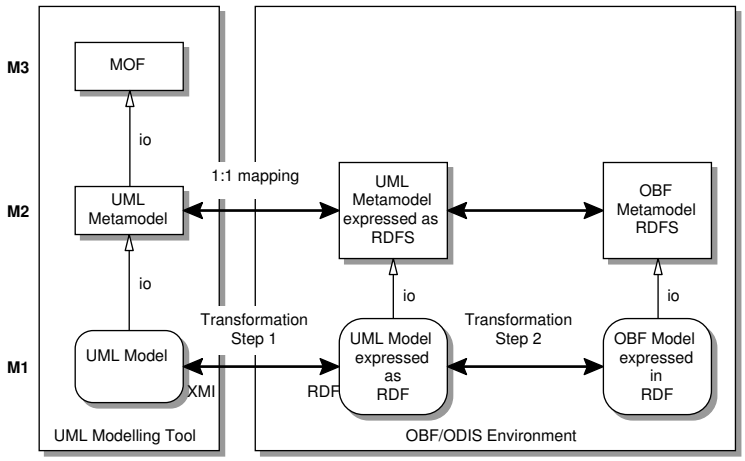

Figure 5.10: Model Transformation between UML and the Framework

A Morphism between RDF and MOF

UML models and metamodels can be handled via a MOF repository implementation. MOF defines a fixed four-layer architecture. The model transformation between UML and RDF targets at the M2 level for a metamodel transformation of UML and at the M1 level for UML model instance transformation. Both transformations are based on the MOF metamodel as defined in [105, p.3-12].While in principle the MOF metamodel can be accessed in an implementation at all levels using the reflection mechanism, we found that in practice it is advantageous to use the generated interfaces of the corresponding model to guide the transformation. Thus, we used the MOF metamodel to translate models on the M2 level. Consequently, we can not only generate UML metamodels but also other M2 metamodels defined in MOF such as CWM. For the M1 level, we instantiated MOF with the UML metamodel and therefore are able to address UML elements directly.

For the specification of the framework's components, we defined UML profiles that mainly augment UML class diagrams and UML state diagrams. Therefore, we do not need to handle every element of MOF for a translation on the M1 level. We exclude *Behavioral Feature*s from our mapping, as we are not interested in expressing the behavior interface of the MOF repository. As a consequence, the *Constant*, *Structure Field* and *Parameter* MOF metaclasses do not need to be considered because they are not relevant for component specification. *Constraint*s similarly represent a 'kind' of a dynamic aspect and are therefore also excluded. *Tag*s are not considered because the standard states that *"As a general rule, the definition of values and meanings for 'tag id' strings is beyond the scope of the MOF specification"* [105, p.2-20]. *Import*s are a visibility mechanism. RDF has global visibility and thus we do not need to map this construct.

Mapping Approach

To map MOF elements into corresponding elements in RDF/RDFS, we need to preserve the structure of the MOF model through adequate naming of elements. There are two options for this: use of MOF-IDs as unique element identifiers or use of structured names composed from a MOF Element's Namespace and Name. RDF does not limit this choice, as a URI, which it uses for identification, can encode arbitrary textual information. We thus based our choice of mechanism on the simplicity of usage for the particular transformations.

Naming based on MOF-IDs relies on a built-in mechanism of MOF, which assigns unique identifiers within the scope of one repository extent. Thus within a model, which is always fully contained in an extent, relative references are consistent. This approach has two disadvantages: Representation of model elements in RDF becomes unreadable to humans. Furthermore, com-

MOF	RDFS
$\forall x \in Class \cdot x :: name = N$	<rdfs:Class rdf:about=getURI(N)/>
$\forall x, y \in Class \cdot$ $x :: name = N,$ $x :: subclassOf = y,$ $y :: name = N2$	<rdfs:Class rdf:about=getURI(N)> <rdfs:subClassOf> <rdfs:Class rdf:about=getURI(N2)/> </rdfs:subClassOf> </rdfs:Class>
$\forall x \in DataType \cdot x :: name = N$	<rdfs:Class rdf:about=getURI(N)/>
$\forall x \in Class, y \in Attribute \cdot$ $x :: name = N1,$ $x :: contains = y,$ $y :: name = N2,$ $y :: type = T,$ $isDataType(T)$	<rdf:Property rdf:about=getURI(N2)> <rdfs:domain rdf:resource=getURI(N1)/> <rdfs:range rdf:resource=rdfs:Literal/> </rdfs:Property>
$\forall x \in Class, y \in Reference \cdot$ $x :: name = N1,$ $x :: referencedEnd = y,$ $y :: name = N2,$ $y :: type = T,$	<rdf:Property rdf:about=getURI(N2)> <rdfs:domain rdf:resource=getURI(N1)/> <rdfs:range rdf:resource=getURI(T)/> </rdfs:Property>

Table 5.1: Example Mapping Rules of a Model Transformation from MOF M2 to RDFS.

parisons are hard, because each time the MOF model is reinstantiated in the repository, IDs are randomly reassigned. As an advantage, even elements that do not have a name can be represented. We choose this approach to transform UML M1 models into RDF models.

Naming based on containment structure assumes that names of elements are unique within its namespace. We can map such structured names into a URI in the following way: $MMNamespace + NamespacePath + \# + ElementName$. The *MMNamespace* refers to a root URI, which represents the metamodel, e.g. 'http://www.omg.org/uml/1.4/'. *NamespacePath* represents the double-colon delimited path leading to the model element of interest, e.g. 'Core'. The 'ElementName' provides the fragment part of the URI. Thus our example denotes as "http://www.omg.org/uml/1.4/Core#Classifier". Because M2 models hardly contain unnamed Elements and we prefer expressiveness. We use this approach to map the M2 level.

Table 5.1 provides some of the mapping rules to transform MOF M2 models, e.g. the UML metamodel into a RDFS representation. The left-hand side of the mapping applies a simple logic-based syntax to the instances of the MOF metamodel.The right-hand side of the mapping shows RDFS encoded in XML. The properties that are used on the left-hand side correspond to the API functions provided by JMI, which is a standardized interface to manage MOF implementations.

*Attribute*s can refer not only to *Class*es, but also to *Data Type*s. In our implementation, we mapped, for practical reasons, references to data types not to the data type elements of the UML metamodel, but to *rdfs:Literal*. Thus, we can directly annotate, for example, the name of a Class as string and do not have to navigate via the 'Name' reference. The table only shows this mapping for Attributes. Model element references are treated analogously. $getURI(N)$ is shorthand for $baseNS() + packages(N) + \# + N$ and creates a URI for a MOF element.

Implementation

In the context of this work, we decided to create a RDFS representation of the UML 1.4 [108] metamodel.[7] We have realized the transformation step based on two open source APIs: Jena and Netbeans MDR. Jena handles RDF and RDFS models, the Netbeans MDR repository is a MOF-based repository, which is able to handle all kinds of models for which MOF compliant metamodels are defined.

[7]The RDFS can be obtained from URL.

In principle, all kinds of models for which a MOF compliant metamodel exist can be translated in the same way. The Netbeans MDR repository provides a reflection mechanism on metamodels that can be used to generate RDF schemata. We have implemented an EVE service that generates for arbitrary MOF 1.4 M2 metamodels a corresponding RDFS metamodel. [8]

5.2.3 Integrity Conflicts

Integrity conflicts identify component specifications that are not compliant to the rules of the core models.

These conflicts can occur in the translation of component specifications from UML into RDF, as the transformation does not prevent invalid specifications.

We do not consider the integrity rules of the framework in this work. Please refer to [77] for an exact definition of the imposed rules.

5.3 Examples

This section provides two examples to demonstrate component specification in the framework. All examples present different aspects of component specification and are shown in a UML representation and a framework representation: The dining philosophers example demonstrates component type specifications as well as a system configuration. The mortage bank example illustrates behavioral attachements and property annotations.

5.3.1 Dining Philosopher Example

Figure 5.11 shows a representation of the well known philosophers problem. The problem is characterized by five philosophers that compete against limited resources (forks).[9] The example demonstrates properties and problems of concurrent programs. Figure 5.11 describes philosophers and forks as component types, which are linked by the 'PhilConnect' connector type. The upper part shows the simple vocabulary of the two involved component types, which are linked via the IFork interface type.

The lower part of figure 5.11 shows a configuration of five philosophers and five forks. The bindings between the philosophers are expressed by the shortcut notation introduced in Section 5.2.1.

Figure 5.12 shows a screenshot from the vocabulary part of the philosopher example: The UML model was converted into RDF and the Metamodel transformed from UML into PICM.

5.3.2 Mortgage Bank Example

The second example (see Figure 5.13) shows two components that are part of a fictive customer information system. They describe part of a possible simulation of financial development for different forms of mortgage contracts. The BLContractMgmt component acts as a customer facade. It uses calculations from a set of worker components described by the BLCalc component type. Both component types are annotated with process descriptions expressed in FSP, a process algebra proposed by Kramer and Magee [86].

In the UML profile, which can be found in appendix B, protocol expressions and language attributes are defined as tagged values of a UML Class which is annotated with the PortType stereotype. In Figure 5.13, however, we use a note to describe this information, because of better readability.

[8] The *UML2RDF* metamodel translator is available for download from URL.
[9] For a brief explanation see [61].

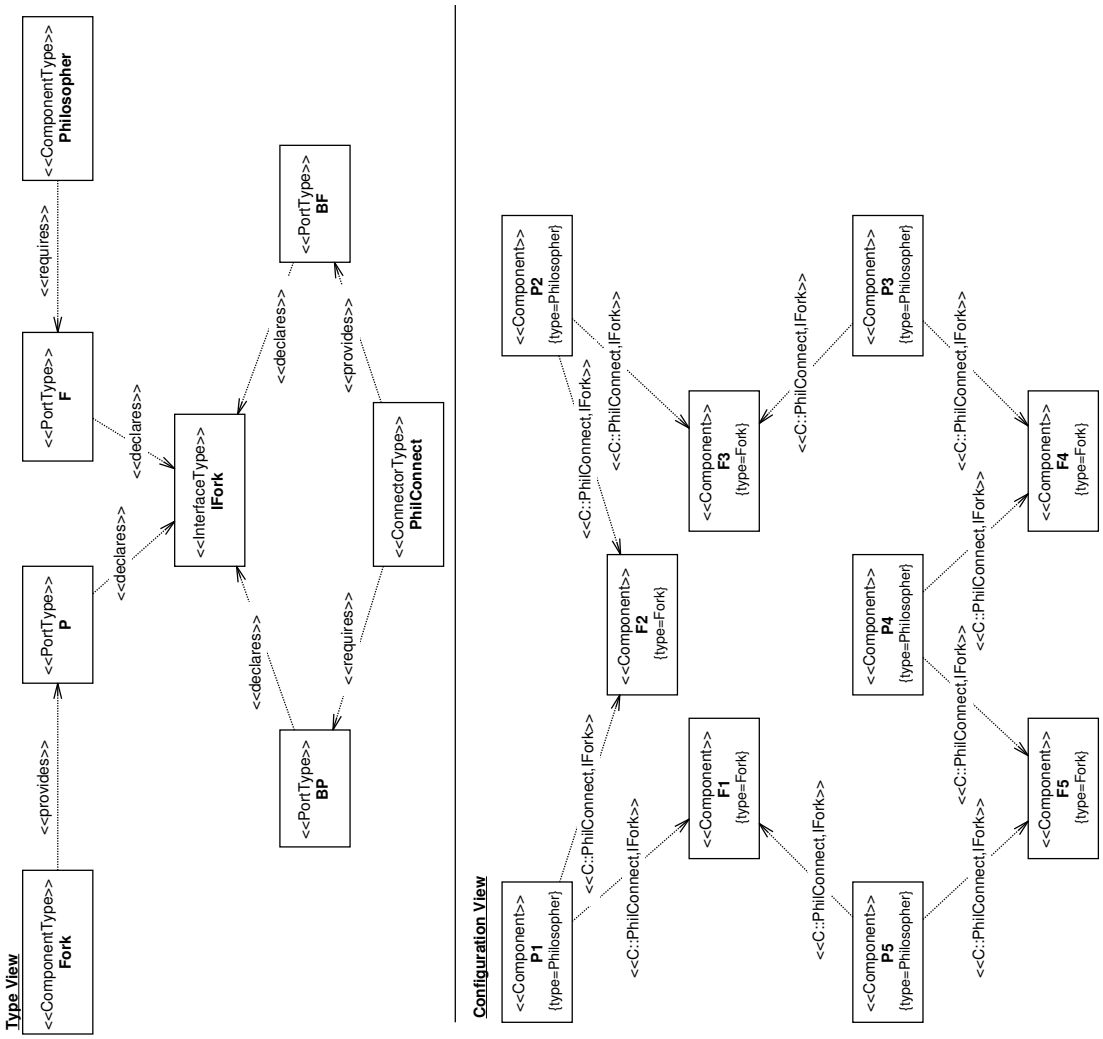

Figure 5.11: The Philosopher's Problem Described by the UML Profile

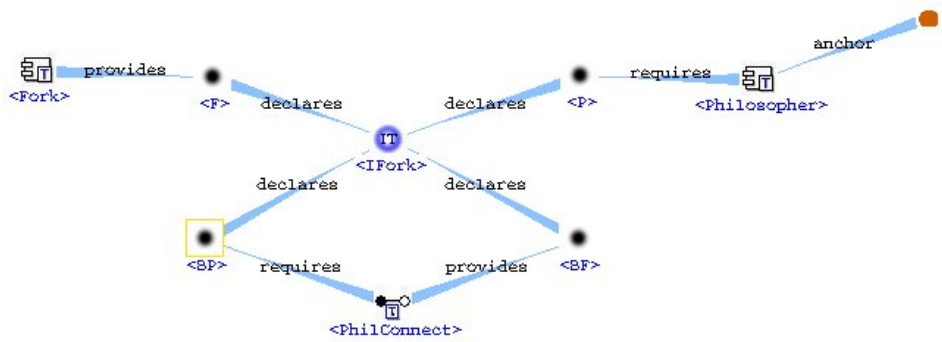

Figure 5.12: Screenshot of the Philosopher Vocabulary

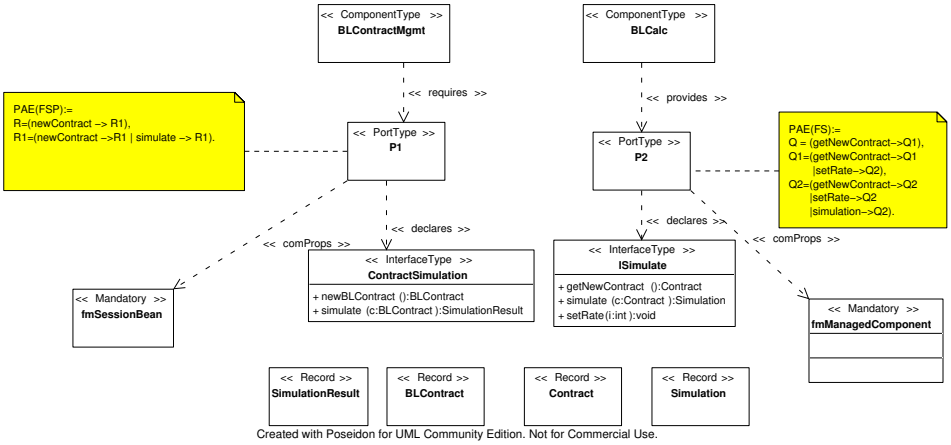

Figure 5.13: Mortgage Bank Example

Figure 5.14 shows the BLContractMgmt component rendered by ODIS. As can be seen, the protocol expression has been automatically translated into the required structure of the framework. Furthermore, the roots of the attached communication properties are shown. Both communication taxonomies are represented in figures 4.1 and 4.2.

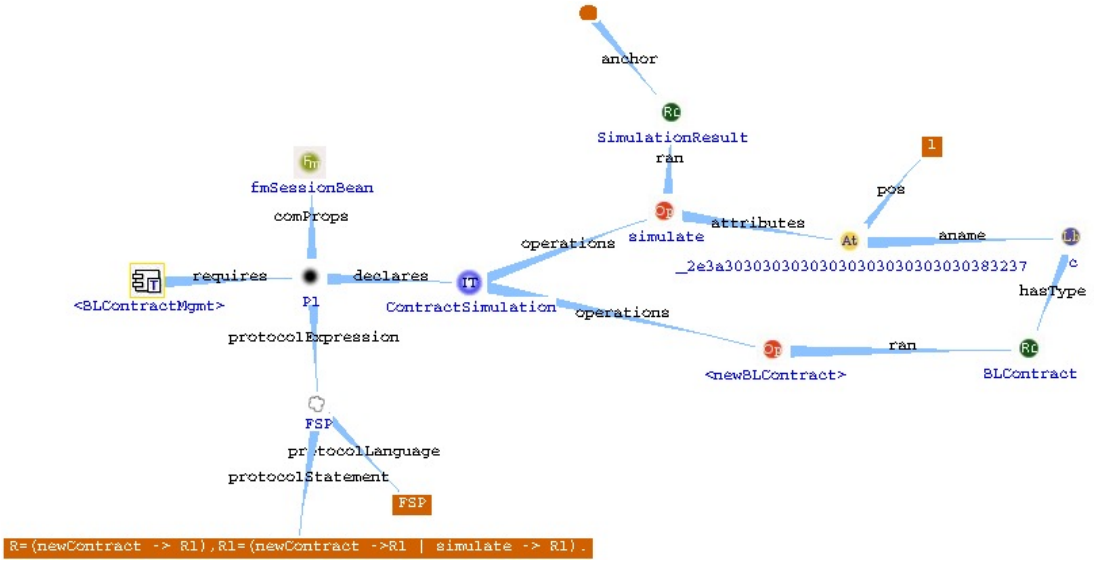

Figure 5.14: Part of the Mortgage Bank Example Rendered By ODIS

Chapter 6

Conflict Analysis

Conflict analysis is the last step in the conflict analysis process described in Section 3.3.1. It requires that components comply to several requirements: At first, components must be specified in the Platform Independent Component Model. Consequently, their types must be expressed in terms of the type system introduced in Section 5.1.2. The type system must eventually be extended by basic (primitive) types that represent primitive types of the technologies from which the components originate. Furthermore, behavior descriptions attached to the components must be stated in the same language and a tool must exist and be included in the framework that can handle the language. At present, we use FSP [86] to specify protocols as process algebra expressions and use the tools fc2tools [117], Aldebaran [47] and LTSA [86] for conflict analysis. Finally, communication requirements of components must be described by properties of the communication taxonomy defined in Section 4.2.

The framework provides two operations for conflict analysis between two components with regard to type, behavior and communication specifications:

A Compatibility operation checks if two components can interact.

A Substitutability operation checks if two components can be exchanged.

Conflict analysis is checked between two port types. Therefore, the core model associates each port type with type information (interface type), a behavior description and a communication context. For compatibility analysis of two component types, a developer has to select two component types and for each component type either a single requires or a provides port type (see Figure 6.1 for an example of a compatibility check). As mentioned above, the component model of the framework is flat. Consequently, we do not look at component dependencies between subcomponents. For conflict analysis, we consider components as black-box specifications. The developer has to choose the granularity level he is interested in.

Compatibility analysis for connectors is not defined. Substitutability is checked for all requires and provides port types of both component types or connector types. Contrary to compatibility checks, port types with the same relationship to a component type (either requires or provides) are compared to decide substitutability.

The first three sections of this chapter describe conflict analysis in the respective areas and define both operations for each area. The last section describes the implementation of the operations.

6.1 Type Analysis

Type analysis is based on subtype relationships between components. In this section, we first discuss the difference between nominal and structural subtyping, then we introduce the struc-

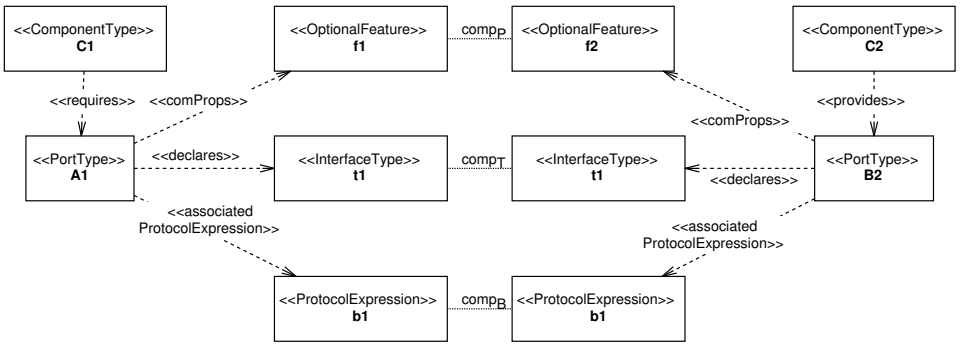

Figure 6.1: Conflict Analysis: Compatibility Checks

tural subtyping rules, specify compatibility and substitutability operations and discuss the handling of important features of object-oriented languages that must be considered.

6.1.1 Nominal and Structural Subtyping

The subtyping rules defined in the following section describe structural subtyping. Two types t_1 and t_2 are in a subtype relationship $t_1 \preceq t_2$ if they satisfy the subtyping rules. Object-oriented languages as Java or C++ often do not use this form of subtyping but nominal subtyping. Nominal subtyping declares subtypes explicitly in class declarations. Consequently, types t_1 and t_2 are only in a subtype relationship $t_1 \preceq t_2$ if t_1 references t_2 in a class declaration (e.g. in Java: **class** t_1 **extends** t_2). A discussion of the advantages and the disadvantages of nominal subtyping can be found in [112].

As we are interested in checking the compatibility of components, which eventually have been imported from different languages, we cannot rely on nominal subtyping. Instead, we use structural subtyping to check for the relationships of complex types such as component types, interfaces types, operations, etc. Only basic types are checked by using the explicit relationships given in the type model. Consequently, the results of subtype analysis for two elements defined in a single language can differ from the results obtained by a type checker of the language if both types are not explicitly related by class extensions.

The type model introduced in Section 5.1.2 also defines explicit subtype relationships as elements that are imported from object-oriented languages or from UML diagrams naturally provide this information. We keep this information because first, it results in a compact graph structure as we do not have to replicate the elements of each type for its subtypes and second, because we additionally can check for an explicit relationship between types.

6.1.2 Subtyping Rules

Conflict analysis requires subtype rules for complex types[1]:

Functions. Subtyping of functions is often defined in literature [112; 132]: A function $f_1 : T_1 \rightarrow T_2$ is a subtype of a function $f_2 : S_1 \rightarrow S_2$, written $f_1 \preceq f_2$, if the propositions $T_2 \preceq S_2$ and $S_1 \preceq T_1$ are satisfied.

Operations. Based on this definition, subtyping for operations can be easily defined. There is only one additional requirement: The exceptions of both operations also need to be in a subtype relationship.

[1] We assume that all elements (instances) in the framework are correctly typed and that additional relationships between basic types and for newly introduced complex types were given as needed.

$$\frac{\begin{array}{c} s = [1..o\_1.attribute] = [1..o_2.attribute] \\ o_1.range \preceq o_2.range \\ \forall e_1 \in o_1.exceptions \exists e_2 \in o_2.exceptions \cdot e_1 \preceq e_2 \\ \forall i \in s \cdot o_2.attribute_i.aname.hasType \preceq o_1.dom.attribute_i.aname.hasType \end{array}}{o_1 \preceq o_2}$$

The first line of the rule defines a list s containing the number of arguments of both operations. Thereby, both operations must have the same number of arguments. The second line demands a covariant relationship of the return types. The third line requires a covariant relationship between the declared exceptions of the subtype o_1 and the supertype o_2 for each exception of the subtype. Thus, a subtyped operation is not allowed to throw more exceptions or exceptions that are not subtypes of the exceptions thrown by the supertype operation. The fourth line of the rule may be read as follows: Each attribute type at position i of the subtype o_1 must by in a contravariant relationship to the attribute type at the same position of the supertype o_2. Thereby, the reference '$o_2.attribute_i.aname.hasType$' represents the navigation in the type model shown in Figure 5.3.

Exceptions. We treat exceptions as a special kind of record. Therefore, they are checked by the record subtyping rule.

Records. We define the usual rules for record subtyping. The following rule represents a combination of the 'width' rule and the 'depth' rule found in [112]. Thereby, record elements are compared by name. Thus, a record A is a subtype of another record B, if all elements of B are also available in A and the element types are in a covariant relationship.

$$\frac{\forall f \in r_1.elements \exists g \in r_2.elements \cdot f.name = g.name \wedge g.hasType \preceq f.hasType}{r_2 \preceq r_1}$$

Records are used to represent classes of object-oriented languages. We restrict the elements of records, which are used for subtype checks to operations and disallow direct manipulation of instance variables, which would demand stronger subtyping rules. We furthermore do not consider other advanced features of classes such as constructors or access attributes. For a detailed discussion of class properties refer for example to Fisher and Mitchell [48].

Interfaces. A subtyping relationship between interfaces can be defined as follows:

$$\frac{\forall o_2 \in it_2.operations \exists o_1 \in it_1.operations \cdot o_1 \preceq o_2 \wedge o_1.name = o_2.name}{it_1 \preceq it_2}$$

Thus, interface subtyping is defined in the same way as record subtyping. However, in our type model the navigation uses different paths and constraints interface type elements to operation types.

Lists. We treat lists as simple collections of objects of a particular type. The subtyping rule corresponds to array subtyping as defined in [112]. As arrays can be accessed and manipulated they are neither covariant nor contravariant.

$$\frac{l_1.listType \preceq l_2.listType \qquad l_2.listType \preceq l_1.listType}{l_1 \preceq l_2}$$

Components & Connectors. Components and Connectors consist of two sets of interfaces: a 'provides' set and a 'requires' set. We define the following subset rules:

$$\frac{\forall p_2 \in ct_2.provides.declares \exists p_1 \in ct_1.provides.declares \cdot p_1 \preceq p_2 \quad \forall r_1 \in ct_1.requires.declares \exists r_2 \in ct_2.requires.declares \cdot r_2 \preceq r_1}{ct_1 \preceq ct_2}$$

Thus component ct_1 needs to provide at least the services of ct_2 but can have additional interfaces. Furthermore, ct_1 must have at most the requirements of ct_2. Connectors are defined analogous.

6.1.3 Compatibility and Substitutability

Compatibility Two components C_1 and C_2 are compatible - written C_1 'compS' C_2 if there exist two interfaces $i_1 \in C_1.provides.delcares$ and $i_2.provides.delcares$, which are in the subtype relationship: $i_2 \preceq i_1$.

Component C_1 uses the services or part of the services of C_2. It cannot require more services in i_1 as there are available in i_2. Therefore, they are compatible if i_2 is a subtype of i_1. In this case, component instances of C_1 and C_2 can be related via a connector that binds the port instances i_1 and i_2. We define this form of connection - a single connection involving a single pair requires and provides ports - as the standard composition mechanism in distributed systems.[2]

A binding of two components C_1 and C_2 via a connector KT is valid if there exist two interface types $k_i \in KT.provides.delcares$ and $k_2 \in KT.requires.declares$ that are in the following subtype relationships to the interface types i_1 and i_2 as defined above: $k_1 \preceq i_1$ and $i_2 \preceq k_2$.

Substitutability. Two element types E_1 and E_2, which are either 'ComponentTypes' or 'ConnectorTypes', can be exchanged - written E_1 $'exchangeS'$ E_2 - if the following predicate evaluates to true:

$$\forall i_2 \in X_2.provides.declares, \exists i_1 \in X_1.provides.declares \cdot i_2 \preceq i_1 \wedge$$
$$\forall i_3 \in X_1.requires.declares, \exists i_4 \in X_2.requires.declares \cdot i_4 \preceq i_3$$
$$where\ X_1\ and\ X_2\ are\ either\ of\ type\ 'ComponentType'\ or\ of\ type\ 'ConnectorType'.$$

Consequently, a component C_1 can be exchanged by a new component C_2 if the new component provides the same or more services to the environment and requires the same or less services from the environment. The same holds true for the exchange of connectors. The predicate is equivalent to the subtyping rule of component and connector types. Again, the expression 'provides.declares' refers to the interfaces that are attached to an element E via a provides port.

Both definitions are based on a flat component model. We therefore does not consider the internal structure of two component types but only the external visible properties.

6.1.4 Constructors, Overloading and Recursion

Object-oriented languages often provide additional features that must be considered for subtype checking. Object classes are usually initialized via constructors. Constructors, are usually described in class definitions as well as in UML diagrams and therefore appear as class features

[2] This definition of a connection only considers a single connection between two ports. Other requires interfaces of a component must be connected via additional 'bindings'.

in the platform independent component model. However, they are not considered as regular members of classes [48; 69] and we therefore filter constructors so that they do not appear in subtype checking.

Another problem is method overloading. Most object-oriented languages allow defining several operations with the same name but different signatures. Therefore, names are not necessarily unique. We treat this issue with the subtyping rules as stated above. We check that for each operation o_1 of the superclass an operation o_2 of the same name exists such that $o_2 \preceq o_1$.

Object-oriented languages also allow class types as return and argument parameter of operations. This can result in cyclic type structures. We treat this issue by defining a subtyping algorithm that uses 'Trails' to remember the states already visited. A trail stores states of the form $t_1 \preceq t_2$ in a list. An algorithm for subtype checking (called by 'check *st t*') stores a tuple of the actually checked types $((st, t))$ in the trail and proceeds with recursive checks for the components of *st* and *t*. Two recursive types t_1 and t_2 are in a subtype relationship $t_1 \preceq t_2$ if the checking algorithm reaches a state that has already been visited and no contradiction was found. Trails as well as a subtyping algorithm for recursive types were defined by Cardelli and Amadio in [13] for example.

6.2 Behavior Analysis

Conflict analysis in the behavioral model is based on either the annotated labeled transition systems or on a particular process algebra, which has been provided by a user. Constraints are not compared, as these require to instantiate the framework with a particular language such as OCL or Z to define conflict predicates. Furthermore, constraints often cannot be compared automatically and are checked via theorem provers. A theorem prover often requires a skilled user to find a proof.

According to Glabbeek [140, p.559], *"a process graph is a connected, rooted, edge-labeled and directed graph"* (a transition system). Each component can be interpreted as an entity that, when deployed on a node (host system) can be associated with a process. To decide compatibility, we identify the action names of two components and compare their process graphs (transition systems). We distinguish two cases: Component composition and component substitution. In the first case, a simulation of the two process graphs needs to be proven, i.e. can the service provider handle each action of the client component. In the second case, bisimulation of the old and new component needs to be proven, i.e. can they be exchanged transparently.

Process expressions can be annotated to component, connector types or to port types. The former are used to identify conflicts for component substitution, whereas the latter are used to decide component composition.

6.2.1 Equivalence Relationships

Informally, a simulation is an asymmetric relationship between two transition systems P and Q - written $P \subseteq Q$. The relationship requires that for all transitions of P, Q offers respective transitions. Thus, each action taken by P can be simulated by Q. More formally, we derive the definition of a simulation from the definition of bisimulation as proposed by Glabbeek [140]:

Definition 17 (Simulation) *As simulation is defined as a relation R on the nodes of graphs g and h satisfying:*

the roots of g and h are related by R,

if $R(r, s)$ and $r \xrightarrow{a} r'$, then there exists a node s' such that $s \xrightarrow{a} s'$ and $R(r', s')$.

This definition is exactly the definition of 'Bisimulation Equivalence' (see next definition) as defined by Glabbeek, except that the third condition is removed. In the following, we distinguish

between different kinds of simulation (see Table 6.1). These correspond to the different definitions of bisimulation (see Table 6.2) and mainly differ in the handling of τ-actions. Informally, a silent action is an internal action of a process.

Bisimulation equivalence is defined by Glabbeek [140, p.559] as follows:

Definition 18 (Bisimulation Equivalence) *"A bisimulation ... is defined as a relation R ... on the nodes of graphs g and h satisfying:*

(1) The roots of g and h are related by R;

(2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then there exists a node s' such that $s \xrightarrow{a} s'$ and $R(r',s')$;

(3) If $R(r,s)$ and $s \xrightarrow{a} s'$, then there exists a node r' such that $r \xrightarrow{a} r'$ and $R(r',s')$. "

Glabbeek distinguished four kinds of equivalence relations between process graphs, depending whether silent actions are included and how they are handled. The four equivalence relationships are defined in Table 6.2.

In the presence of silent actions it is important that an equivalence relationship preserves the branching structure of a process graph. From the four equivalence relationships presented in the table, only branching bisimulation preserves this structure. For a detailed description refer to [16; 139; 140].

6.2.2 Compatibility and Substitutability

Compatibility. Figure 6.1 shows two components and their associated ports. The ports are linked with two process algebra expressions. We use simulation to test whether the two expressions are compatible.

Two components C_1 and C_2 are compatible regarding their protocols - written C_1 $'compB'$ C_2 if there is a simulation relationship $C_1 \subseteq_* C_2$.

The $*$ refers to one of the defined simulation relationships (strong, weak, eta, delay, branching).

A compatibility check starts with two protocol specifications. According to the defined process of conflict analysis (Section 3.3) these specifications are associated with two components that should be integrated. As we have no reason to hide any operation, we start analysis by testing for a strong simulation relationship, which contains no τ-actions. If conflicts are identified, a developer can specify correspondences between actions or use other relationships to trace the reasons for mismatches and to create solutions to these mismatches.

Substitutability. Two components C_1 and C_2 are exchangeable regarding their protocols - written C_1 $'exchangeB'$ C_2 if there is a bisimulation relationship $C_1 \leftrightarrow_* C_2$.

The $*$ refers to one of the defined bisimulation equivalences shown in Table 6.2.

Substitutability requires a bisimulation relationship to hold. Again, we start analysis by using strong bisimulation. A developer can modify the protocols by providing correspondences and possibly τ-actions to resolve the mismatches. τ-actions can be used to hide additional functions of the new component, which provide additional but optional behavior that is not required by the environment.

6.3 Analysis of Communication Properties

We assume that the communication taxonomy as well as the taxonomy instances for technology related features contain all relevant properties for communication of particular technologies of interest. Component comparison is based on the comparison of annotated features. Two entities are compatible if all features annotated to the ports of the entities are compatible. For example, in figure 6.1 C_1 and C_2 are compatible - $C_1' compP' C_2$, if the properties f_1 and f_2 are compatible.

Equivalence Relationship	Definition	Silent Actions
Strong Simulation[a]	Two graphs g and h in G^b are (strongly) similar - notation - $g \subseteq_h h$ - if there exists a relation R between the nodes of g and h (called a simulation) such that: (1) The roots of g and h are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then there exists a node s' such that $s \xrightarrow{a} s'$ and $R(r',s')$.	no
Weak Simulation	Two graphs g and h in G are (weakly) similar - notation: $g \subseteq_w h$ - if there exists an asymmetric relation R between the nodes of g and h (called a weak simulation) such that: (1) The roots are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $R(r',s)$, or there exists a path[c] $s \Longrightarrow s1 \xrightarrow{a} s2 \Longrightarrow s'$ such that $R(r',s')$.	yes
Eta Simulation	Two graphs g and h in G are η-similar - notation $g \subseteq_\eta h$, if there exists an asymmetric relation R between the nodes of g and h (called a η-simulation) such that: (1) The roots are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $R(r',s)$, or there exists a path $\Longrightarrow s1 \xrightarrow{a} s2 \Longrightarrow s'$ such that $R(r,s1)$ and $R(r',s')$.	yes
Delayed Simulation	Two graphs g and h in G are delay similar - notation $g \subseteq_d h$ - if there exists an asymmetric relation R between the nodes of g and h (called a delay simulation) such that: (1) The roots are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $R(r',s)$, or there exists a path $s \Longrightarrow s1 \xrightarrow{a} s2 \Longrightarrow s'$ such that $R(r',s2)$ and $R(r',s')$.	yes
Branching Simulation	Two graphs g and h in G are branching similar - notation: $g \subseteq_b h$ - if there exists an asymmetric relation R between the nodes of g and h (called a branching simulation) such that: (1) The roots are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $R(r',s)$, or there exists a path $s \Longrightarrow s1 \xrightarrow{a} s2 \Longrightarrow s'$ such that $R(r,s1)$, $R(r',s2)$ and $R(r',s')$.	yes

[a] The following definitions are equivalent to the definitions of bisimulation shown in Table 6.2. They are only changed to handle an asymmetric relationship between the graphs g and h. Consequently, the following definitions correspond closely - almost one-to-one - to the definition of Glabbeeck [140]

[b] Domain of process graphs

[c] $p \Rightarrow s \Rightarrow q$ is a sequence of actions with an arbitrary number of τ actions that is reduced to a after removing the τ actions.

Table 6.1: Definitions of Simulation Relationships Based on the Definitions of Glabbeek [140, pp.559-564].

Equivalence Relationship	Definition	Silent Actions
Strong Bisimulation[a]	Two graphs g and h in G[b] are (strongly) bisimilar - notation - $g \underline{\leftrightarrow}_h h$ - if there exists a symmetric[c] relation R between the nodes of g and h (called a bisimulation) such that: (1) The roots of g and h are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then there exists a node s' such that $s \xrightarrow{a} s'$ and $R(r',s')$.	no
Weak Bisimulation	Two graphs g and h in G are (weakly) bisimilar - notation: $g \underline{\leftrightarrow}_w h$ - if there exists a symmetric relation R between the nodes of g and h (called a weak bisimulation) such that: (1) The roots are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $R(r',s)$, or there exists a path[d] $s \Longrightarrow s1 \xrightarrow{a} s2 \Longrightarrow s'$ such that $R(r',s')$.	yes
Eta Bisimulation	Two graphs g and h in G are η-bisimilar - notation $g \underline{\leftrightarrow}_\eta h$, if there exists a symmetric relation R between the nodes of g and h (called a η-bisimulation) such that: (1) The roots are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $R(r',s)$, or there exists a path $\Longrightarrow s1 \xrightarrow{a} s2 \Longrightarrow s'$ such that $R(r,s1)$ and $R(r',s')$.	yes
Delayed Bisimulation	Two graphs g and h in G are delay bisimilar - notation $g \underline{\leftrightarrow}_d h$ - if there exists a symmetric relation R between the nodes of g and h (called a delay bisimulation) such that: (1) The roots are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $R(r',s)$, or there exists a path $s \Longrightarrow s1 \xrightarrow{a} s2 \Longrightarrow s'$ such that $R(r',s2)$ and $R(r',s')$.	yes
Branching Bisimulation	Two graphs g and h in G are branching bisimilar - notation: $g \underline{\leftrightarrow}_b h$ - if there exists a symmetric relation R between the nodes of g and h (called a branching bisimulation) such that: (1) The roots are related by R; (2) If $R(r,s)$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $R(r',s)$, or there exists a path $s \Longrightarrow s1 \xrightarrow{a} s2 \Longrightarrow s'$ such that $R(r,s1)$, $R(r',s2)$ and $R(r',s')$.	yes

[a] We use the term strong bisimulation instead of the original term 'Bisimulation'. Glabbeek uses this term in the online version 'http://theory.stanford.edu/'.

[b] Domain of process graphs

[c] Here, Glabbeek defines a symmetric relationship. Consequently, the second rule in each of the following definitions correspond to rules two and three of definition 18!

[d] $p \Rightarrow s \Rightarrow q$ is a sequence of actions with an arbitrary number of τ actions, that is reduced to a, after removing the τ actions.

Table 6.2: Definitions of Behavioral Equivalence Relationships as Defined by [140, pp.559-564].

6.3.1 Requirements

We distinguish three types of properties that can be associated with components to describe their communication capabilities and requirements: mandatory properties, optional properties and unsupported properties. We further interpret the meaning of optional features differently for components and connectors.

We need to deal with optional or 'unknown' values, because

(1) technologies support several properties of communication mechanisms that may be used by application components, but are not compulsory. Consequently, we must distinguish between a property that is supported as an option or that is required (mandatory).

(2) often, it is difficult to decide if a property is required by a component or not supported. Middleware specifications describe communication protocols coarse grained only. Lower level properties are often either partially described or omitted.

Therefore, if a property is optional in a communication mechanism assigned to a component and evidence whether the property is required or not supported by a component cannot be found, we must deal with 'unknown' values. We therefore distinguish between mandatory properties, which are required for a communication, unsupported properties, which are not required, and 'unknown' properties, for which no evidence for or against their usage can be found. We model 'unknown' properties as optional properties in feature models. In a feature model, an optional property can be selected for communication but it is not required. For components, we interpret optional properties as 'unknown' values and generate warnings in an analysis process.

A component composition demands that all required features have been identified. We cannot compose components for which we are not sure if particular properties such as transactions are required or not. Furthermore, a composition of two components results in a connector generation that fits the requirements of involved components. Such a connector generation is not possible if the features are not clearly defined, i.e. there must not be any free variation points (optional features) annotated to the components. Therefore, a warning indicates that a developer must manually investigate the components and identify whether the properties are used or are unsupported.

If optional features are attached to connectors, they are interpreted as supported features. If a component requires a feature (mandatory) and a connector supports that feature, we can be sure that the connector can actually communicate with the component, because it provides the required functionality.

6.3.2 Compatibility and Substitutability

Compatibility. Regarding this discussion, we describe compatibility relationships 'compP' between elements E_1 and E_2 by comparing the feature models (taxonomy instances describing a particular communication mechanism) that are attached to the ports of the elements E_1 and E_2. The ports are connected to E_1 and E_2 via a requires and a provides relationship.[3] We define the compatibility operation for the following element combinations:

(1) Component 'compP' Component

This relationship needs to be true in order to compose two components. Two components are only compatible if they are annotated with the same mandatory features and the attributes of each feature are also compatible. The relationship 'compP' yields true

[3]We only consider a component composition by a single link between a requires and a provides interface.

Component vs. Component	mandatory	optional	unsupported
mandatory	√	w	f
optional	w	w	w
unsupported	f	w	√
w = warning f = failure			

Table 6.3: Compatibility Matrix between two Components

between two components C_1 and C_2 if the following proposition holds:

$$\begin{aligned} &\forall n \in C_1.r.comProps \cdot \\ &\quad isMandatoryFeature(n) \rightarrow \\ &\quad \exists m \in C_2.q.comProps \cdot \\ &\quad n.fname = m.fname \wedge \\ &\quad isMandatoryFeature(m) \end{aligned} \tag{6.1}$$

The symbols 'r' and 'q' refer to provides and requires ports of components. The following two combinations are allowed (r = requires, q = provides) or (r = provides, q = requires). The predicate requires for each mandatory (required) feature of component C_1 a mandatory (supported) feature of the same name exists in C_2. The predicate 'isMandatoryFeature' iterates the feature model instance annotated to a component and identifies mandatory features. Therefore, it selects features that are themselves mandatory and for which all parent features are also mandatory.

Mismatches are generated for all other cases. The matrix shown in Table 6.3 shows the generated mismatches for all value combinations.

(2) Component 'compP' Connector

This relationship indicates compatibility between components and connectors. A component is compatible to a connector if the connector provides the properties required by the component. Here, optional features are interpreted differently for the component and the connector. The relationship yields true between a component C and a connector K if the following propositions hold:

$$\begin{aligned} &\forall n \in C.r.comProps \cdot \\ &\quad isMandatoryFeature(n) \rightarrow \\ &\quad \exists m \in K.q.comProps \cdot \\ &\quad n.fname = m.fname \wedge \\ &\quad (isMandatoryFeature(m) \vee \\ &\quad isOptionalFeature(m)) \end{aligned} \tag{6.2}$$

$$\begin{aligned} &\forall n \in C.r.comProps \cdot \\ &\quad isOptionalFeature(n) \rightarrow \\ &\quad \exists m \in K.q.comProps \cdot \\ &\quad n.fname = m.fname \wedge \\ &\quad isOptionalFeature(m) \end{aligned} \tag{6.3}$$

The first predicate can be read as follows: for each mandatory feature of the component must exist a corresponding property in the connector that is either mandatory or optional.

Component vs. Connector	mandatory	optional	unsupported
mandatory	√	√	f
optional	w	√	w
unsupported	f	√	√
w = warning f = failure			

Table 6.4: Compatibility Matrix between a Component and a Connector

We interpret optional values of a connector as features supported by the connector if they are requested by a client component. The second predicate matches properties even if they are defined optional by the component - it is unknown whether they are required or not - because the connector supports the properties regardless of whether they are required by the component or not. Generated mismatches are shown in Table 6.4.

Substitutability. We define substitutability similar to compatibility between the feature models attached to two elements E_1 and E_2. Contrary to the compatibility relationships, the models to be compared are attached via the same relationship to the ports of both elements. Two components are exchangeable with regard to their communication - written E_1 'exchangeP' E_2 - if for each feature model attached to a port of E_1 a compatible model for a port E_2 exists. This is another difference from the compatibility check. For substitution, all feature models attached to the ports of E_1 must have compatible models in E_2.

We define substitutability between two element combinations:

- Component 'exchangeP' Component. Two components C_1 and C_2 can be exchanged - written C_1 'exchange' C_2, if $\forall r \in C_2.X \exists p \in C_1.X$, where X refers either to a 'requires' or an 'provides' relationship, the following predicates evaluate to true:

$$\begin{aligned} &\forall n \in C_2.X.comProps\cdot \\ &\quad isMandatoryFeature(n) \rightarrow \\ &\quad \exists m \in C_1.X.comProps\cdot \\ &\quad n.fname = m.fname \wedge \\ &\quad isMandatoryFeature(m) \end{aligned} \tag{6.4}$$

$$\begin{aligned} &\forall n \in C_1.X.comProps\cdot \\ &\quad isMandatoryFeature(n) \rightarrow \\ &\quad \exists m \in C_2.X.comProps\cdot \\ &\quad n.fname = m.fname \wedge \\ &\quad isMandatoryFeature(m) \end{aligned} \tag{6.5}$$

  The predicates are almost identical to the predicate defined for component compatibility. However, they perform the check from both sides. The number of mandatory features must be invariant: The component must use the same context as the old component. A new component cannot use a slightly changed environment with different properties.

- Connector 'exchangeP' Connector. Two connectors K_1 and K_2 can be exchanged - $K1$ 'exchange' $K2$, if $\forall r \in K_1.X \exists p \in K_2.X$, where X refers either to a 'requires' or an 'provides' relationship, the following predicate evaluates to true:

Component vs. Connector	mandatory	optional	unsupported
mandatory	√	√	f
optional	√	√	√
unsupported	f	√	√
w = warning f = failure			

Table 6.5: Compatibility Matrix between two Connectors

$$
\begin{aligned}
&\forall n \in K_1.r.comProps\cdot \\
&\quad isMandatoryFeature(n) \vee \\
&\quad isOptionalFeature(n) \rightarrow \\
&\quad \exists m \in K_2.p.comProps\cdot \\
&\quad n.fname = m.fname \wedge \\
&\quad (isMandatoryFeature(m) \vee \\
&\quad isOptionalFeature(m))
\end{aligned}
\tag{6.6}
$$

The predicate matches all features of both connectors that correspond to each other and are either optional or mandatory. In each case, both connectors can deal with the required properties of the other connector. The complete matrix describing the compatibility of each feature combination can be found in Table 6.5.

6.4 Conflict Generation

Conflicts are generated in two steps: Firstly, mismatches are identified by predicates that are based on the proposed relationships and secondly, conflict statements are generated based on the identified violations. Examples of both rules are shown below.

```
forall ?c,?s,?f,?n,?pc,?ps
  unsupportedMandatoryFeatures(?c,?pc,?s,?ps,?f)<-
    getComFeatures(?c,?pc,?f)@core:Util and
    hasOnlyMandatoryParentFeatures(?f)@core:Util and
    getFeatureName(?f,?n)@core:Util and
    isFeatureNotBound(?s,?ps,?n)@core:Util.

forall C,S,PC,PS @failure(C,S,PC,PS) {
  forall ?x, ?f, ?ns
    ?ns:?x[sys:directType->core:FeatureConflict;
       core:concerns->C;
       core:relates->S;
       core:concernsFeature->?ns:?f;
       core:cause->'Mandatory feature of client unsupported by server.']
    <-
       unsupportedMandatoryFeatures(C,PC,S,PS,?ns:?f)@core:Features and
       concatConflict(?x,?f,'Failure').
}
```

The upper predicate returns all mandatory features f of a client c bound to a port pc that are unsupported by the server component s in port ps. Therefore, it first collects all communication features attached to the client component to port pc and filters for the features that are itself mandatory and for which all parent features are also mandatory (second predicate). In the

third line, it looks up the name of the feature and tests if the feature is unsupported by the server (line four).

The second rule generates conflict statements. The rule is included in a parameterized mapping with parameters C, PC, S and PS. C and S refer to component types, whereas PC and PS refer to two interfaces of the components. The body of the rule identifies conflicts, which result in new conflict statements in the head of the rule. The head generates conflict statements of the form described by the conflict model in Section 5.1.5. In the framework, a complete set of rules for each case shown in Table 6.3 are implemented.

Part III

Applications of Conflict Analysis and Transformation

Chapter 7

Examples for Conflict Analysis

The previous sections discussed the idea and theory of conflict analysis. Conflict analysis was embedded into a tool suite that supports different technologies, customization of conflict analysis capabilities and model transformation.

In this section, we provide examples of conflict analysis. We describe the analysis process, give examples for each conflict category, discuss two complex examples as well as the problems and advantages of conflict analysis.

7.1 Analysis Process and Scenario Setup

This chapter covers the 'Conflict Analysis' step of the conflict analysis process described in Section 3.3.1. Consequently, it identifies conflicts between platform independent component descriptions, which have already been transformed from UML models and from platform specific component descriptions.

Conflict analysis is an iterative process: Conflicts between two descriptions are identified using the conflict predicates introduced before. Then, a user, who reviews the produced conflicts, can specify correspondences between mismatched elements and reapply conflict analysis.

Correspondences supported by the framework include name matching and matching of operation sequences used for behavioral analysis. The correspondences have to be specified by a user, as the framework does not try to automatically identify matching operations. This would require a kind of semantic matching as proposed, for example, by Paolucci et. al. [111] in the area of web services. However, a user can augment the framework to support also semantic matching for particular application areas as the framework is technically able to handle semantic matching. A respective master thesis explores a kind of semantic matching in the context of the ADAM repository [102].

The following examples are specified in UML and checked either in the Ontology-Based Domain Repository (ODIS) framework or with respective tools such as Aldebaran and Haskell. We use the following configuration of the conflict analysis and transformation framework:

- Type conflicts are identified based on a simple subtype algorithm that implements the subtype rules as introduced in Section 6. The algorithm is implemented in Haskell. The subtype rules are based on the rules described by Pierce [112] and Amadio [13].

  For the purpose of the analysis, the framework's type system has been initialized with common primitive types and their subtype relations. The primitive type 'void'[1] is treated as the 'bottom' type of the system. It is therefore a subtype of each type. The bottom type can only appear as a return parameter of an operation.

[1] The 'void' type is exported as a primitive type from the used UML tool (Poseidon).

The example algorithm supports recursive types and includes only public operations of records for conflict analysis.

- Behavior conflicts are analysed with two external model checkers: LTSA and Aldebaran [47]. We use LTSA to check for progress and deadlock properties of a component configuration (not shown in the following examples) and Aldebaran to check for simulation and bisimulation relationships. Therefore, the examples are annotated in the respective process algebra language: FSP [86].
- Property conflicts are analysed based on the communication taxonomy introduced in Section 4.2.2. The taxonomies for particular technologies were created by Gaedicke [51] and Liao [81].

In the following, Section 7.2 exemplifies simple cases of conflict analysis, separated for each of the three categories. The following two sections then describe more complex examples of conflict analysis. Section 7.4 describes a data warehouse solution of Cedavis Ltd. that is written in Java and should be integrated with .Net technology. The chapter concludes with a summary of conflict analysis and a discussion of connector generation.

7.2 Basic Applications

In this section, we give three simple examples of conflict analysis. Thereby, we aim to create an idea of the framework's capabilities and do not aim at exhaustive examples that cover most conflict categories. The first example discusses subtype analysis, the second simulation, and the third communication conflicts. For each example, we first show the UML model, which is compatible with the UML profile of the framework, then we describe the expected and found conflicts.

7.2.1 Type Analysis

Figure 7.1 shows a simple UML component model that is defined according to the specified UML profile. The model consists of two component types, their interface types, and some additional elements describing the involved types. Thereby, UML classes without any stereotype are treated as 'Record' types of the framework's type system.

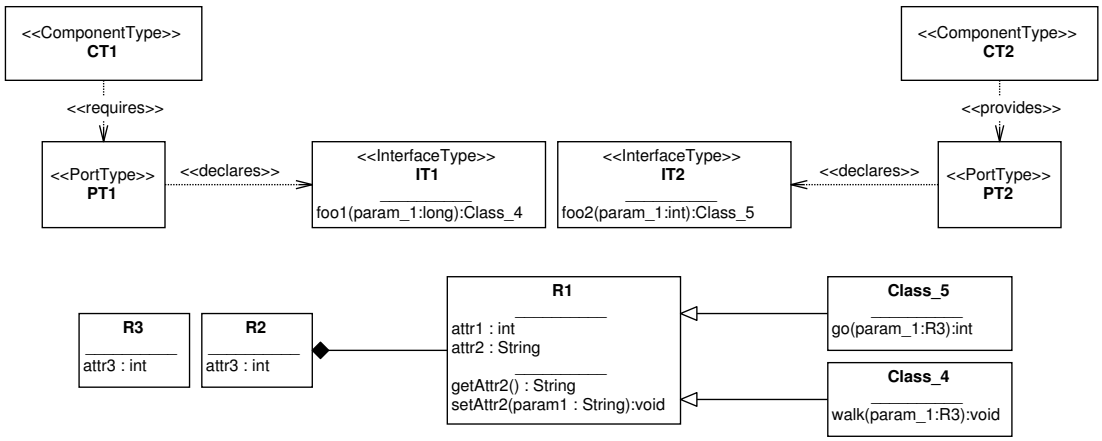

Figure 7.1: Type Example UML Model

From the UML model, the framework generates the type expressions shown in Table 7.1. The example $CT1$ and $CT2$ are defined on lines 18 and 19. The type expressions can directly be

No.	Type Expression
1.	ref_int=B "int"
2.	ref_long=B "long"
3.	ref_void=B "void"
4.	ref_string=Rc "String"NoST []
5.	ref_r2=Rc "R2"NoST [("attr3",BT ref_int),("R1",RT ref_r1)]
6.	ref_class_4=Rc "Class_4"(SubT (RT ref_r1)) [("walk",OT ref_walk)]
7.	ref_r1=Rc "R1"NoST [("attr2",RT ref_string),("setAttr2",OT ref_setattr2),("attr1",BT ref_int),("getAttr2",OT ref_getattr2)]
8.	ref_class_5=Rc "Class_5"(SubT (RT ref_r1)) [("go",OT ref_go)]
9.	ref_r3=Rc "R3"NoST [("attr3",BT ref_int)]
10.	ref_setattr2=Op "setAttr2"[("param1",RT ref_string)] (BT (ref_void)) []
11.	ref_go=Op "go"[("param_1",RT ref_r3)] (BT (ref_int)) []
12.	ref_walk=Op "walk"[("param_1",RT ref_r3)] (BT (ref_void)) []
13.	ref_foo1=Op "foo1"[("param_1",BT ref_long)] (RT (ref_class_4)) []
14.	ref_getattr2=Op "getAttr2"[] (RT (ref_string)) []
15.	ref_foo2=Op "foo2"[("param_1",BT ref_int)] (RT (ref_class_5)) []
16.	ref_it2=In "IT2"[] [(OT ref_foo2)]
17.	ref_it1=In "IT1"[] [(OT ref_foo1)]
18.	ref_ct1=Co "CT1"[] [(IT (ref_it1))]
19.	ref_ct2=Co "CT2"[(IT (ref_it2))] []

Table 7.1: Generated Types for Type Example

used with the Haskell subtype checker. The type expressions correspond to Haskell types shown in Figure 7.2.

Given Figure 7.1, we expect the following subtype relationships - written <: - to hold:

- Record $R2$ (line 5) is a subtype (<:) of $R3$ (line 9).
- Record $R3$ is a subtype (<:) of $R2$[2].
- Record $R1$ (line 7) is an subtype (<:) of $R1$ and $R2$
- Record $Class\_4$ (line 6), $Class\_5$ (line 8) <: $R1$.

A comparison of $Class\_4$ <: $Class\_5$ or $R1$ <: $Class\_4$ does not succeed as we define record subtyping via the names of the included elements. Consequently, the type checker produces the following conflict message (for $R1$ <: $Class\_4$):

> " Mismatch between record types (R1 <: Class_4). Designated Subtype Record (R1) has too few elements. Feature mismatch in Record. For walk exists no corresponding element in the subtype term."

Another conflict results from a comparison of the two operations from $Class\_4$ and $Class\_5$: *go* (line 11) <: *walk* (line 12):

> " Mismatch between operation types (go <: walk). Mismatch between return types: Basic types are not in a subtype relation. (BT (B "int")) (BT (B "void"))"

Finally $IT1$ and $IT2$ are also not in a subtype relationship ($IT2$ <: $IT1$), because of naming conflicts:

[2] As we check the subtype relationship only on the operations of classes (records) both records are empty and therefore equivalent regarding the subtype relationship.

```
1. Type definitions
> type Name = String
> data Basic = B Name
>          deriving (Show,Eq)

Void is the bottom type rendered as
simple data type from UML:

> void = B "void"

2. Complex Types
The following complex types are defined:

2a. Operations
> type Arg = (Name,Type)
> data Operation = Op Name [Arg] Type
>                  [TException]
>          deriving (Show, Eq)

> type TException = Type

2c. Interfaces
> data Interface = In Name
>                  [Supertype]
>                  [TOperation]
>          deriving (Show, Eq)

> type TOperation = Type
```

```
2d. Components
> data Component = Co Name
>                  [Interface]
>                  [Interface]
>          deriving (Show, Eq)

2e. Connectors
> data Connector = Ko Name
>                  [Interface]
>                  [Interface]
>          deriving (Show, Eq)

2f. Records
> data Record = Rc Name Supertype
>                  [Arg]
>          deriving (Show, Eq)

> data Supertype = NoST | SubT Type
>          deriving (Show, Eq)

2g. Lists
> data List = Lt Type
>          deriving (Show, Eq)

2h. Root type definition
> data Type = BT Basic
>           | OT Operation
>           | IT Interface
>           | CT Component
>           | KT Connector
>           | RT Record
>           | LS List
>          deriving (Show,Eq)
```

Figure 7.2: Haskell Type Constructors for Subtype Relationships.

> " Mismatch between interface types (IT2 <: IT1). For foo1 exists no corresponding element in the subtype term."

At this point, a developer can specify correspondences between names and then reapply conflict analysis. If he defines the two correspondences: $walk \sim go$ for the operations of $Class\_4$ and $Class\_5$, and $foo1 \sim foo2$ for the operations in both interface types, the type check deduces additional subtype relationships:

- Operation $walk$ (line 12) <: go (line 11).
- Record $Class\_4$ <: $Class\_5$.
- Operation $foo1$ (line 13) <: $foo2$ (line 15).
- Interface Type $IT1$ (line 17) <: $IT2$ (line 16).

Compatibility Checks

According to Section 6 compatibility between two components C_1 and C_2 - written C_1 'compS' C_2 is checked for a single pair of port types and their associated interface types. The components C_1 and C_2 are compatible - written C_1 'compS' C_2 if these interfaces $i_1 \in C_1.provides.delcares$ and $i_2.provides.delcares$ are in the subtype relationship: $i_2 \preceq i_1$.

Consequently, $CT1$ and $CT2$ of the example are not directly compatible. Even if we compare both components $CT1$ 'compS' $CT2$ under consideration of both correspondences defined above, the predicate fails because no subtype relation between $IT2$ <: $IT1$ can be inferred:

```
"Test compatibility (IT2<:IT1):
"Mismatch between interface types (IT2 <: IT1):
  Mismatch between operation types (foo2 <: foo2):
    Mismatch between return types:
      Mismatch between record types (Class_5 <: Class_4):
        Feature mismatch in Record:
          Mismatch between operation types (go <: go):
            Mismatch between return types:
              Basic types are not in a subtype relation.
              (BT (B "int")) (BT (B "void"))
    Mismatch between arguments:
      Basic types are not in a subtype relation.
      (BT (B "long")) (BT (B "int"))"...
```

The conflict depends on a mismatch between the return types of the operations *go* and *walk*(go) and on mismatched argument types of *foo*2 and *foo*2(foo1). Consequently, a connector is required, which maps the differently named operations between both components and resolves their subtype incompatibilities.

7.2.2 Protocol Analysis

We use the protocol descriptions of the Mortgage Bank example from section 5.3.2 to provide an example of protocol analysis. Both components in the example describe part of a possible simulation of financial development for different forms of mortgage contracts. The 'BLContractMgmt' component acts as a customer facade. It uses calculations from a set of worker components described by the 'BLCalc' component type. Both example components are annotated with the following FSP protocol expressions:

```
ContractMgmt:
R = (newContract→R1),
R1= (newContract→R1
     |simulate→R1).
```

```
BLCalc:
Q = (getNewContract→Q1),
Q1= (getNewContract→Q1
     |setRate→Q2),
Q2= (getNewContract→Q2
     |setRate→Q2
     |simulation→Q2).
```

The expressions describe behavior of both components. 'BLContractMgmt' queries the 'BLCalc' component for an contract object, which is then used to simulate the terms of a mortgage under consideration of a given rate. The corresponding transition system is shown in Figure 7.3.

For conflict analysis, we analyse the compatibility of *'BLContractMgmt' 'compB' 'BLCalc'*. We use the Aldebaran model checker to check for a simulation relationship between both process expressions. The analysis results in a conflict for the first transition: *'BLContractMgmt' can do a 'getNewContract' transition that is not available for 'BLCalc'*. The mismatch results from the different alphabets of both expressions Q and R.

Thus, a developer needs to specify correspondences between the action names. In the example, the following two correspondences can be defined: *getNewContract* $\sim$ *newContract* and *simulation* $\sim$ *simulate*.[3] As a result, we obtain the following conflict from Aldebaran:

```
" In the states (S1 = 1, S2 = 1) Only BLContractMgmt can do a "simulate"-
transition from these states."
(where S1 refers to BLContractMgmt and S2 to BLContract.)
```

[3]The 'setRate' operation has no direct counterpart in the process description of 'BLContractMgmt'.

This mismatch occurs because of a difference in specifying the rate for calculation. BLCalc first needs a 'setRate' call before it can 'simulate' a contract. BLContractMgmt provides a respective value as a parameter of 'simulate' and does not support an explicit operation to set the rate. In summary, both expressions would correspond if we define another correspondence: *simulate* $\sim$ *setRate* $\rightarrow$ *simulation*.

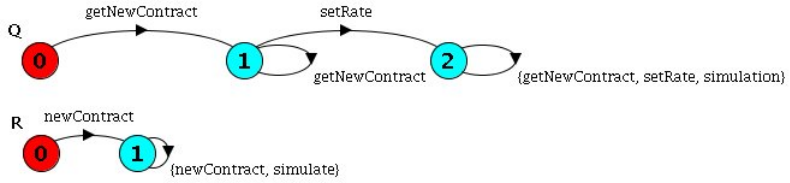

Figure 7.3: Transition system for the Mortgage Bank example.

7.2.3 Property Analysis

Figure 7.4 shows two components that we use to demonstrate the analysis of communication properties. Both components expose and require their services via a Java EntityBean communication mechanism. They are connected to a feature model instance describing the communication properties of EntityBeans. The only difference in the components concerns transaction support for their operations, which is specified by the awareness property. 'QueryControl' does not support transactions, whereas 'ComponentRepository' requires transactions. Consequently, the only conflict that should occur the both components should refer to this difference.

The annotation of an EntityBean communication model results in the addition of all communication properties of the respective communication mechanism to the component. The direct annotations to the component override the respective features of the mechanism. In the example, we get two EntityBean communication models annotated to both components. The only difference is the properties annotated to 'Transaction Awareness'. QueryControl has the property 'FeatureNone' attached to the Transaction Awareness property, whereas ComponentRepository has the property 'FeatureRequired' associated with its instance of the awareness property.

Different from the prior analysis example, property conflicts are checked directly in the framework, without external tool support. The analysis predicates can be found in Section 6.3. They are implemented in [77]. Communication properties are differentiated in two categories: failures and warnings.

A computation of the failures between the communication properties of both example components - *'QueryControl' 'compP' 'ComponentRepository'* yields two conflicts:

> " 1. Feature53Required[4] - Mandatory feature of server unsupported by client.
> 2. Feature52Never - Mandatory feature of client unsupported by server."

Both conflicts describe the difference in the transaction awareness properties from the perspectives of both components.

7.3 Federated Information System Example

The example shown in Figure 7.5 describes part of a mediator of a federated information system. A mediator is a kind of middleware that performs queries against heterogeneous distributed data sources (see for example [147]). If a client queries a mediator, the mediator first calculates which

[4]Features are enumerated in the implementation.

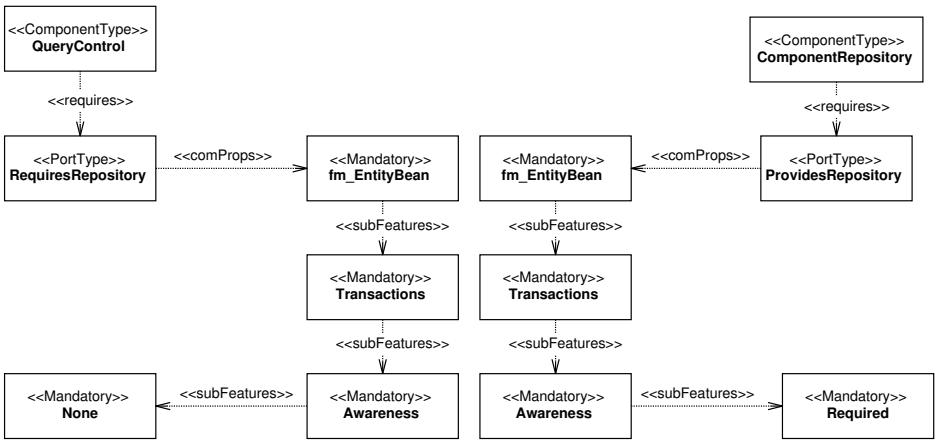

Figure 7.4: UML View on Communication Properties

data sources are capable of answering the query or part of it (Planner component). Then, it queries these sources, integrates the answers and delivers the result back to the client.

The Planner calculates its plans based on specified interfaces of the data sources. These interface descriptions are called query capabilities (QC). A query capability, shown at the bottom of Figure 7.5, consists of parameters that a data source can process as well as of result attributes returned by the data source. Query capabilities are managed by the 'QCManager' component of the mediator. The Planner uses QCs, obtained by the QCManager, to decide which data sources have to be queried. In the scenario, the old implementation of QCManager should be replaced with a new implementation. The old component was implemented as a simple Java class, which was located together with the Planner component in a single JVM. The new QCManager component should be implemented as a SessionBean. It should communicate with several Planner components in a distributed setting.

Figure 7.5 shows a UML representation of both components. The QCManager component type is a platform independent representation of a SessionBean, whereas the Planner component type is a platform independent representation of a Java class. QCManager is associated with communication properties required by Session EJBs, whereas the Planner is linked with communication properties of a normal Java component (class).

7.3.1 Type Compatibility

A check for compatibility of both components fails. The Planner component is not directly compatible with the QCManager component. For compatibility, the interface 'IQueryCapabilitiesNew' must be a subtype of 'IQueryCapabilities'. A comparison returns:

> " For getQCSet exists no corresponding element in the subtype term."

A further check between the operations of both interfaces reveals that the operations 'getQC' and 'getQCSet' are similar. However, they are not in a subtype relationship - 'getQC <: getQCSet' - because the *'Arguments are of different length.'* Operation 'getQC' requires a parameter specifying for which data sources (specified by a URL) query capabilities should be returned. The interface 'IQueryCapabilitiesNew' therefore provides the operation 'getURLs' to return the URLs to all data source that are registered in QCManager. The Planner component, however, tries to collect QueryCapabilities for all sources that are available in QCManager. Consequently, the conflict that hinders compatibility between the components - *Planner* 'compS' *QCManager* - depends on the one argument of 'getQC'if we consider both operations as corresponding to each other ($getQC \sim getQCSet$).

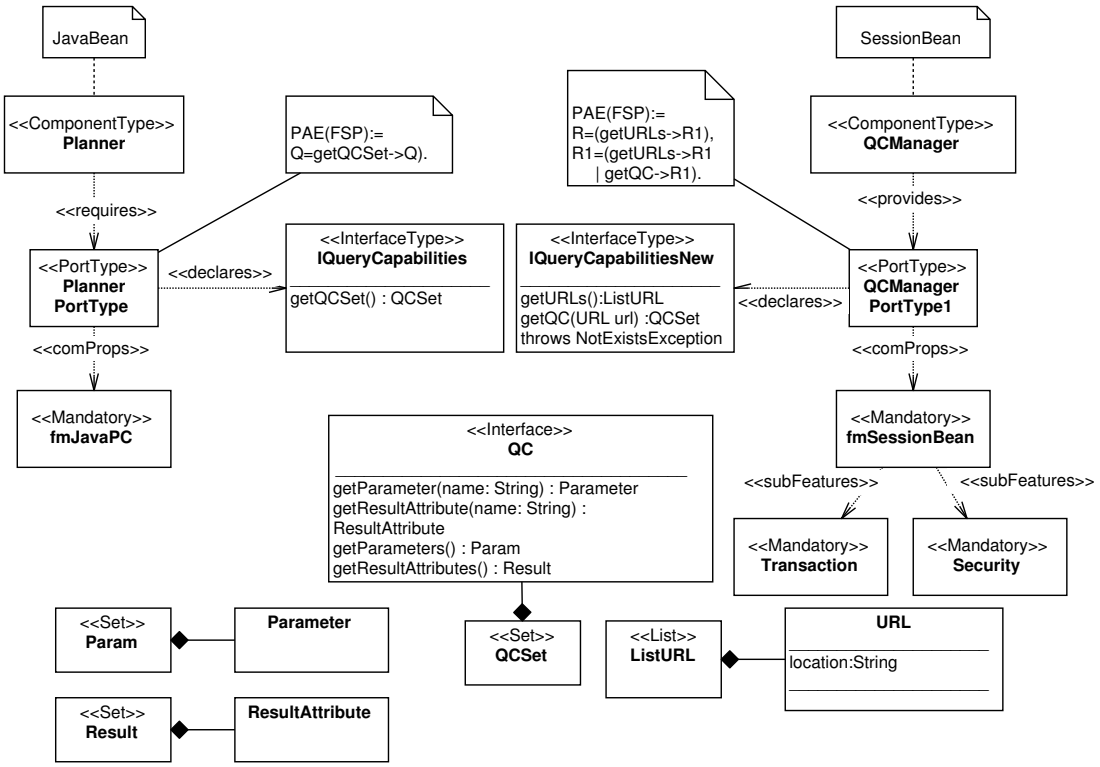

Figure 7.5: Federated Information System Example.

7.3.2 Behavioral Compatibility

In the example, two trivial process algebra expressions are attached to the components. The first expression is attached to the requires port of the 'Planner' component, whereas the second expression is bound to the provides port of 'QCManager':

Planner:	QCManager:
Q = (getQCSet→Q).	R = (getURLs→R1),
	R1= (getURLs→R1
	|getQC→R1).

Analysing both trivial expressions results in the obvious simulation conflict:

> (S1 = 0, S2 = 0)
> Only FedExampleQ.aut can do a "getQCSet"-transition from these states. (where S1 refers to Planner and S2 to QCManager.)

Consequently, there is no simulation relationship between the associated LTS. Furthermore, a correspondence between the action labels $getQCSet \sim getQC$ does not create a simulation. The correspondence $getQCSet \sim getURLs$ would result in a simulation, but is semantically invalid, because both actions refer to different operations. The obvious correspondence $getQCset \sim getURLs \rightarrow getQC$ cannot be defined because it requires an intermediate component that accepts the $getQC$ call and forwards the sequence of both actions $getURLs$ and $getQC$. Consequently, a connector is required to implement this correspondence.

7.3.3 Communication Compatibility

The analysis of the example components yields several conflicts, a part of them are shown in Figure 7.6. The main reasons for the 15 failures of Figure 7.6 can be interpreted as follows:

- Transactions are unsupported by the Planner component but required by the QCManagerComponent.
- The Planner component requires a static binding, whereas the QCManagerComponent as a distributed component requires a dynamic binding and furthermore requires a initialization based on a Naming service (JNDI).
- The QCManager requires an environment that provides security handling.

Query result

S_	0_
FailureFeature32Resources	Mandatory feature of server unsupported by client.
FailureFeature31Single	Mandatory feature of server unsupported by client.
FailureFeature34Awareness	Mandatory feature of server unsupported by client.
FailureFeature36New	Mandatory feature of server unsupported by client.
FailureFeature33Distributed	Mandatory feature of server unsupported by client.
FailureFeature38Security	Mandatory feature of server unsupported by client.
FailureFeature22Hierarchical	Mandatory feature of server unsupported by client.
FailureFeature30Nesting	Mandatory feature of server unsupported by client.
FailureFeature54Runtime	Mandatory feature of server unsupported by client.
FailureFeature21Structurebased	Mandatory feature of server unsupported by client.
FailureFeature52ImplicitRef	Mandatory feature of server unsupported by client.
FailureFeature29Transactions	Mandatory feature of server unsupported by client.
FailureFeature20Naming	Mandatory feature of server unsupported by client.
FailureFeature25ExplicitRef	Mandatory feature of client unsupported by server.
FailureFeature27Compiletime	Mandatory feature of client unsupported by server.

Figure 7.6: Property Conflicts - failures

7.4 Analysing the Cedavis Data Warehouse Tool

Cedavis Ltd. provides Business Intelligence (BI) solutions that aim at supporting a company's decision-making process. From a technical perspective Cedavis analyses and optimizes information important to a company by providing data warehouse tools.

A data warehouse [88] is a large database, often containing data used for a company's decision support. Cedavis Ltd. aims to improve decisions by providing an Online Analytical Processing (OLAP) tool that allows browsing and aggregating data via ad hoc queries. Thereby, data is represented as a n dimensional data cube. A company decides, which dimensions as well as which kind of statistics are of interest. This information constitutes the content of the data cube. The OLAP tool aggregates this information and presents it to a user. The user can explore the database by selecting dimensions of interest and defining filters on the data.

A major contribution of the Cedavis OLAP tool relates to performant calculations on issued queries. This feature is technically realized by managing the data entirely on the heap of an appropriate number of servers.

7.4.1 Example Application: Cedavis Health

The OLAP tool has been applied by customers of several business lines. Figure 7.7 shows a screenshot of the Cedavis Web Analyzer customized for business information in a hospital.

The upper part of the Analyzer displays the categories of the underlying data cube inclusive of the subdimension for each category. A user can drill down or refine the analysis by setting filters on subdimensions. The lower left part shows information relevant to calculate the results of queries. For a query a user has to select two dimensions and at least one value type. Furthermore, he can select filters as mentioned above. The result of a query is displayed on the lower right side.

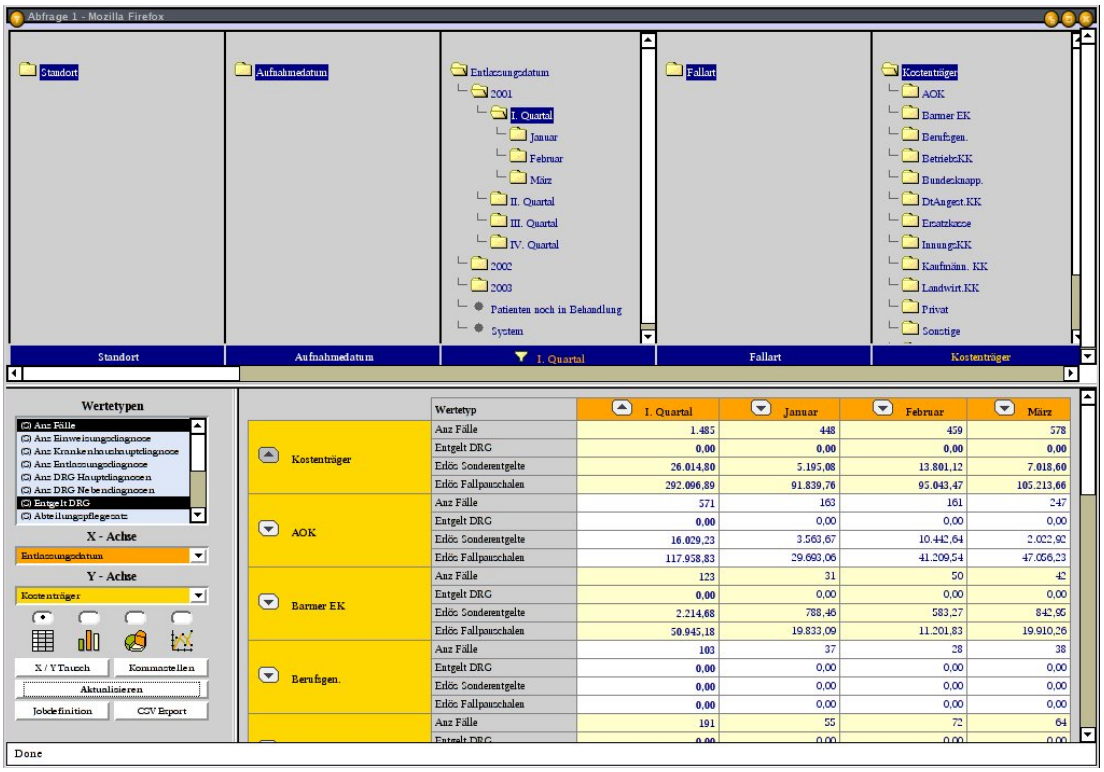

Figure 7.7: Cedavis Web Analyzer

7.4.2 Architecture

The Architecture of the Cedavis OLAP tool, shown in Figure 7.8, can be described as a four tier architecture. The client tier consists of web clients that communicate with the application server. At present, web clients are the only choice for accessing the Cedavis tool. The application tier manages user-related activities such as queries, security, job control, and reports. The persistency tier processes user queries based on the data actually stored on the heap. Furthermore, it actualizes data from a data warehouse. The lowest tier consists of life databases containing a company's business data and a data warehouse which aggregates and manages this data.

From a technical perspective, the Cedavis OLAP tool is written entirely in Java. It does not take advantage of a middleware to implement its services, but only uses socket communication between its distributed components. The main components in each tier are described as follows:

Web Analyzer. The Cedavis Web Analyzer component mainly provides a web representation to the users. It further manages connections of clients and checks for security violations. Data queried by the clients is stored in a cache to economize network transfer with the query manager. The Web Analyzer is implemented as a Java web application using Servlet and JSP technologies. Therefore, it is executed in an adequate Web Server such as the Apache Server.

Query Manager. The Query Manager is realized as a 'daemon' that listens on a given port for user requests. It processes queries and communicates with the data dispatcher to execute the queries. The Query Manager mainly implements a role-based security subsystem including fine grained access rights for dimensions and subdimensions on any level. Further, it manages job execution. Jobs are predefined queries that can be executed at a given point in time.

Processing Server. Processing servers manage the data to be analysed on the heap. They are

responsible for calculating the results of user queries. If the amount of data to be analysed exceeds the captivity that can be handled by a single Java VM, processing servers can be deployed on different servers. In this case, the data is split by means of a hashing algorithms into several data sets that are managed by different servers. Consequently, calculations are performed concurrently, which also results in a higher speed of calculation.

Query Dispatcher. The Query Dispatcher is a helper component that distributes queries and collects responses from the processing servers.

Data Warehouse. In general, a company's operating data cannot be used for analysis without converting it into a suitable representation. Data conversion results in cleansed data that is stored in the Cedavis Data Warehouse. Data conversion is performed based on rules that need to be manually configured. Cedavis uses an external Extraction Transformation and Loading (ETL) tool for data conversion.

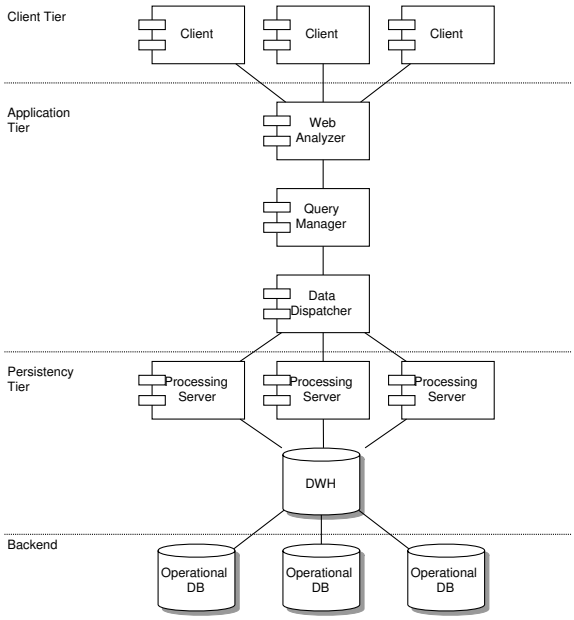

Figure 7.8: Architecture of the Cedavis OLAP Tool

7.4.3 Integration Scenario

As mentioned above, Cedavis presently provides solely a web interface to interact with its analysis tool. The Web interface is appropriate for viewing and analysing data. However, it is difficult to integrate the tool with other components to perform advanced operations, because a tight coupling with applications such as Excel is not available.

Consequently, Cedavis decided to augment its tool with a .Net client based on .Net Remoting technology. The client should be designed as a single component that allows querying the database.

Figure 7.9 shows the intended .Net component. It is designed to request queries from the database via the 'ICedavisOLAP' interface. The interface supports five operations: The 'connect' and 'close' operations perform the connection with Cedavis 'QueryManager' components. The 'getDimensions' and 'getValueType' operations query the connected processing server for available dimensions such as, for example, cost unit or time frame as shown in Figure 7.7 as well as for value types, which describe financial ratios of interest. The final operation 'olapQuery' performs queries against the connected Cedavis server. The operation requires arguments that specify two dimensions as well as a list of value types that should be extracted from the database.

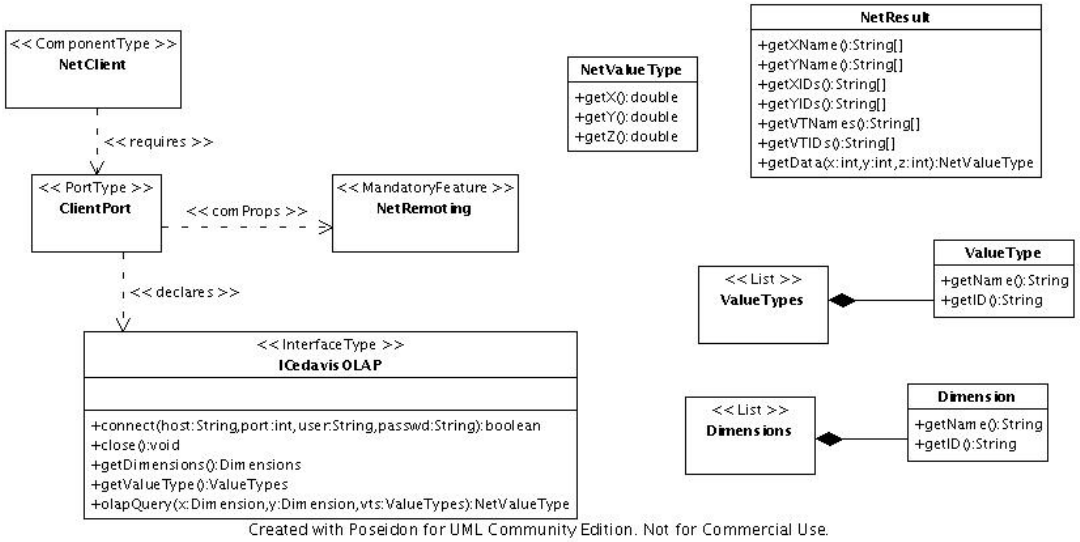

Figure 7.9: Intended .Net Client Component for the Cedavis OLAP Tool

7.4.4 Analysis Process

This section describes a conflict analysis between the Cedavis tool and the proposed Windows interface that is based on .Net Remoting. The analysis is performed according to the steps described in Section 3.3.

Reengineering of the Cedavis Tool

The analysis process requires a UML representation of the existing Cedavis OLAP tool. In this particular case, we created this representation automatically from Java source code by using the Java import function of Poseidon UML. The most interesting classes with regard to database queries are shown in Figure 7.10. The class 'CedavisClient' provides the key functionality to perform database queries on Cedavis processing servers. It consists of operations for connection initialization, retrieving information on available dimensions and value types as well as for query execution. These functionalities, however, are provided via helper objects, which can be retrieved from the 'CedavisClient' class.

For the Cedavis tool, 'CedavisClient' acts as a client of processing servers. In our scenario however, we intend to use the client as a kind of 'middle tier' to access Cedavis processing servers from the .Net world. Therefore, we defined a 'CedavisServer' component type, which provides query functionality via the 'ITServer' interface type. The interface type provides the four relevant operations for querying Cedavis processing servers. The four operations were extracted from the 'CedavisClient' class: The 'init' operation creates a connection to a Cedavis processing server. The 'getValueTypeRepository' and 'getDimensionRepository' return two objects that can be used to query the respective information about dimensions and value types as explained above. The 'getQueryManager' is used to perform the actual query.

Contrary to the intended .Net client, a query has to be assembled by instantiating a 'Query' object and assigning relevant information to it. A setup includes providing information about the dimensions of the query ('setXAxis' and 'setYAxis' operations as defined in 'Query'), value types of interest ('setSelectedVTs'), the identification of the query ('setName') and the required granularity of the query result ('setFilter'). A result consists of a two-dimensional table, which is created over the two specified dimensions. Thereby, each dimension can be recursively subdivided into parts. For example a time span can be defined as an aggregation of subdimensions such as months, which themselves are constituted of weeks, days, etc. A filter specifies, which

subdimensions should be summarized and which should be made available as columns and rows in the result.

In summary, a query consists of the following steps: Firstly, a connection to a Cedavis processing server has to be established. Secondly, dimensions and value types have to be queried from the server. Thirdly, a query has to be created and to be executed via the 'ClientQueryManager' class. This involves executing the 'addQuery' and 'getQueryResult' operations of that class. A detailed process description, which has been created with the help of Cedavis developers, can be found below.

Extraction Process

From the UML representation of the Cedavis tool including both component type specifications, a XMI file was exported and translated into a RDF model based on the UML metamodel. The model was directly transformed into the PICM as defined by the framework's core models (see Section 5.1).

Subtype Analysis

To check subtype compatibility between the client and the server component type, the subtype relationship between both component types is calculated based on the predicate 'compS' as defined in Section 6.1. A check ($ITServer <: ICedavisOLAP$) results in the following mismatch:

> " Mismatch between interface types (ITServer <: ICedavisOLAP):
> For getValueType exists no corresponding element in the subtype term.
> For close exists no corresponding element in the subtype term.
> For connect exists no corresponding element in the subtype term.
> For olapQuery exists no corresponding element in the subtype term.
> For getDimensions exists no corresponding element in the subtype term."

This is an obvious result, as we defined a subtype relationship of interface types based on name equivalence. Therefore, we needed to define correspondences between semantically equivalent operations. We decided to use the following correspondences: $close \sim init$, $connect \sim init$, $olapquery \sim getQueryManager$, $getDimension \sim getDimensionRepository$. This results in the following slightly condensed conflict between $ITServer <: ICedavisOLAP$:

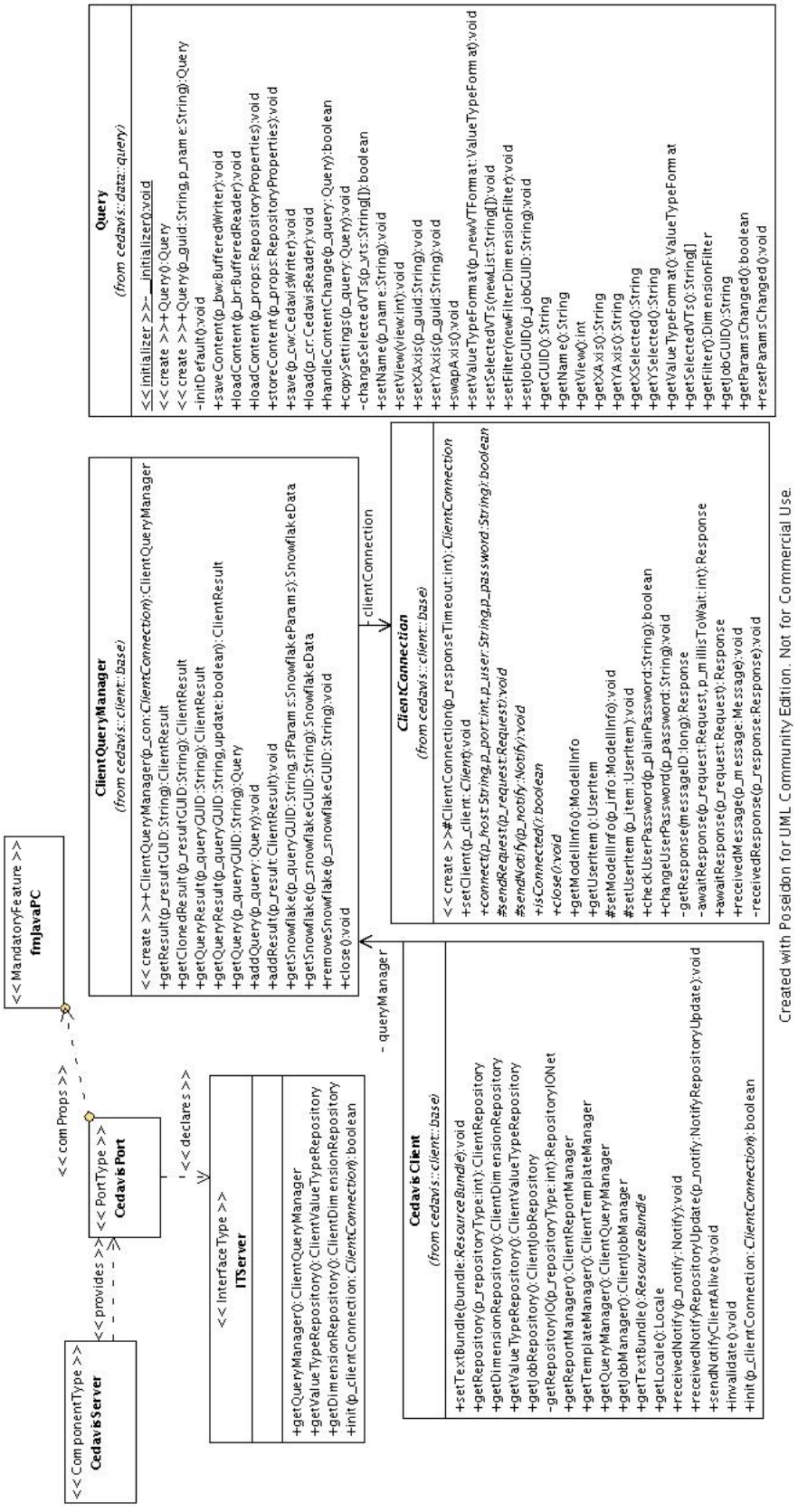

Figure 7.10: Overview Cedavis Server Component

```
" Mismatch between interface types (ITServer <: ICedavisOLAP):
  Mismatch between operation types (getValueTypeRepository <: getValueTypeRepository):
    Mismatch between return types: Structural mismatch!
  Mismatch between operation types (init <: init):
    Arguments are of different length.
    Mismatch between return types:
      Basic types are not in a subtype relation. (BT (B "boolean")) (BT (B "void"))
  Mismatch between operation types (init <: init):
    Arguments are of different length.
    Mismatch between arguments:
      Mismatch between record types (String <: ClientConnection):
        Designated Subtype Record (String) has too few elements:
        Feature mismatch in Record:
        For changeUserPassword exists no corresponding element in the subtype term.
        For setUserItem exists no corresponding element in the subtype term."
        For setModellInfo exists no corresponding element in the subtype term
        For getModellInfo exists no corresponding element in the subtype term. ...
  Mismatch between operation types (getQueryManager <: getQueryManager)
    Arguments are of different length.
    Mismatch between return types.
      Mismatch between record types (ClientQueryManager <: NetValueType):
        Feature mismatch in Record:
        For getX exists no corresponding element in the subtype term....
  Mismatch between operation types (getDimensionRepository <: getDimensionRepository):
    Mismatch between return types: Structural mismatch!"
```

Both interface types are incompatible because of a different design. In 'ICedavisOLAP' interface, we directly retrieve a list of dimensions, a list of value types, and assemble a query in a single operation. In the 'ITServer' interface, we first retrieve manager objects on which we can perform the operations. Consequently, the specified correspondences are semantically invalid and a connector is needed to resolve the indirection layer required by the 'ITServer' interface type.

However, we can check if the operations in 'ICedavisOLAP' are supported in the manager objects by searching for corresponding functions in the utility objects:

connect. The corresponding operation for connect can be found by the 'connect' operation in the ClientConnection class. A type check between both operations (*ClientConnection.connect* <: *ICedavisOLAP.connect*) succeeds.

close. Again the corresponding operation is the 'close' operation of the 'ClientConnection' class. A type check between both operations succeeds.

getDimension. The corresponding operation is 'getDimensionRootGUIDs' (not shown in the figure) defined in the 'ClientDimensionRepository'. A subtype check between both operations ('ClientDimensionRepository.getDimensionRootGUIDs <: ICedavisOLAP.getDimension') fails because the return types of both operations are different. getDimensionRootGUIDs has a string array as return value, whereas getDimension has a list of 'Dimension' as return value.

getValueType. The corresponding operation is 'getValueTypeGUIDs' (not shown in the figure) defined in the class 'ClientValueTypeRepository'. Again the return types are incompatible. Operation 'getValueType'returns a list of ValueType as shown in Figure 7.9, whereas 'getValueTypeGUIDs' returns a list of ValueTypeItem. However, this conflict is again a name mismatch between the 'getID' operation in 'ValueType' and the corresponding operation 'getGUID' in 'ValueTypeItem'.

olapQuery. For this operation no corresponding operation can be found in 'ClientQueryManager'because a query is split into two operations 'addQuery' and 'getQueryResult'.

Behavioral Analysis

A check for behavioral compatibility between the 'CedavisClient' component type and the 'CedavisServer' component type involves testing for a simulation if the behavior of the 'CedavisClient' component type can be simulated by the 'CedavisServer' component type. We use FSP to specify the valid operation orders of both components. These result in two transition systems that are checked for a simulation relationship. We use the Aldebaran model checker for the test.

The behavior descriptions are annotated to the components ports ('CedavisPort' and 'ClientPort'). We omitted the description in the diagrams because of the size of the expressions.

The behavioral description associated with the 'CedavisServer' component type is very simple. It consists of an initial 'init' transition and subsequent transitions of the remaining three actions: $Q = (init \rightarrow Q1), Q1 = (getQueryManager \rightarrow Q1 \mid getValueTypeRepository \rightarrow Q1 \mid getDimensionRepository \rightarrow Q1)$. As the operations of both interface types do not correspond to each other, it would not make sense to choose this process for compatibility analysis. Therefore, we decided to model a process description involving not only the behavior of the four operations defined in 'ITServer' but also of the relevant operations of the manager objects associated with the operations of the interface type. Furthermore, we simplified the process description to provide an impression of a 'direct' interface without intermediate manager objects. For example, a query for the dimensions of a Cedavis processing server would require two calls: The first, 'getDimensionRepository', returns a 'ClientDimensionRepository' and the second 'getDimensionRootGUIDs' on this object to retrieve the information. We modeled both operations as a single transition ('dim'). We also modeled the five operations on the 'Query' class that are necessary to pose a query.

We expressed the transition system of the CedavisServer component by a parallel composition, which describes the valid operation sequences for a communication. The resulting transition system, which we use for conflict analysis, consists of 38 states and 280 transitions:

```
// Definition of the CedavisServer Component/
Server=(init→R0),
R0= (ready1→R2 | close→Server),
R2 = (ready → R3| close→Server),
R3 = (query→R3 | close→Server).

// Abstraction for Dimensions
// getDimension = dim
DIM=(init→D1),
D1 =(dim→D2 | close→DIM),
D2 =(dim→D2 | ready1 → D3 | close→DIM),
D3 =(dim→D3 | close → DIM).

// Abstraction for Value Types
// getValueTypes = val
VAL=(init→V1),
V1 =(val→V2 | close→VAL),
V2 =(val→V2 | ready1 → V3 | close→VAL),
V3 =(val→V3 | close → VAL).

// The following five expressions represent the
// operations of the Query class.
// They can be called in any order.

// Abstraction for setName in a Query Object
Name=(ready1→N1|close→Name),
N1=(setN→N2 | close → Name),
N2=(ready → N3 | setN → N2 | close → Name),
N3=(setN → N3 | close → Name).

// Abstraction for the setXAxis operation
// in a Query object
SetX=(ready1→SetX1|close→SetX),
SetX1=(setX→SetX2 | close → SetX),
SetX2=(ready → SetX3 | setX → SetX2
      | close → SetX),
SetX3=(setX → SetX3 | close → SetX).

// Abstraction for the setYAxis operation
// in a Query object
SetY=(ready1→SetY1|close→SetY),
SetY1=(setY→SetY2 | close → SetY),
SetY2=(ready → SetY3 | setY → SetY2
      | close → SetY),
SetY3=(setY → SetY3 | close → SetY).

// Abstraction for the setFilter operation
// in a Query object
Fil=(ready1→Fil1|close→Fil),
Fil1=(setFil→Fil2 | close → Fil),
Fil2=(ready → Fil3 | setFil → Fil2
      | close → Fil),
Fil3=(setFil → Fil3 | close → Fil).

// Abstraction for the setSelectedVTs operation
// in a Query object
VT=(ready1→VT1|close→VT),
VT1=(selVTs→VT2 | close → VT),
VT2=(ready → VT3 | selVTs → VT2
      | close → VT),
VT3=(selVTs → VT3 | close → VT).

// Definition of the transition system for the Cedavis Server Component
||CedavisServer=(Server||DIM||Name||SetX||VAL||SetY||Fil||VT).
```

The protocol expression of the Client involves the operations of the 'ICedavisOLAP' interface type:

```
CedavisClient = Q0,
Q0 = (init → Q1),
Q1 = (close → Q0
      |dim → Q2
      |val → Q5),
Q2 = (close → Q0
      |dim → Q2
      |val → Q3),
Q3 = (close → Q0
      |dim, val → Q3
      |ready1 → Q4),
Q4 = (close → Q0
      |dim, query, val → Q4),
Q5 = (close → Q0
      |dim → Q3
      |val → Q5).
```

We already used the same action labels for both process descriptions to compare the processes for a simulation relationship. A check with the Aldebaran model checker results in the following conflict:

sequence 1:
initial states
(S1 = 0, S2 = 0)
"init"
(S1 = 1, S2 = 1)
"dim"
(S1 = 2, S2 = 37)
"val"
(S1 = 3, S2 = 3)
"ready1"
(S1 = 4, S2 = 4)
Only CedavisClient.aut can do a "query"-transition from these states

The identified mismatch results from the fact that the CedavisClient component does not use a 'Query' object to initialize a query. It directly assembles the required parameters as attributes in the call. Therefore, it does not use the five actions for initialization of a query object. However, this is the only mismatch. If we define a correspondence for the query action of the CedavisClient, which consists of a sequence of actions for explicit query initialization, e.g. (query $\sim$ $setN \rightarrow setX \rightarrow setY \ldots$), we get a simulation relationship between both process descriptions.

In summary, both components are incompatible regarding their types and their behavior. The root cause for the mismatch can be determined by the usage of manager objects in the CedavisServer component. A connector has to provide structure and behavior to establish a proper communication.

Communication Analysis

Both components differ in their communication mechanism. The CedavisClient component uses the .Net Remoting mechanism whereas the CedavisServer uses normal Java procedure calls. The taxonomy instance for .Net Remoting is defined in the appendix (Figure C.4), whereas the instance that describes a Java procedure call can be found in Figure 4.2.

An analysis of the differences between communication mechanisms result a number of conflicts that can be summarized in two points:

(1) Both mechanisms have different assumptions about the distribution of communication. Java Procedure Calls (PCs) assume local communication in a single JVM, whereas .Net Remoting aims at distributed communication.

(2) Consequently, Java PCs do not support name lookup and do not support the 'At-Most-Once' delivery semantics for distributed calls.

7.5 Summary of Conflict Analysis

This chapter provided several examples for conflict analysis of small and medium-sized integration problems. The examples were discussed by directly using the framework (ODIS/OBF) for conflict analysis. A front-end UML tool was only used to generate the UML models. In this case, a developer directly use the framework to investigate compatibility and substitutability of components. She can enter the respective analysis predicates, can further investigate the causes of mismatches and can specify correspondences between names to modify analysis results.

The main point of this chapter was to demonstrate that conflict analysis is an iterative process. A simple query for deciding the compatibility of two components does not provide enough information to understand the causes of a conflict. A developer needs to interpret the results. Iterative queries against particular types and the definition of correspondences provide a better understanding on conflicts. Furthermore, conflict analysis locates the areas of mismatch so that the user can concentrate on the right points. Thus, an experienced user would probably use the ODIS/OBF framework directly instead of interacting via a UML tool with the framework. It is cumbersome to further analyse compatibility directly from a UML tool as this means to explicitly select elements for analysis and correspondences to consider in analysis. Each iteration requires converting the UML model into the framework's representation, to perform the analysis and to deliver the results back to UML.

The conflict analysis framework should be customized for a particular application domain to provide the best possible conflict analysis. Specialized languages and additional rules, which cover the constraints of a domain provide a higher potential regarding conflict analysis of that domain. Furthermore, the framework includes at the same time the capability of model transformation. This can be used for automatic model abstraction, which is necessary to perform analysis as well as for model generation. In particular domains, it should be possible to generate parts of connectors, which bridge identified mismatches automatically.

A major problem of conflict analysis concerns UML tools and source code import. If a system needs to be analysed for which source code is available (which was the case in the examples), the source code first needs to be imported into a UML tool. The imported elements can then be used for analysis. However, the way the source code is imported into UML is not standardized and therefore tool dependent. For this work, we used Poseidon UML to import Java source code. The tool includes some stereotypes and notes in the integration process that are - in our opinion - tool dependent. These additional elements require special treatment in the RDF transformation. It is likely that another tool introduces other elements or uses different mapping rules.

Regarding connector generation, the identified conflicts are useful because they illuminate integration mismatches from a platform independent level. A direct comparison of the platform specific types of two components does not easily show which types are easy and which are more complicated to integrate. A platform independent view, however, shows the differences in the light of an independent type system. Furthermore, communication properties describe communication from a platform independent viewpoint. It is easier to identify conflicts from the provided taxonomies than from searching in the source code. Consequently, it is simpler to define connectors based on the platform independent conflicts. We will discuss connector generation further in the last part of the work.

The examples provided in this section do not give an exhaustive overview of the conflicts that can be identified by the framework. Further conflict categories concern substitutability of components and property checking of component configurations. Substitutability of components is similar to compatibility checking. However, other predicates are used which often impose further constraints in comparison to compatibility predicates. In particular, deciding substitutability demands equivalence checking to decide if a component can be replaced by another. Property

conflicts describe deadlock and progress properties of a system (component configuration) and are checked with LTSA.

Chapter 8

Parameterized Transformations

This chapter gives an example of parameterized model transformation. The text is mainly taken from a publication on the Middlware Conference [25], which was a joint work with my colleagues Susanne Busse, Andreas Billig, and Jörn Guy Süß.

8.1 Motivation

The OMG proposes the Model Driven Architecture (MDA) [98] to target fully automated component generation. Therefore, it distinguishes two kinds of models: platform independent models (PIM) and platform specific models (PSM). A PSM is normally described in a modeling language such as UML and corresponds in a one-to-one fashion to an implementation of the system. A platform independent models (PIM) can be defined without reference to a platform, and therefore without describing a particular form of technology.

A problem of many approaches for model transformation concerns the lack of customization of refinement transformations. A transformation should be applied according to specific user requirements. For example, a platform independent component should be transformed into an EntityBean or a SessionBean according to particular requirements. Unfortunately, existing software development tools (such as Rose, ArgoUML, etc.) do not support platform independent models. They often provide source code mappings only for one or a small number of technologies: These mappings are defined as one-to-one relationships between UML classes and source code classes. As a consequence, it is rarely possible to customize these mappings according to user requirements.

As a solution to the problem, we propose to realize PIM-to-PSM refinements with TRIPLE rules. The concentration of PIM-to-PSM refinements allows handling transformations aimed for several platforms such as Enterprise JavaBeans, CORBA, .Net, COM, etc. Furthermore, TRIPLE rules are customizable, which allows parameterization of PIM-to-PSM transformations. We parameterize the rules with features to describe the situation when the rule should be applied. Thus, a particular transformation is selected based on the requirements of a specific situation.

This chapter first introduces the requirements for parameterized model transformation (Section 8.2). Then, Section 8.3 describes the different parts of an example, consisting of a platform independent model, a feature model describing customization of EJB mappings, and a platform specific EJB model. Section 8.4 describes two customized transformations of the PIM into EJB PSMs. Section 8.5 discusses related work.

8.2 Parameterized Model Transformation

Different from common practice we do not map a PIM directly into source code. Instead, we map the PIM into a PSM expressed in UML. The appropriate mapping is chosen according to selected requirements. A PIM-PSM transformation is based on three elements:

- the platform independent model that should be mapped. In our context this will be a PIM represented as an instance of the framework's metamodel. In particular, elements like components are annotated with further properties that can be used to customize a model transformation to the specific situation.
- a feature model instance describing requirements that should be considered in the transformation. It is used to choosing an appropriate mapping. Thus, feature models allow the customization of a model transformation.
- mappings that define rules for possible transformations. Mappings formally specify design knowledge that is used when realizing a system with a specific middleware technology. They enable the automatization of the transformation process.

A PIM-PSM transformation consists of three steps: First, the developer designs a system independent from a platform technology. Second, he specifies his requirements for a specific PIM-to-PSM transformation by choosing features from the feature model. This selection results in a Feature Model Instance (FI), which does not contain any variation points. Third, the framework transforms the PIM according to the specified requirements.

Formally, the basis for a parameterized model transformation is a set of Triple transformation rules (see Section 3.1.5). Each rule takes the PIM and a Feature Model Instance model as input arguments and creates a PSM which corresponds to the given requirements. In our framework all participating models are defined as RDF models: A platform independent component model is defined in Section 5.1, a feature model is defined in Section 3.1.3, and a platform specific model to describe Enterprise JavaBeans is described in this section. On this basis, a mapping can be defined as a TRIPLE-mapping with parameterized contexts:

$$
\begin{array}{l}
\forall PIM, FI\ @pim2psmMapping(PIM, FI)\{ \\
\quad //Transformation\ rule1 \\
\quad \forall < \ldots\ necessaryVariables \ldots > \\
\qquad < \ldots\ elements \ldots > @PIM \wedge \\
\qquad < \ldots\ constraint \ldots > @FI \\
\quad \longrightarrow \\
\qquad < \ldots\ PSMelements \ldots > \\
\quad //Transformation\ rule2 \\
\quad \ldots
\end{array}
$$

8.3 Example

Our running example comes from federated information systems and was already described in Section 7.3. We describe two possible PIM-to-PSM transformations from the platform independent model to EJB specific models regarding specific requirements on distribution and optimization.

8.3.1 Platform Independent Model

Figure 8.1 shows the two components that are part of a mediator. Althought the example is described in Section 7.3, we repeat the description for an overview: If a client queries a mediator, the mediator first calculates which data sources are capable of answering the query

or part of it by using the Planner component. Then, it queries these sources, integrates the answers and delivers the result back to the client. The Planner calculates its plans based on specified interfaces of the data sources. These interface descriptions are called query capabilities (QC). A query capability consists of parameters that a data source can process as well as of result attributes returned by the data source. In Figure 8.1, the QCManager component stores the query capabilities (QC) of managed data sources. The Planner uses QC, obtained by the QCManager, to decide which data sources have to be queried. These query plans are provided to the execution component of a mediator.

Contrary to the former representation of the example, the figure also shows some annotations that are used for the transformation later on. These annotations are properties describing a component's role regarding its interoperation with another component. For example, the Planner is a client using the interface of the QCManager.

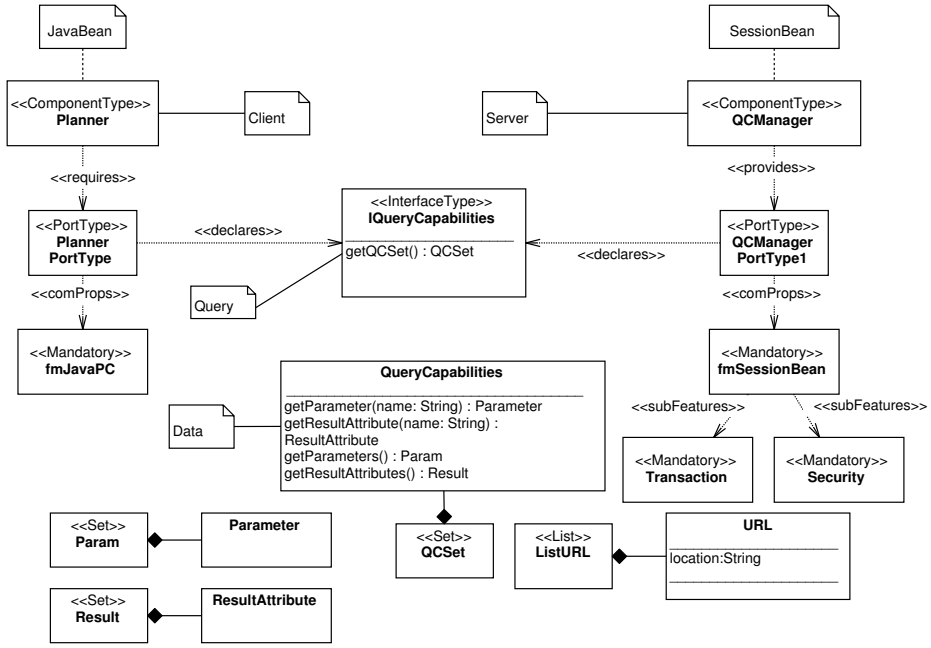

Figure 8.1: Example - PIM Component Type View

8.3.2 A Feature Model for Optimization of EJB Transformations

We use feature models to manage transformation variants. Figures 8.5 and 8.6 show instances of the feature model shown in Figure 8.2. The EJB feature model contains two points of variation: Components can either be co-located or distributed and communication can be further optimized for a minimal number of procedure calls or for an optimization of transmitted data.[1] A feature model instance (FI) represents the features chosen by the developer and is the input for the PIM-PSM transformation. For example, Figure 8.5 contains a feature model instance that describes a transformation in which both components are co-located and are optimized for direct data transmission.

The developer chooses the features from the feature model that should be considered in a specific transformation. We will examine a local PSM transformation optimized for the amount of transmitted data, as well as a distributed PSM transformation optimized for the amount of remote procedure calls. Both are realized with the EJB platform.

[1] In a local setting, data structures such as QueryCapabilities or Parameters can be accessed directly by the planner component, as there is no communication overhead as for a remote communication.

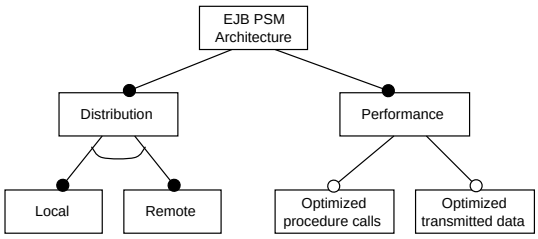

Figure 8.2: Feature diagram for EJB-based architectures

8.3.3 A Platform Specific Model for EJB

EJB components are described according to the EJB Profile for UML [58]. They mainly consist of a home interface, a remote interface and an implementation class. We use a slightly adapted form of the profile because it doesn't support EJB2.x local interfaces. Figure 8.3 shows a simplified specification of the EJB model.

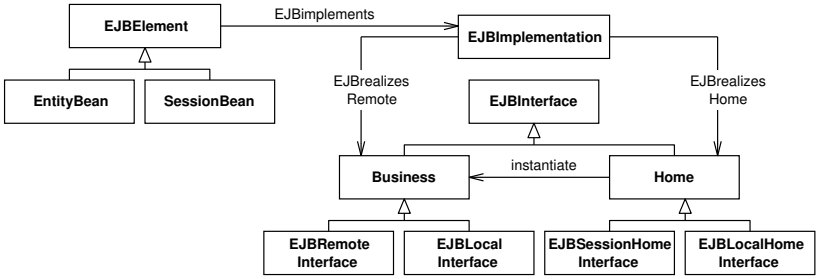

Figure 8.3: The EJB Component Model

It is important to notice that the EJB specification and the Profile do not model connectors. Thus, we have to translate connectors from the platform independent level into EJB Components. Figure 8.4 shows the corresponding PSM metamodel for EJB expressed in TRIPLE.

```
Class [
    typeOf -> { EJBElement, EJBImplementation, EJBEntityBean, SessionBean,
                Home, Business, EJBSessionHomeInterface, EJBLocalHomeInterface,
                EJBRemoteInterface, EJBLocalInterface } ].

EJBElement [
    subClass -> EntityBean;
    subClass -> SessionBean ].
EJBInterface [
    subClass -> Home [
       subClass -> EJBSessionHomeInterface;
       subClass -> EJBLocalHomeInterface ];
    subClass -> Business [
       subClass -> EJBRemoteInterface;
       subClass -> EJBLocalInterface ]
    ].

Property [ typeOf -> { EJBRealizesHome, EJBRealizesRemote, EJBImplements,
                       instantiate } ].

EJBRealizesHome   [ domain -> EJBImplementation; range -> Home ].
EJBRealizesRemote [ domain -> EJBImplementation; range -> Business ].
EJBimplements     [ domain -> EJBElement; range -> EJBImplementation ].
instantiate [ domain -> Home; range -> Business ].
```

Figure 8.4: Platform Specific Component Model for EJB expressed in TRIPLE

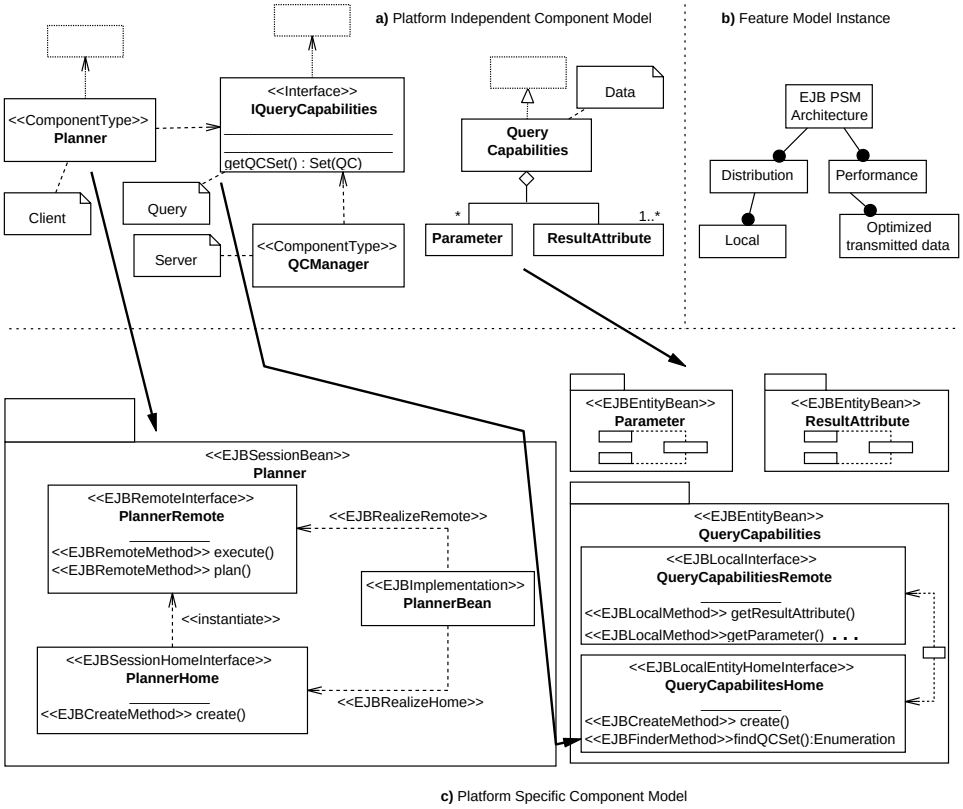

Figure 8.5: Example - PIM-PSM Transformation for a Local Configuration

8.4 PIM-PSM Transformation

Starting from the PIM of a mediator and the feature model for EJB architectures we show two possible transformations to the EJB PSM. In our example the transformation is based on patterns [12; 87] that were developed to optimize EJB communication and performance. We will discuss a local and a distributed configuration of the mediator components.

Local Configuration Scenario

This transformation is done according to the features that were chosen by the developer: both components are co-located and optimized for the amount of transmitted data. Using the patterns in [12; 87] the transformation results in the PSCM shown in Figure 8.5. The Planner component as a client in this example is mapped into a Session Bean because it is used as a business logic component, e.g. it provides computations. The QCManager, which includes the QC data structure (determined by the relationship to an element with a Data annotation), is mapped into three EntityBeans, as it presents persistent data. We don't need an extra QCManager Bean as this component would introduce another layer of indirection. Instead, we directly access the persistence layer. This leads to optimized data transfer, as we don't have to collect all data of the persistence layer and send it to the Planner component. Instead, data is returned in the form of a set of references to locally available QC EntityBeans. The operation to obtain all QC is renamed to *findQCSet* as described in the profile. If the Planner needs parameters, it issues calls to obtain the required information.

In order to save space, only the Planner component in Figure 8.5 shows all parts of an EJB according to the profile. Otherwise, we only show interesting parts of a bean and represent other elements as small boxes.

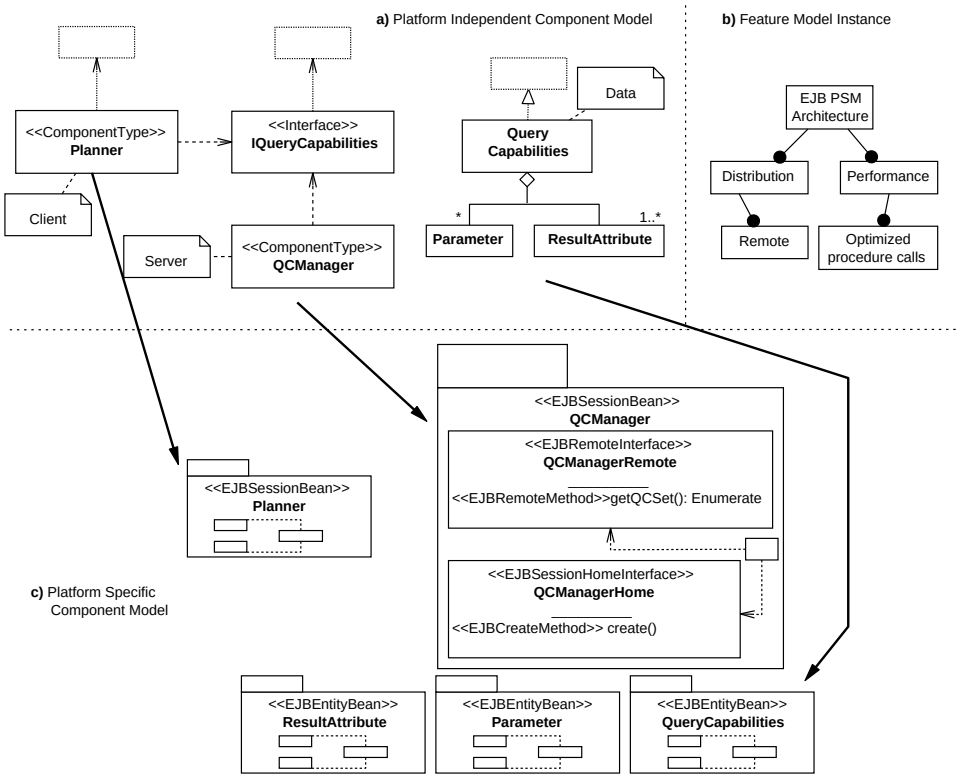

Figure 8.6: Example - PIM-PSM Transformation for a Distributed Configuration

Distributed Configuration Scenario

The distributed transformation optimizes remote procedure calls between distributed Planner components and a single QCManager component. Again, the transformation is based on the chosen features from the feature model. Regarding the EJB platform, several patterns for performance optimization were developed. We will use the Data Transfer Object pattern (DTO)[12; 87] and the Data Transfer Object Factory pattern (DTOF)[87] for the PIM-to-PSM transformation in this example.

Figure 8.6 shows the mapping result. The QCManager component is mapped into a stateless SessionBean following the DTOF pattern that provides a facade to the persistence layer consisting of three EntityBeans. The QCManager locally assembles Data Transfer Objects for each query by calling the Entity Beans. Different from the local mapping these objects are copies of persistent data. Thus, a query of the Planner component results in a single remote procedure call.

Parameterized Transformations with TRIPLE

The specification of the mappings for our transformation[2] described before consists of two parts: the first one defines the general mapping from PIM elements to EJB, the second one defines the mapping depending on the possible features.

Figure 8.7 shows part of the general mapping definition from PIM models to SessionBeans or EntityBeans. The arguments of the bean mapping are the resource variables *X*, *PIM*, and

[2]The following rules, shown in figures 8.7 and 8.8, precisely cover the available choices of the feature model shown in Figure 8.2.

Kind. *PIM* is the context of the mapping source, X is the element from that source which shall be mapped, and *Kind* states whether a SessionBean or an EntityBean shall be the result. The mapping definition consists of one rule which expresses the following: If an element with an appropriate *provides* InterfaceType can be derived within the source context (lower part of the rule) then a Bean and the remote/home interfaces corresponding to the EJB PSM will be generated (upper part (head) of the rule). The instantiation of the *type*-variables (B, R, H) in the target structure depends on the *Kind* parameter, which determines whether an EntityBean or an SessionBean should be generated. The mapping also copies the operations of the 'provide' interfaces into the remote interface of the Bean.

```
 ... // namespaces and abbreviations
// This is a transformation between the platform independent
// model and the EJB model.
// General mapping to create a bean.
forall X,PIM,Kind  @ beanMapping(X,PIM,Kind) {
  forall B,R,H,Y,TF,P,ZBEAN,ZREMOTE,ZHOME,ns,ns2,PT,N,NHOME,O
    ns:X[sys:directType -> B;            // create an EJB of type B
         ejb:name->N;
         ejb:EJBimplements-> ns:ZBEAN[
             sys:directType->ejb:EJBImplementation;
             ejb:EJBRealizesRemote->ns:ZREMOTE[
                 sys:directType->R;
                 ejb:operations->O;     // copy operations in the
                 ejb:name->P];          // remote interface
             ejb:EJBRealizesHome->ns:ZHOME [
                 sys:directType->H;
                 ejb:name->NHOME;
                 ejb:instantiate->ns:ZREMOTE
    ] ] ] <-
    (ns:X[sys:directType->core:ComponentType;
          core:name->N;
          core:provides->PT[sys:directType->core:PortType;
                core:declares->ns2:Y[core:name->P;
                   core:operations->O]]]@@PIM and
      cond_(Kind='Entity',(            // variables of an 'EntityBean'
         B=ejb:EntityBean,
         R=ejb:EJBLocalInterface,
         H=ejb:EJBLocalHomeInterface),true
      ) and
      cond_(Kind='Session',(           // variables of an 'SessionBean'
         B=ejb:SessionBean,
         R=ejb:EJBRemoteInterface,
         H=ejb:EJBSessionHomeInterface),true
      ) and
      concat(ZBEAN,'Bean',X) and      // create literals
      concat(ZREMOTE,'Remote',Y) and
      concat(ZHOME,'Home',X),
      concat(NHOME,N,'Home')
    ).
}
```

Figure 8.7: Example - A General Bean Mapping Rule

Figure 8.8 shows the specific PIM-to-PSM-mapping depending on possible feature model instances. Within the utility mapping *util* two specific FIs are considered: the distributed and the local configuration variant. The mapping *pim2psmMapping* has two parameters: the context of the PIM source *PIM* and the context of the feature instance *FI*. The body of the mapping contains the mapping rules according to the variants of transformation explained above:

- Any client element is mapped to a session bean.
- A server will become a session bean if the variant of remote distribution is chosen.
- All aggregated elements of a data element are mapped to entity beans.
- A data element x also become an entity bean. If the variant of local data is chosen then an EJB conform conversion of any *Query*-annotated operation which uses the interface of x is placed within the home interface of the generated bean.

8.5 Related Work

Several proposals for model transformation have been recently published in response to the OMG's RFP. These proposals can be classified regarding several categories such as how they define transformation rules or rule application strategies. Czarnecki and Helsen [42] provide a classification of model transformation approaches. According to this classification, our approach, which is based on TRIPLE, is a declarative relational model-to-model approach. Other model transformation languages are based directly on UML. [133] for example defines an extension of the Object Constraint Language OCL using database manipulation operations of SQL. We use an existing language - TRIPLE - to define mappings. Thus, the model transformation is 'automatically' done by the inference engine. It allows declaring transformations in a very flexible and compact syntax, similar to F-Logic.[3] Additionally, the TRIPLE concept of parameterized contexts allows a modularization of rule sets and enables the reuse of mappings by parameterized mapping specifications.

Additionally, our approach uses feature model instances, which describe mapping variants to parameterize mappings. Feature models are important in the context of product line engineering and domain analysis ([39; 41]). They are used to describe variants within a system family and to generate applications as instances of this system family from the application's specification.

Mostly, the generative approach is used on the implementation level. [15] defines the KobrA methodology for a component-based engineering with UML very similar to our approach. KobrA also contains the specification of variable parts of a system and feature models called decision models. But these concepts are only discussed in the context of product line engineering. We use them to support the general development process wherein alternative realizations must be chosen according to the requirements. Additionally, [15] discusses no explicit specification of relationships between decisions and realizing system variants so that the transformation has to be done manually.

[3] As described in [57] a great advantage is the ability to express the model and instance level in a uniform way and to define multiple targets in a single rule.

```
 ... // namespaces and abbreviations
// helper predicates for feature model instance
forall FI @ util(FI) {
  forall V,X,Y,Z remoteCall <-
    X[sys:directType->core:MandatoryFeature;
      core:name->'Distribution'; core:subFeatures->Y[
        sys:directType->core:MandatoryFeature;
        core:name->'Remote']]@@FI and
    V[sys:directType->core:MandatoryFeature;
      core:name->'Performance';  core:subFeatures->Z[
        sys:directType->core:MandatoryFeature;
        core:name->'OptimizedProcedureCall']]@@FI.

  forall V,X,Y,Z localCall <-
    X[sys:directType->core:MandatoryFeature;
      core:name->'Distribution'; core:subFeatures->Y[
        sys:directType->core:MandatoryFeature;
        core:name->'Local']]@@FI and
    V[sys:directType->core:MandatoryFeature;
      core:name->'Performance'; core:subFeatures->Z[
        sys:directType->core:MandatoryFeature;
        core:name->'OptimizedTransmittedData']]@@FI.
}

forall fi isRemoteCall(fi) <- remoteCall@util(fi).
forall fi isLocalCall(fi) <- localCall@util(fi).

forall PIM,FI @ pim2psmMapping(PIM,FI) {

  // Map Operations
  forall S,P,O
    S[sys:directType->ejb:Operation;ejb:name->?name]
  <-
    S[sys:directType->core:Operation;core:name->?name]@@PIM.
  // Map other types
  ...

  // Create Session Beans for ComponentTypes annotated with 'Client'
  forall X,S,P,O,n,PRO
      S[P->O]
    <-
      n:X[sys:directType->core:ComponentType;
        core:properties->PRO[core:name->'Client']]@@PIM and
      S[P->O]@beanMapping(X,PIM,'Session').

  // Create Session Beans for ComponentTypes
  // that are annotated as 'Server' if the
  // feature instance describes a remote scenario
  forall X,S,P,O,n,PRO
      S[P->O]
    <-
      n:X[sys:directType->core:ComponentType;
        core:properties->PRO[core:name->'Server']]@@PIM and
      S[P->O]@beanMapping(X,PIM,'Session') and
      isRemoteCall(FI).

  // Create EntityBeans for Instances and
  // their components that are market as Data.
  forall S,P,O,X,Y,L,n,m,PRO
     S[P->O]
   <-
     m:X[core:properties->PRO[core:name->'Data'];core:elements->L[core:hasType->n:Y]]@@PIM and
     S[P->O]@beanMappingRc(Y,PIM,'Entity').

 forall S,P,O,X,Y,L,n,m,PRO
     S[P->O]
   <-
     m:X[core:properties->PRO[core:name->'Data'];core:elements->L[core:hasType->n:Y]]@@PIM and
     S[P->O]@beanMappingRc(X,PIM,'Entity').

  // Create Operations as part of the Home interface
  // of EntityBeans marked as 'Data', if the
  // operation is annotated with 'Query' in a local  scenario.
   forall S,P,O,X,Y,L,n,m,PRO1,PRO2
       S[P->O]
    <-
       m:Y[core:properties->PRO1[core:name->'Query']]@@PIM and
       n:X[core:properties->PRO2[core:name->'Data']]@@PIM and
       isLocalCall(FI) and S[P->O]@beanMappingQuery(X,PIM).
  ...
}
```

Figure 8.8: Example - Specific PIM-PSM Transformation Rules

Part IV

Conclusion and Outlook

Chapter 9

Conclusion

This thesis presented a Framework for *Component Conflict Analysis* and *Composition* that serves as a basis for component integration. The framework fulfils the required objectives for component integration and is placed in a technological context that provides several advantages compared to other approaches: It allows analysing components with type, behavior and communication properties, it is adaptable for application domains and for additional kinds of analysis and it integrates model transformation capabilities. In this section, we will summarize the main objectives for component integration, the architecture of the framework and the resulting advantages. Finally, we give an outlook for future research.

9.1 Objectives for Component Integration

In the work we divided component integration into two steps: Conflict Analysis and Connector Generation. The work primarily aimed at the first step and focused on relationships to decide compatibility and substitutability of components.

The work focused on the integration of small to medium-sized components that originate in object-oriented middleware technologies such as CORBA, COM, .NET, Jini, etc. As component specifications of these technologies differ in form, complexity and content, a canonical representation was required that represented the least common denominator of components in these technologies. The presented framework supports such a platform independent representation by defining several models that represent the core concepts of components: structure, behavior and communication properties.

As components are normally specified in terms of concrete technologies, they cannot be directly compared unless they are translated into the platform independent component models provided by the framework. The framework principally supports model abstractions, which demerge platform specific details from component descriptions and create a canonical representation of those components. In this thesis, however, we have concentrated the more complex refinement transformation that transforms platform independent models into platform specific models based on parameters which modify the results according to user requirements.

Based on the platform independent component model, we provided conflict analysis. We aimed to check for as many conflict categories as possible, because the more conflicts the analysis identifies, the more accurate the cost assessment of an integration effort will be. We proposed two complex relationships for conflict analysis: A compatibility relationship checks if two components can interact, whereas a substitutability relationship decides if two components can be exchanged. Both relationships are composed of several other relationships that cover the specification categories of components. These include subtype, simulation and bisimulation relationships to cover essential conflict categories. It further introduced a new relationship that evaluates the compliance of communication requirements based on the exposed features of two components.

We proposed two taxonomies to describe communication requirements as well as technology-related metadata. The first taxonomy is a modified version of the connector taxonomy proposed by Medvidovic/Mehta. We used this taxonomy, because it provides the most fine grained properties available today. We modified the original taxonomy to describe communication in currently available middleware systems and to express customizable properties of those communication mechanisms. The second taxonomy covers technology-related properties.

A Property-based conflict analysis of components' communication requirements is often underestimated at present. Only a few approaches cover property-based description and analysis. Examples are the connector taxonomy of Mehta/Medvidovic or architectural styles proposed by Allan/Shaw. However, a property-based analysis provides several advantages: Properties can be used to transform conceptual models into specific representation covering particular communication requirements and they allow gaining more information regarding incompletely specified components.

Components are rarely fully specified in current technologies. Some facts are stated implicitly or not at all. For example, a major problem of present middleware technologies is that they ignore the need for a behavioral specification. Most popular technologies cannot associate this kind of information with their component definitions. Furthermore, communication requirements are often not explicitly stated in middleware technologies. This is sufficient as in the context of one technology communication must be based on predefined mechanisms for which the technology transparently takes care of. However, when dealing with more than one technology, with different communication paradigms, or with adaptable communication paradigms, it is likely that the requirements of components regarding communication will vary. Consequently, explicit knowledge about communication properties is required.

We further aimed to support model refinement. Model refinement covers component transformation from a platform independent level to a platform specific level. It is required to support the second step of integration: connector generation. We proposed to use a property-based transformation that is based on feature models to describe user requirements.

A parameterized transformation paves the way for design reuse. It becomes possible to reuse a conceptual model for different technologies. One can create a EJB model or a CORBA model or a model for another language from a single conceptual model. Regarding communication properties, different kinds of communication can be generated for a conceptual model. For example, one can create an EJB component accepting synchronous communication (e.g. a SessionBean) or a component accepting asynchronous communication (e.g. a MessageBean) from a single conceptual component. Based on properties in a feature model, model transformation can also be parameterized in a more flexible way. In [25], we demonstrated that model transformations can be parameterized based on J2EE patterns.

As a last objective, we aimed to support a seamless integration with design tools used in software development. As UML is the de facto modeling language, we support conflict analysis from within UML tools.

9.2 Architecture of the Framework

The objective of this thesis concerns component conflict analysis in the context of a UML-based software development process. As no technology is able to cover all objectives by itself, we propose a technology combination to satisfy the objectives: We use UML as a front-end modeling language for the end-user. We use concepts of Architecture Description Languages to compose the platform independent component models on which conflict analysis is based. As a formalization of those models, we chose RDF, as it is a standardized language that can be easily used in several application domains. To cope with models on different abstraction levels, we orientated on the Model-Driven Architecture of the OMG and implemented model transformation on TRIPLE. TRIPLE, which is a 'RDF-aware' extension of F-Logic provides reasoning support and at the same time allows describing parameterized mappings.

The architecture of the framework was divided into five parts: The first part consists of a UML modeling tool that is able to import and export XMI. The second part transforms models between XMI and RDF. The third part, the Ontology-Based Framework, realizes component analysis and transformations between platform independent and platform specific models. The fourth part consists of several external tools that provide conflict analysis capabilities such as type checking and model checking. The framework is augmented with a component repository, which stores all kinds of component specification regardless of whether they are formalized as XMI, RDF, artifacts or represent PIMs or PSMs.

Technically the framework is based on the ODIS repository, which itself is based on a deductive knowledge base (XSB). The models and their associated rules that are used within the framework are realized based on TRIPLE. Type and behavioral conflicts are checked with external tools such as Haskell, LTSA, FDR, fc2tools, or Aldebaran. We used these tools because they provide more sophisticated analysis than was possible with a deductive knowledge base.

9.3 Summary of Contributions

The framework explicitly aims at conflict analysis in several areas. It supports type and behavior checks and additionally communication requirements of components. Consequently, a far better overview concerning the interoperability of components can be achieved than by using analysis tools that deal with only one or two issues. As a result, an estimation of integration costs becomes more accurate and simpler to calculate. Furthermore, a user can customize the framework with additional analysis methods or with different specification languages.

At the same time, the framework represents a 'minimal' model-driven development system. It supports abstraction and refinement operations on models and handles models on different abstraction levels. As mentioned above the refinement of models can be parameterized and therefore adapted to satisfy user requirements. This is contrary to one-to-one transformations that are usually provided by development tools.

The framework further supports a UML-based model driven development and integrates reasoning and transformation capabilities to 'existing' modeling tools. The framework uses RDF to represent component specifications. UML models can be transformed into this internal representation (RDF) and vice versa. Consequently, we are able to reason on UML models, which is not directly supported by UML, and further to attach arbitrary background information on UML elements and to interpret the background information as RDF statements.

9.4 Connector Generation

The framework provides the foundation for the second step of integration: connector generation. Connector generation depends on the results of the analysis process. We propose to start connector generation on the platform independent level and then to refine the connector for particular technologies. This process is briefly described in the following steps. The first two of these steps require user interaction. The fourth step can be customized for certain requirements.

(1) For each structural conflict a correspondence relationship needs to be specified manually if possible. If correspondences cannot bridge identified incompatibilities in a reasonable way, connector generation stops at this or at the next step. We distinguish between mismatched operation and parameter names and mismatched types. A correspondence is a relationship between operations of the client's required interface and the server's provided interface. In case of a type mismatch an additional conversion function needs to be defined manually (and implemented).

(2) For behavioral conflicts (protocol conflicts), a simulation conflict results if a server cannot react on the issued operations of a client. A common cause relates to mismatched

operation granularity. For example, the client issues an operation co_1, but the server requires an explicit initialization so_1 before the request of the client can be handled by another operation so_2. In this case the connector needs to implement the correspondence $co_1 \sim (so_1 \rightarrow so_2)$.

(3) Based on this information a platform independent connector can be automatically specified. We propose to use two kinds of diagrams to define a connector: a structural diagram and a state diagram. The structural diagram consists of the internal operation needed to map diverging types as well as of the required operations of the client. The behavioral diagram specifies the mediation of the required operation orders of both components. We can derive information for a connector's behavior by utilizing the algorithm proposed by Inverardi [65]. She defines an algorithm that handles behavioral mismatches between several components by defining a connector that resolves these mismatches. The resulting state diagram consists of transitions where the events correspond to the operations issued by the client and the actions are either internal type transformations or called operations of the server component.

(4) Based on the platform independent description, we transform the connector into platform specific representations. The specified communication properties are important in this step as they determine the structure and additional behavior of the resulting platform specific connector. For example, if the property description requires a distributed connector and a name server lookup, the connector can be mapped according to the proxy pattern with an additional operation call to a name server.

9.5 Future Work

We plan to augment the framework with additional analysis capabilities and to enhance the concept of parameterized transformation.

At present, matching algorithms for web services are developed that use 'semantic' matching to improve the identification of suitable services. These algorithms are used to search for components that are the most suitable compared to a given query. The 'semantic' matching is often based on ontologies that are used for calculations. At present, a semantic matching service is developed for the ADAM repository. This enables framework users to query for suitable components that are stored in the ADAM repository based on metadata attached to the components. The prototype of the searching algorithm uses a taxonomy, which describes the technological requirements of a component. Further, the functionality of a component is described. These metadata can be used for a ranking of suitable components. An advantage of the framework compared to solutions for web services concerns the additional checking of types, behavior and communication properties that can be calculated at the same time. Furthermore, the framework supports different technologies, from which the web service technology is only one possible incarnation. Consequently, we can search the database for components origin in different technologies, which are specified on different abstraction levels.

We further plan to enhance parameterized transformation. We first exploit the defined communication taxonomies for parameterized transformation. The customizable properties of a communication mechanisms can naturally be used to describe mapping variations of platform independent component descriptions for a particular technology. At the moment, parameterized transformations for the Enterprise JavaBeans technology, which are based on the customizable properties of the communication mechanisms defined for that technology, are being researched.

At the same time, we plan to provide mappings for J2EE patterns defined in the EJB community. The patterns represent optimized solutions for particular use cases. We exploit the existing patterns in transformations so that a platform independent model can be optimized for a particular use case. Contrary to the approach we already proposed at the Middleware 2004 conference, we encode the patterns and the available variations of the patterns as feature models and do not create a feature model, which describes the applicability and the the consequences

for a number of patterns. This solution, which was presented at the middleware conference, is more complicated to extend than the new concept. However, we encode information regarding the applicability and the consequences as feature attributes of the feature models describing the patterns. This allows for an efficient search for suitable patterns.

Part V

Appendix

Appendix A

Acronyms

Acronyms

ADAM Analytical-Data-on-Artifacts-and-Models

ADL Architecture Description Language

CBSE Component Based Software Engineering

COTS Commercials Off-the-Shelf

DNS Domain Name Service

EJB Enterprise JavaBean

EVE Evolution and Validation Environment

FI Feature Model Instance

FM Feature Model

IC Integrated Circuit

LIS Legacy Information System

MDA Model Driven Architecture

MDD Model-Driven Development

MOF Meta-Object Facility

OBF Ontology-Based Framework for component analysis and transformation

ODIS Ontology-Based Domain Repository

OLAP Online Analytical Processing

OMG Object Management Group

PIM Platform Independent Model

PICM Platform Independent Component Model

PSM Platform Specific Model

RDF Resource Description Framework

RDFS RDF Schema

RM-ODP Reference Model of Open Distributed Processing

UI User Interface

UML The Unified Modeling Language

URI Uniform Resource Identificator

W3C World Wide Web Consortium

XMI XML Metadata Interchange

Appendix B

UML Profile for the Ontology-Based Framework

This section defines a UML representation of the framework's models. The representation is formalized as a profile that basically provides an one-to-one mapping between the framework's elements and UML model elements. Most of the framework models are represented based on UML class diagrams, where the standard elements are annotated by stereotypes. The only exception is the framework's definition of state diagrams which is based on UML elements defined for use within state diagrams.

The profile definition orientates on UML 1.4.2[1]. However, it should be valid within most UML 1.x versions. The profile does not define a stereotype hierarchy. It differs from the OMG recommendation as it provides no well-formedness rules expressed in OCL as we perceive the profile as a representation of the framework's models, we check validity by integrity rules defined on the framework's models.

In this section, we first give an overview of the diagram types together with a listing of the defined stereotypes and then provide a description of each stereotype.

B.1 Stereotype Summary

The following profile defines four diagram types: a component type diagram, a run-time configuration diagram, a behavior diagram, and a property diagram:

Component Type Diagram

[1] formal/04-07-02

Name	BaseClass	Stereotype
ComponentType	Class	≪ComponentType≫
ConnectorType	Class	≪ConnectorType≫
InterfaceType	Interface	≪InterfaceType≫
PortType	Class	≪PortType≫
Record	Class	≪Record≫
List	Class	≪List≫
requires	Dependency	≪requires≫
provides	Dependency	≪provides≫
supertype	Dependency	≪supertype≫
extends	Dependency	≪extends≫
declares	Dependency	≪declare≫
refines	Dependency	≪refines≫
comProps	Dependency	≪comProps≫
techProps	Dependency	≪techProps≫

Configuration Diagram

A configuration diagram represents system configurations.

Name	BaseClass	Stereotype
Connector	Class	≪Connector≫
Component	Class	≪Component≫
Interface	Interface	≪Interface≫
Port	Class	≪Port≫
binds	Association	≪binds≫
requiredPort	Dependency	≪requiredPort≫
providedPort	Dependency	≪providedPort≫
represents	Dependency	≪represents≫

Behavior State Diagram

Name	BaseClass	Stereotype
State	State	≪State≫
Action	Transition	≪Action≫

Property Diagram

The property diagram defined a feature model for annotation of communication properties.

Name	BaseClass	Stereotype
MandatoryFeature	Class	≪Mandatory≫
OptionalFeature	Class	≪Optional≫
Xor	Class	≪Xor≫
Or	Class	≪Or≫
FeatureAttribute	Class	≪FeatureAttribute≫
subFeatures	Dependency	≪subFeatures≫
feAttributes	Dependency	≪feAttributes≫
excludes	Dependency	≪f_excludes≫
requires	Dependency	≪f_requires≫

These diagrams are placed within a particular model and a top level package:

Name	BaseClass	Stereotype
OBFStructuralModel	Model	≪OBF_SModel≫
OBFStructuralPackage	Package	≪OBF_SPackage≫

B.2 Elements of the Component Type Diagram

In the following, we provide descriptions for each stereotype. We do not specify well-formedness constraints as mentioned above.

Element: ComponentType

Stereotype	Base Class	Description
≪ComponentType≫	Class	This element represents a component type. Its type is calculated by aggregating the associated requires & provides interface types, which are indirectly referred to via port types. The component body is not represented by this element. It is not allowed to place any UML features in this element. Features are described by interface types associated with the component type.

The notation used for a component specification is a class stereotyped as ≪ComponentType≫. Only one compartment of a class is needed as it is not allowed to add any feature (operations or attributes).

Tagged Values

The following tagged values can be associated with a component type.

Tagged Value	Description
ProtocolExpression	A protocol expression describes the behavior of a component type. This expression defines the behavior of the whole component and is not used for conflict analysis. A protocol expression can also be defined in a comment in the following form: PAE(Language)=<expression>.
Language	Language of the protocol expression. It is required if a protocol expression is defined.
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings. The tagged value corresponds to the property name defined in the core models. If also the property value as defined in the core model 'property' element needs to be specified or if several properties need to be associated with an stereotyped UML element, the properties are to be specified as comments in the form: property(name)=value.

Element: ConnectorType

Stereotype	Base Class	Description
≪ConnectorType≫	Class	The type of a connector depends on the associated interface types. It is not allowed to place any UML feature in this element. Features are described by interface types associated with the component type.

The notation used for a connector specification is a class stereotyped as ≪ConnectorType≫.

Tagged Values

The following tagged values can be associated with a component type.

Tagged Value	Description
ProtocolExpression	A protocol expression describes the behavior of a component type in terms of a process algebra expression. The expression is no used for conflict analysis.
Language	Language of the protocol expression. It is required if a protocol expression is specified.
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Element: InterfaceType

Stereotype	Base Class	Description
≪InterfaceType≫	Interface	An interface type describes component interactions in terms of operations. It consists of a set of operations. Attributes have to be encapsulated as operations.

The notation used for an interface type is a UML interface stereotyped as ≪InterfaceType≫. Alternatively, a UML class with the same stereotype can be used. Both elements are mapped into the framework's interface types.

Tagged Values

The following tagged value can be associated with an interface type.

Tagged Value	Description
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Element: PortType

Stereotype	Base Class	Description
≪PortType≫	Class	A port type defines a point of interaction. The interaction is expressed by an interface type and a protocol expression. Port types are referenced by component types and connector types to declare required and provided interface types. It is illegal to place any feature in this element.

The notation used for a port type is a UML class stereotyped as ≪PortType≫.

Tagged Values

The following tagged values can be associated with a component type.

Tagged Value	Description
ProtocolExpression	A protocol expression describes the behavior of a component type in terms of a process algebra expression.
Language	Language of the protocol expression.
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Element: Record

Stereotype	Base Class	Description
≪Record≫	Class	A record refers to a normal UML class. Each class that has no stereotype or is stereotyped with 'Record' is translated into a Record in the framework.

The notation used for a port type is a UML class stereotyped as ≪Record≫.

Tagged Values

The following tagged values can be associated with a record type.

Tagged Value	Description
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Element: List

Stereotype	Base Class	Description
≪List≫	Class	A list type represents any kind of array or collection.

The notation used for a port type is a UML class stereotyped as ≪List≫.

Tagged Values

The following tagged values can be associated with a list type.

Tagged Value	Description
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Relationship: Requires

Stereotype	Base Class	Description
≪requires≫	Dependency	A requires relationship defines services of a component type or a connector type that are needed by this element. The services are expressed via a port type.

The notation used for a requires relationship is a dependency stereotyped as ≪requires≫.

Relationship: Provides

Stereotype	Base Class	Description
≪provides≫	Dependency	A provides relation identifies the services that are offered by a component type or mediated by a connector type. This functionality is expressed via a port type.

The notation used for a provides relationship is a dependency stereotyped as ≪provides≫.

Relationship: Supertype

Stereotype	Base Class	Description
≪supertype≫	Generalization	The supertype relationship describes single inheritance between component types or connector types. It restricts the UML generalization to component and connector types. The generalization hierarchy has to be a tree.

Several modeling tools do not allow specifying a stereotype for a generalization. Therefore, we use a *dependency* to visualize this relationship. Thus, the notation used for a supertype relationship is a dependency stereotyped as ≪supertype≫.

Relationship: Extends

Stereotype	Base Class	Description
≪extends≫	Generalization	The extends relationship describes multiple inheritance between interfaces. It restricts the UML generalization to interface types. The generalization hierarchy can be a graph.

We use a *dependency* to visualize this relationship. Thus, the notation used for a extends relationship is a dependency stereotyped as ≪extends≫.

Relationship: Declares

Stereotype	Base Class	Description
≪declares≫	Dependency	This relationship links an interface type to a port type.

The notation used for a declares relationship is a dependency stereotyped as ≪declares≫.

Relationship: Refines

Stereotype	Base Class	Description
≪refines≫	Generalization	The refines relationship describes single inheritance between port types. The generalization hierarchy has to be a tree. The refines relationship is mainly used to inherit the properties associated with a property type.

We use a *dependency* to visualize this relationship. Thus, the notation used for a refines relationship is a dependency stereotyped as ≪refines≫.

Relationship: ComProps

Stereotype	Base Class	Description
≪comProps≫	Dependency	This relationship links a single feature to a port type. The feature should correspond to a feature that is predefined in a communication taxonomy or in a technology taxonomy. If the port type extends another port type which is also associated with features, the compProps relationship overrides previously defined and equally named features. The property can be associated with component types, connector types as well as with port types.

The notation used for a comProps relationship is a dependency stereotyped as ≪comProps≫.

Relationship: TechProps

Stereotype	Base Class	Description
≪comProps≫	Dependency	This relationship corresponds to the 'comProps' relationship. Instead of targeting on communication properties it describes technology-related properties.

The notation used for a techProps relationship is a dependency stereotyped as ≪techProps≫.

Type Mappings

Type conversion between UML and the framework depends on the actual type system installed. As mentioned above the framework only provides a basic type system for conflict identification. This type system should be customized for the environment at hand. The mapping of the provided type system is one-to-one:

- Primitive UML types are mapped to distinct basic types defined in the framework.
- UML operations are mapped to operations as defined by the framework, including arguments, exceptions and return types.
- UML interfaces are mapped directly to interfaces of the framework.
- UML classes are mapped into record types.
- For other complex UML types a corresponding element needs to be defined in the framework.

B.3 Elements of the Configuration Diagram

This diagram shows the configuration of components and connectors. It is not used for conflict identification, only for the representation of systems.

Element: Component

Stereotype	Base Class	Description
≪Component≫	Class	This element describes a component instance in a runtime configuration. This instance has to be of a predefined type. It neither can redefine the interfaces of the corresponding type nor can it change existing properties or constraints.

Tagged Values

The following tagged value can be associated with a component type.

Tagged Value	Description
Type	The associated type of this instance as defined in a type model.
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Element: Connector

Stereotype	Base Class	Description
≪Connector≫	Class	This element describes connector instances in a runtime configuration. The instances are always connected with two components. Additionally, any instance is related to a connector type.

The notation used for a connector instance is a class stereotyped as ≪Connector≫.

Tagged Values

The following tagged value can be associated with a connector type.

Tagged Value	Description
Type	The associated type of this instance as defined in a type model.
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Element: Interface

Stereotype	Base Class	Description
≪Interface≫	Class	An interface describes the services required or provided by a component or mediated by a connector. The services are attached to a port, which defines an interaction point. An interface inherits every operation from it associated type. It is not allowed to change the signature of this type.

The notation used for an interface instance is a class stereotyped as ≪Interface≫. It can also be defined as a UML class stereotyped as an interface.

Tagged Values

The following tagged value can be associated with an interface type.

Tagged Value	Description
Type	The associated type of this instance as defined in a type model.
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Element: Port

Stereotype	Base Class	Description
≪Port≫	Class	A port binds the interfaces of a component. It specifies the connection end points of a component. For outgoing connections the port must be bound by a 'requiredPort' dependency to a component. An incoming connection is represented by a 'providedPort' dependency.

The notation used for a port is a class stereotyped as ≪Port≫.

Tagged Values

The following tagged value can be associated with a port type.

Tagged Value	Description
Type	The associated type of this instance as defined in a type model.
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Relationship: Binds

Stereotype	Base Class	Description
≪binds≫	Dependency	The binds relationship defines a connection between components via a connector. It links a connector with a a port of a component. The connector needs to provide a compatible port to that of the component.

The notation used for the bind relationship is a dependency stereotyped as ≪binds≫.

Relationship: RequiredPort

Stereotype	Base Class	Description
≪requiredPort≫	Dependency	The relationship binds a port to a component or a connector. It corresponds to the 'requires' relationship of the type model. It therefore describes the link between components (connectors) and their required interaction points. Interaction points are expressed by a port and an interface.

The notation used for the relationship is a dependency stereotyped as ≪requiredPort≫.

Relationship: ProvidedPort

Stereotype	Base Class	Description
≪providedPort≫	Dependency	The relationship corresponds to the 'provides' relationship of the type model. The semantics of the relationship is analogue to that of the 'requiredPort' relationship described above.

The notation used for the relationship is a dependency stereotyped as ≪requiredPort≫.

Relationship: Represents

Stereotype	Base Class	Description
≪represents≫	Dependency	The relationship corresponds to the 'declares' relationship of the type model. It associates a port to exactly one interface.

The notation used for the relationship is a dependency stereotyped as ≪represents≫.

B.4 Elements of the Behavior State Diagram

A state diagram can be used to describe the behavior of components in form of labeled transition systems (LTS). Conflict analysis is based on the comparison of two LTS.

Element: State

Stereotype	Base Class	Description
≪State≫	State	This element represents a state as defined in UML. As UML already defines state machines a normal state diagram as defined by the UML can be used to describe a behavioral model of a component. A state does not contain any features.

The notation used for a state is a UML state stereotyped as ≪State≫.

Tagged Values

The following tagged value can be associated with a component type.

Tagged Value	Description
property	This tag associates a component type with context information. The information is used for example to describe refinement mappings.

Relationship: Action

Stereotype	Base Class	Description
≪Action≫	Transition	This element represents a transition between two states. As UML defines this transition between states, the 'to' and 'from' properties described by the behavioral model are expressed by the UML 'target' and 'source' associations defined between transition and StateVertex.

The notation used for a transition is a UML transition stereotyped as ≪State≫.

Tagged Values

The following tagged value can be associated with a transition.

Tagged Value	Description
refersTo	Each transition has to be annotated with a tagged value, describing the operation that is responsible for that transition. If it is not possible in a modeling tool to attach a tagged value to a 'UML association' the 'refersTo' relationship can also be expressed by defining a UML CallAction for each transition. The CallAction has to be linked to the corresponding operation.

B.5 Elements of the Property Diagram

The property model describes hierarchies of properties. As explained in Section 5.1.4, we perceive properties as features, which we use to create feature models. The following definitions can be used to describe feature models and feature instances in UML[2].

Element: MandatoryFeature

Stereotype	Base Class	Description
≪Mandatory≫	Class	This element represents a mandatory feature in a feature model.

The notation used for a mandatory feature is a UML class stereotyped as ≪Mandatory≫.

Element: OptionalFeature

Stereotype	Base Class	Description
≪Optional≫	Class	This element represents an optional feature in a feature model.

The notation used for an optional feature is a UML class stereotyped as ≪Optional≫.

Element: Xor

Stereotype	Base Class	Description
≪Xor≫	Class	Variability elements are either be alternatives or options. They are represented by filled or blank arcs in a feature diagram. We explicitly model variability elements as stereotyped UML classes. The 'xor' represents an alternative (filled arc).

The notation used for a 'xor' variability element is a UML class stereotyped as ≪Xor≫.

[2]Other approaches to represent feature models in UML are proposed for example by [37].

Element: Or

Stereotype	Base Class	Description
≪Or≫	Class	The 'or' represents an option group(blank arc).

The notation used for a 'or' variability element is a UML class stereotyped as ≪Or≫.

Element: FeatureAttribute

Stereotype	Base Class	Description
≪FAttribute≫	Class	A common extension to feature models are attributes that are attached to features. They are often used to describe issues such as cardinality etc.

The notation used for a feature element is a UML class stereotyped as ≪FAttribute≫.

Relationship: SubFeatures

Stereotype	Base Class	Description
≪subFeatures≫	Dependency	The relationship is used to organize features into a hierarchy and variability elements. In a feature model, a 'subFeature' represents an arc to a dependent feature.

The notation used for the relationship is a dependency stereotyped as ≪subFeatures≫.

Relationship: feAttributes

Stereotype	Base Class	Description
≪feAttributes≫	Dependency	The relationship associates feature attributes to features.

The notation used for the relationship is a dependency stereotyped as ≪feAttributes≫.

Relationship: Excludes

Stereotype	Base Class	Description
≪f_excludes≫	Dependency	Selection of the source feature prohibits a selection of the target feature.

The notation used for the relationship is a dependency stereotyped as ≪f_excludes≫.

Relationship: Requires

Stereotype	Base Class	Description
≪f_requires≫	Dependency	Selection of the source feature requires the selection of the target feature.

The notation used for the relationship is a dependency stereotyped as ≪f_requires≫.

B.6 Well-Formedness Rules

We define no well-formedness rules for OCL as constraints normally are not checked in current modeling tools. Instead, the constraints are formulated in Triple. At present, our group works on a service that enables checking of OCL constraints based on metamodels. This service will allow checking the described diagrams in UML tools such as Poseidon.

Appendix C

Taxonomies

The appendix summaries the taxonomies of .Net and J2EE. The taxonomies are composed based on the works of Liao [81] and Gädicke [51], who researched J2EE, Jini and .Net technologies. Properties that describe the interna of a communication mechanism - the - implementation have been removed.

C.1 Remote Method Invocation Taxonomy

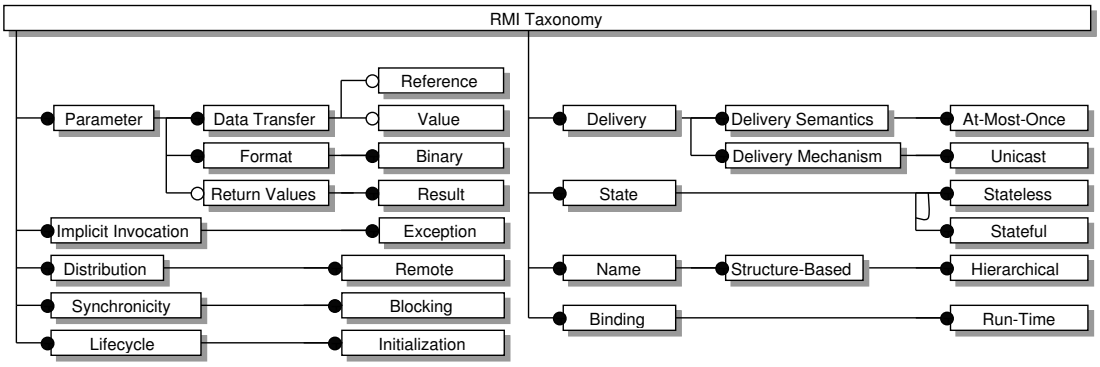

Figure C.1: Remote Method Invocation Taxonomy

C.2 EntityBean Taxonomy

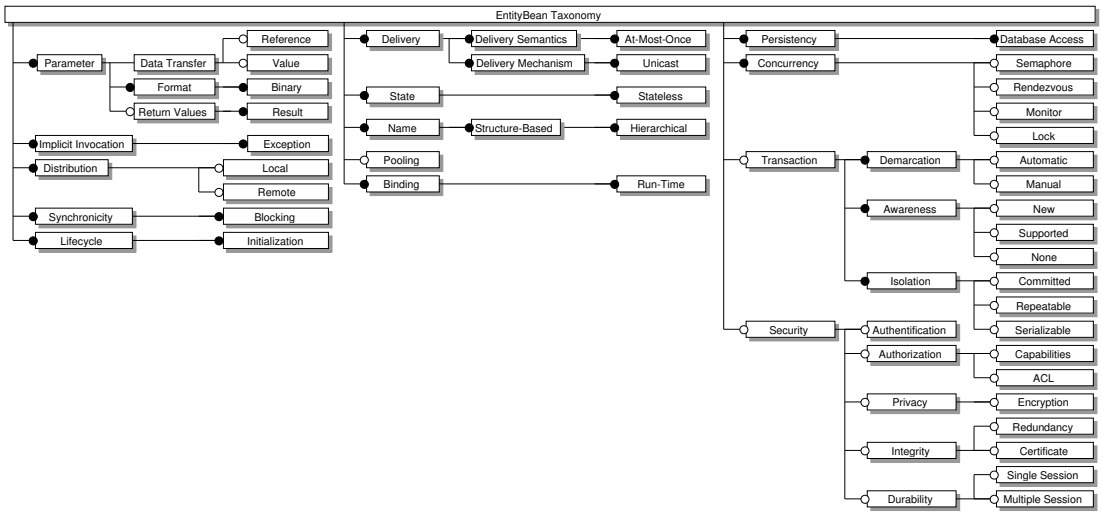

Figure C.2: EntityBean Taxonomy

C.3 Message Taxonomy

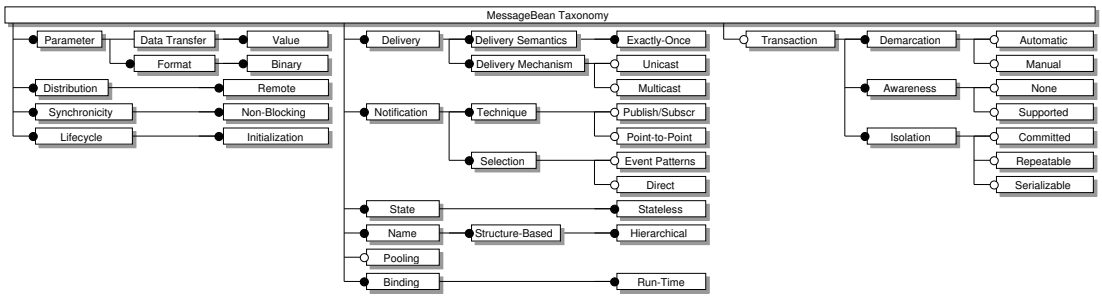

Figure C.3: MessageBean Taxonomy

C.4 Net Remoting Taxonomy

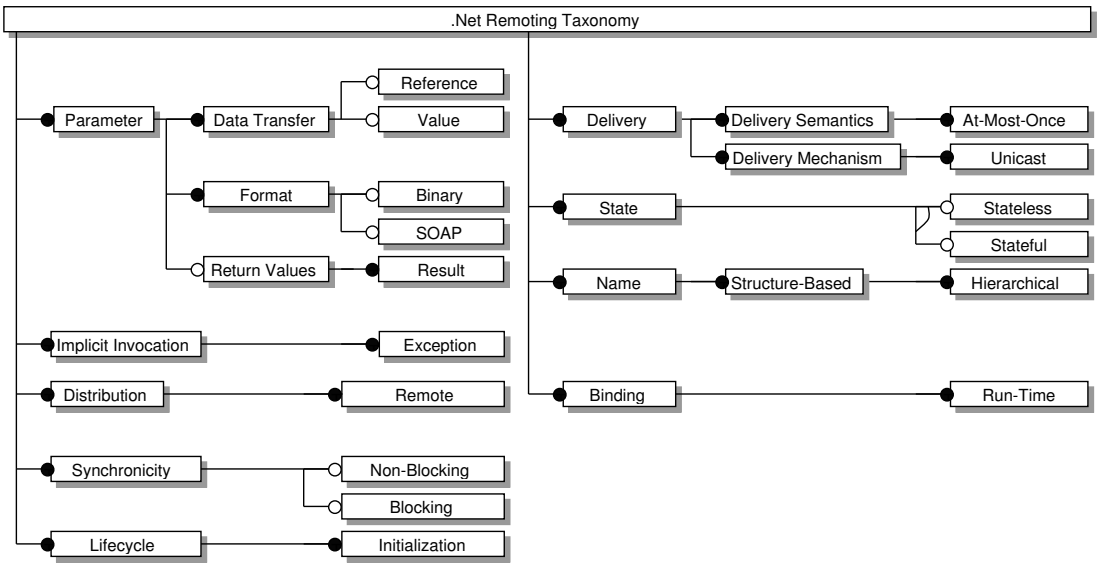

Figure C.4: Net Remoting Taxonomy

C.5 Serviced Component Taxonomy

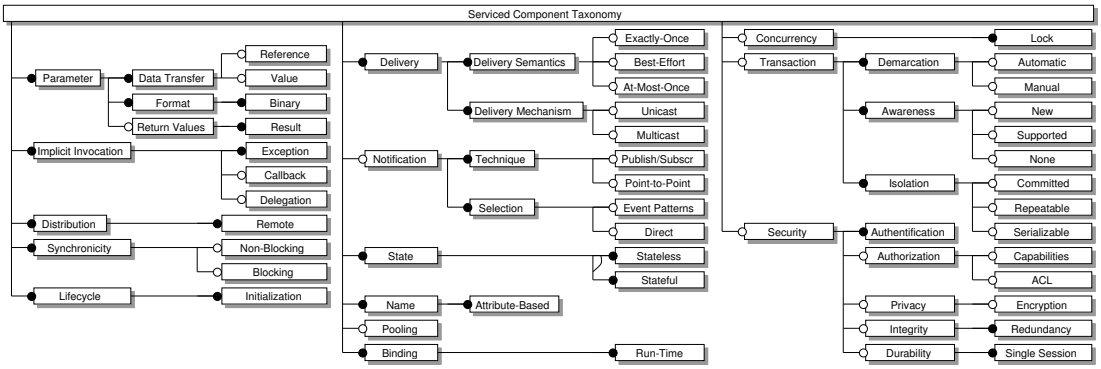

Figure C.5: Serviced Component Taxonomy

Appendix D

Published Papers

The appendix summaries previously published papers from which segments were integrated in this work:

- Leicher, A. A framework for identifying compositional conflicts in component-based systems. Tech. Rep. 2004-23, TU Berlin, 2004.
- Leicher, A., Busse, S., and Süß, J. G. Analysis of compositional conflicts in component-based systems. In Software Composition: 4th International Workshop, SC 2005 (2005), no. 3628 in Lecture Notes in Computer Science (LNCS).
- Leicher, A., and Süß, J. G. Augmenting uml models for composition conflict analysis. In Fundamental Approaches to Software Engineering (FASE) (2005).
- Billig, A., Busse, S., Leicher, A., and Süß, J. G. Platform independent model transformation based on TRIPLE. In Proceedings of Middleware 04, ACM/IFIP/USENIX International Middleware Conference (2004), H.-A. Jacobsen, Ed., no. 3231 in Lecture Notes in Computer Science (LNCS).
- Süß, J. G., Leicher, A., Weber, H., and Kutsche, R.-D. Model-centric engineering with the evolution and validation en vironment. In UML 2003 - The Unified Modeling Language: Modeling Lan guages and Applications, 6th International Conference, San Francisco, CA, USA (2003), P. Stevens, J. Whittle, and G. Booch, Eds., vol. 2863 of Lecture Notes in Computer Science (LNCS), Springer, pp. 31 43.

List of Figures

List of Tables

References

[1] JBoss enterprise middleware system (JEMS). http://www.jboss.org/products/index, 2002. 21 June 2005.

[2] Borland Janeva. http://www.borland.com/janeva, 2005. Borland, 21 June 2005.

[3] How do you define software architecture? http://www.sei.cmu.edu/architecture/definitions.html, 2005. Carnegie Mellon University, Software Engineering Institute (SEI), 24 June 2005.

[4] J-Integra interoperability suite: Bridging java to microsoft (com, .net, exchange). http://j-integra.intrinsyc.com/, 2005. Intrinsyc Software International, Inc., 21 June 2005.

[5] Osgi technology. http://www.osgi.org/osgi_technology/index.asp?section=2, 2005. OSGi, 6 October 2005.

[6] Overview - k2 - icmg: K2 component server. http://www.icmgworld.com/corp/k2/k2.overview.asp, 2005. iCMG, 21 June 2005.

[7] Plan 9 (operating system). http://en.wikipedia.org/, 2005. Wikimedia Foundation Inc., 21 June 2005.

[8] Programming:ada:tasking. http://en.wikibooks.org/wiki/Programming:Ada:Tasking, 2005. Wikimedia Foundation Inc., 21 June 2005.

[9] The protege ontology editor and knowledge acquisition system. http://protege.stanford.edu/, 2005. Stanford Medical Informatics, 21 June 2005.

[10] Aldrich, J., Chambers, C., and Notkin, D. Archjava: Connecting software architecture to implementation. In *Proceedings of the 22rd International Conference on Software Engineering, ICSE* (2002).

[11] Allen, R., and Garlan, D. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM) 6*, 3 (1997), 213–249.

[12] Alur, D., Crupi, J., and Malks, D. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall / Sun Microsystems Press, 2001.

[13] Amadio, R. M., and Cardelli, L. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS) 15*, 4 (1993), 575–631.

[14] Arnold, K., O'Sullivan, B., Scheifler, R. W., Waldo, J., and Wollrath, A. *The Jini Specification*. Addison-Wesley, 1999.

[15] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., and Zettel, J. *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Weseley, 2002.

[16] Baeten, J. A brief history of process algebra. Tech. Rep. CSR 04-02, TU Eindhoven, 2004.

[17] Baeten, J., and Weijland, W. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[18] Balek, D., and Plasil, F. Software connectors: A hierarchical model. Tech. Rep. 2000/2, Dep. of SW Engineering, Charles University, Prague, November 2000.

[19] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*. Software Engineering Institute. Addison-Wesley, 1998.

[20] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice - Second Edition*. Addison-Wesley, 2003.

[21] BEA Systems, Inc. *BEA WebLogic Server, Using WebLogic Server Clusters*, March 2001.

[22] Bennett, K. Legacy systems: Coping with success. *IEEE Software 12*, 1 (1995), 19 – 23.

[23] Bernstein, P. A. Middleware: A model for distributed system services. *Communications of the ACM 39*, 2 (1996).

[24] Billig, A. ODIS - Ein Domänenrepository auf der Basis von Semantic Web Technologien. In *Tagungsband der Berliner XML Tage* (2003), XML-Clearinghouse. english version: http://www-berlin.isst.fhg.de/~abillig/Odis/xsw2003.

[25] Billig, A., Busse, S., Leicher, A., and Süss, J. G. Platform independent model transformation based on TRIPLE. In *Proceedings of Middleware 04, ACM/IFIP/USENIX International Middleware Conference* (2004), H.-A. Jacobsen, Ed., no. 3231 in Lecture Notes in Computer Science (LNCS).

[26] Biron, P. V., and Malhotra, A. *XML Schema Part 2: Datatypes*. World Wide Web Consortium (W3C), 2001.

[27] Bisbal, J., Lawless, D., Wu, B., and Grimeson, J. Legacy information systems: Issues and directives. *IEEE Software 16*, 5 (1999), 103 – 111.

[28] Brien, S., and Nicholls, J. *Z Base Standard*. Oxford University, 1991.

[29] Brodie, M. L., and Stonebraker, M. *Migrating Legacy Systems, Gateways, Interfaces & The Incremental Approach*. Morgan Kaufmann Publishers, Inc., 1995.

[30] Bruneton, E., Coupaye, T., and Stefani, J. *The Fractal Component Model*. The ObjectWeb Consortium, February 2004. http://fractal.objectweb.org/.

[31] Bures, T., and Plasil, F. Communication style driven connector configurations. In *Software Engineering Research and Applications: First International Conference, SERA 2003, San Francisco* (2004), C. V. Ramamoorthy, R. Lee, and K. W. Lee, Eds., vol. 3026 of *Lecture Notes in Computer Science (LNCS)*.

[32] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture: A System of Pattern*. John Wiley & Sons, 1996.

[33] Busse, S. A specification language for model correspondence assertions, part i: Overlap correspondences. Forschungsberichte des Fachbereichs Informatik Nr. 99-8, Technische Universität Berlin, 1999.

[34] Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR) 17*, 4 (1985), 471–523.

[35] Carnegie Mellon University, Software Engineering Institute. *CBS Overview*, 2004. http://www.sei.cmu.edu/cbs/overview2.htm,27. June 2004.

[36] Chinnici, R., Gudgin, M., Moreau, J.-J., Schlimmer, J., and Weerawarana, S. Web services description language (wsdl) version 2.0 part 1: Core language. Tech. rep., World Wide Web Consortium (W3C), 2004. W3C Working Draft.

[37] Clauss, M. Modeling variability with uml. In *Proc. 3rd Int. Conference Generative and Component-Based Software Engineering (GCSE)* (2001), vol. 2186 of *Lecture Notes in Computer Science (LNCS)*, Springer.

[38] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. *Documenting Software Architecture: views and beyond*. Software Engineering Institute (SEI). Addison-Wesley, Boston, 2003.

[39] Clements, P., and Northrop, L. *Software Product Lines: Practices and Patterns*. Kluwer, 2001.

[40] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems, Concepts and Design, Third Edition*. Addison-Wesley, 2001.

[41] Czarnecki, K., and Eisenecker, U. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[42] Czarnecki, K., and Helsen, S. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture* (Anaheim, October 2003).

[43] Davis, L., Flagg, D., Gamble, R., and Karatas, C. Classifying interoperability conflicts. In *COTS-Based Software Systems: Second International Conference, ICCBSS 2003 Ottawa, Canada* (2003), T. W. H. Erdogmus, Ed., no. 2580 in Lecture Notes in Computer Science (LNCS), Springer, pp. 62–71.

[44] Davis, L., Gamble, R., and Payton, J. The impact of component architectures on interoperability. *Journal of Systems and Software 61*, 1 (2002), 31–45. based on the Technical Report UTULSA-MCS-99-30.

[45] Day, J., and Zimmerman, H. The OSI Reference Model. *Proceedings of the IEEE 71*, 12 (1983).

[46] Emmerich, W. Software engineering and middleware: a roadmap. In *Proceedings of the conference on The future of Software engineering* (2000), ACM Press, pp. 117–129.

[47] Fernandez, J.-C. Aldébaran: A tool for verification of communicating processes. Tech. rep., Laboratoire de Génie Informatique - Institut IMAG, 1989.

[48] Fisher, K., and Mitchell, J. C. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems 4*, 1 (1998), 3–25.

[49] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[50] Freeman, E., Hupfer, S., and Arnold, K. *Java Spaces - Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[51] Gädicke, J. Metadatengestützte Analyse Der Kommunikationsfähigkeit Von Enterprise Java Beans und .Net. Master's thesis, TU Berlin, 2004. german.

[52] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[53] Garlan, D., Allen, R., and Ockerbloom, J. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering* (Seattle, Washington, April 1995), pp. 179–185.

[54] Garlan, D., Monroe, R. T., and Wile, D. Acme: An architecture description interchange language. In *Proceedings of CASCON'97* (Toronto, Ontario, November 1997), pp. 169–183.

[55] Garlan, D., Monroe, R. T., and Wile, D. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.

[56] Gelernter, D. Generative communication in linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (Januar 1985), 80–112.

[57] Gerber, A., Lawley, M., Raymond, K., Steel, J., and Wood, A. Transformation: The missing link of mda. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation* (London, UK, 2002), Springer, pp. 90–105.

[58] Greenfield, J. UML Profile For EJB. Tech. rep., Rational Software Corporation, May 2001. http://www.jcp.org/jsr/detail/26.jsp, Java Community Process (JCP).

[59] Guttag, J. V., and Horning, J. J., Eds. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[60] Hoare, C. Monitors: An operating system structuring concept. *Journal of the ACM 17*, 10 (1974).

[61] Hoare, C. *Communicating Sequential Processes*. Prentice Hall PTR, 1985.

[62] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*, 1990. IEEE Std 610.12-1990.

[63] IEEE Computer Society. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, 2000. IEEE Std 1471-2000.

[64] International Telecommunications Union (ITU). *Message Sequence Charts*, 2004. Recommendation Z.120.

[65] Inverardi, P., and Tivoli, M. Software architecture for correct components assembly. In *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 2003), M. Bernardo and P. Inverardi, Eds., vol. 2804 of *LNCS Tutorial*, Springer.

[66] ISO/IEC. *Reference Model of Open Distributed Processing: Part I: Overview*, 1995. International Standard 10746-1.

[67] ISO/IEC. *Reference Model of Open Distributed Processing: Part II: Foundations*, 1995. International Standard 10746-2.

[68] ISO/IEC. *Reference Model of Open Distributed Processing: Part III: Architecture*, 1995. International Standard 10746-3.

[69] Joy, B., Steele, G., Gosling, J., and Bracha, G. *The Java Language Specification*. Addison-Wesley, 2000.

[70] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, nov 1990.

[71] Kaplan, Y. API spying techniques for windows 9x, NT and 2000. http://www.internals.com/articles/apispy/apispy.htm, 2001. 21 June 2005.

[72] Kelkar, A., and Gamble, R. Understanding the architectural characteristics behind middleware choices. In *1st International Conference in Information Reuse and Integration* (1999).

[73] Keshav, R., Gamble, R., Payton, J., and Frasier, K. Architecture integration elements. Tech. Rep. UTULSA-MCS-16-99, University of Tulsa, May 1999.

[74] Kifer, M., Lausen, G., and Wu, J. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM 42* (Juli 1995), 741–843.

[75] Konstantas, D. Interoperation of object-oriented applications. In *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis, Eds. Prentice Hall PTR, 1995.

[76] Kruchten, P. Modeling component systems with the unified modeling language - a position paper for icse'98 workshop, 1998. Int. Workshop on Component-Based Software Engineering.

[77] Leicher, A. A framework for identifying compositional conflicts in component-based systems. Tech. Rep. 2004-23, TU Berlin, 2004.

[78] Leicher, A., Busse, S., and Süss, J. G. Analysis of compositional conflicts in component-based systems. In *Software Composition: 4th International Workshop, SC 2005* (2005), vol. 3628 of *Lecture Notes in Computer Science (LNCS)*, Springer.

[79] Leicher, A., and Süss, J. G. Augmenting uml models for composition conflict analysis. In *Fundamental Approaches to Software Engineering (FASE)* (2005).

[80] Leman, D. Spying on com objects. *Windows Developer's Journal 10*, 7 (July 1999).

[81] Liao, Y. Metadatengestützte Analyse der Kommunikationsfähigkeit zweier ausgewählter Komponententechnologien. Master's thesis, TU Berlin, 2004. german.

[82] Labelled transition system analyser (ltsa). http://www.doc.ic.ac.uk/ jnm/book/ltsa/LTSA.html, 21 June 2005, 1999.

[83] Luckham, D. C., Kenney, J. L., Augustin, L. M., Vera, J., Bryan, D., and Mann, W. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering 21*, 4 (1995), 336–355.

[84] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying Distributed Software Architectures. In *Proc. 5th European Software Engineering Conf. (ESEC 95)* (Sitges, Spain, 1995), W. Schafer and P. Botella, Eds., vol. 989, Springer-Verlag, Berlin, pp. 137–153.

[85] Magee, J., Dulay, N., and Kramer, J. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering 1*, 5 (1994), 304–312.

[86] Magee, J., and Kramer, J. *Concurrency, State Models & Java Programs*. John Wiley & Sons, 1999.

[87] Marinescu, F. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, Inc., 2002.

[88] Mattison, R. *Data Worehousing: Stratgies, Technologies, and Techniques*. McGraw-Hill, 1996.

[89] McBride, B. Jena: Implementing the rdf model and syntax specification. Semantic Web Workshop, WWW, 2001.

[90] McDirmid, S., Flatt, M., and Hsieh, W. C. Jiazzi: new-age components for old-fasioned java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (New York, NY, USA, 2001), ACM Press, pp. 211–222.

[91] McIlroy, M. Mass produced software components. In *NATO Conference on Software Engineering* (Oktober 1968), P.Naur and B. Randel, Eds., NATO Science Commitee.

[92] Medvidovic, N. A classification and comparison framework for software architecture description languages. Tech. rep., Department of Information and Computer Science, University of California, Irvine, 1996.

[93] Mehta, N. R. Software connectors: A taxonomy approach. In *Workshop on Evaluating Software Architectural Solutions 2000* (2000), Institute for Software Research University of California, Irvine. http://www.isr.uci.edu/events/wesas2000/position-papers/mehta.pdf.

[94] Mehta, N. R., and Medvidovic, N. Understanding software connector compatibilites using a connector taxonomy. In *In Proceedings of First Workshop on Software Design and Architecture (SoDA02), Bangalore, India* (December 2002).

[95] Mehta, N. R., Medvidovic, N., and Phadke, S. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering* (2000), ACM Press, pp. 178–187.

[96] Mellor, S. J., Clark, A. N., and Futagami, T. Model-driven development. *IEEE Software 20*, 5 (2003), 14–18.

[97] Meyer, B. *Object-Oriented Software Construction, Second Edition*. Prentice Hall PTR, 1997.

[98] Miller, J., and Mukerji, J. Model Driven Architecture(MDA). Tech. Rep. ormsc/2001-07-01, Object Management Group(OMG), Architecture Board ORMSC, July 2001.

[99] Miller, J., and Mukerji, J. MDA Guide Version 1.0. Web (www.omg.org), May 2003. Document Number omg/2003-05-01.

[100] Miller, M., Yee, K.-P., and Shapiro, J. S. Capability myths demolished. Tech. Rep. SRL2003-02, Johns Hopkins University, Systems Research Laboratory, 2003.

[101] Morisio, M., and Sunderhaft, N. Commercial-off-the-shelf (cots): A survey. Tech. rep., Data & and Analysis Center for Software (DACS), December 2000. Prepared for: Ari Force Research Laboratory - Inforation directorate (AFRL/IFED), http://iac.dtic.mil/dacs.

[102] Neubus, N. Entwicklung eines Matching-Algorithmus fü ein Komponenten-Repository . Master's thesis, TU Berlin, 2005. german.

[103] Oberleitner, J., Gschwind, T., and Jazayeri, M. Vienna component framework enabling composition across component models. Tech. rep., Technische Universität Wien, 2002.

[104] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. http://www.omg.org/cgi-bin/doc?ptc/03-08-02.

[105] Object Management Group (OMG). *Meta Object Facility (MOF) Specification*, April 2002. Version 1.4.

[106] Object Management Group (OMG). *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, April 2002. http://www.omg.org/cgi-bin/apps/do_doc?ad/2002-04-10.pdf.

[107] Object Management Group (OMG). *OCL 2.0 Specification*, 2003. Final Adopted Specification.

[108] Object Management Group (OMG). *Unified Modeling Language Specification Version 1.4.2*, Juli 2004. formal/04-07-02.

[109] OMG. *UML Profile for CORBA Specification V1.0*, 2000.

[110] Pahl, C. An ontology for software component matching. In *6th International Conference, FASE 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland* (April 2003), M. Pezze, Ed., no. 2621 in Lecture Notes in Computer Science (LNCS).

[111] Paolucci, M., Kawmura, T., Payne, T., and Sycara, K. Semantic matching of web services capabilities. In *First Int. Semantic Web Conf.* (2002).

[112] Pierce, B. C. *Types and programming languages*. MIT Press, 2002.

[113] Plasil, F., Balek, D., and Janecek, R. SOFA/DCUP: Architecture for component trading and dynamic updating. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems* (Washington, DC, USA, 1998), IEEE Computer Society, p. 43.

[114] Plasil, F., and Stal, M. An architectural view of distributed objects and components in CORBA, java RMI and COM/DCOM. *Software - Concepts and Tools 19*, 1 (1998), 14–28.

[115] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., and Burt, C. A unified approach for the integration of distributed heterogeneous software components. In *Proceedings of the Monterey Workshop on Engineering Automation for Soware Intensive System Integration* (2001).

[116] Reisig, W. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer, Berlin, Germany, 1985.

[117] Ressouche, A., de Simone, R., Bouali, A., and Roy, V. *The FCTools User Manual*. INRIA and Ecole des Mines/CMA. http://www.iist.unu.edu/ alumni/software/other/inria/www/fc2tools/index.html.

[118] Schmid, F. Erstellung eines Repositories für die Verwaltung und Transformation von Komponentenmodellen. Master's thesis, TU Berlin, 2004. german.

[119] Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Distributed Objects*. John Wiley & Sons, 2000.

[120] Schwanke, R. W., Strack, V. A., and Werthmann-Auzinger, T. Industrial software architecture with gestalt. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design* (Washington, DC, USA, 1996), IEEE Computer Society, p. 176.

[121] Seacord, R. C., Plakosh, D., and Lewis, G. A. *Moderinizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. SEI Series in Software Engineering. Addison-Wesley, 2003.

[122] Seco, J. C., and Caires, L. A basic model of typed components. In *ECOOP 2000 - Object-Oriented Programming* (2000), E. Bertino, Ed., Lecture Notes in Computer Science (LNCS).

[123] Shaw, M., and Clements, P. C. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of the 21st International Computer Software and Applications Conference* (1997), IEEE Computer Society, pp. 6–13.

[124] Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall PTR, April 1996.

[125] Singh, I., Stearns, B., Johnson, M., and Team, E. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley, 2002.

[126] Sintek, M., and Decker, S. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In *Proceedings of International Semantic Web Conference ISWC 2002* (2002), vol. 2342 of *Lecture Notes in Computer Science (LNCS)*, Springer.

[127] Siram, N. N., Raje, R. R., and Olson, A. M. An architecture for the uniframe resource discovery service. In *Proceedings of the 3rd International Workshop on Software Engineering and Middleware (SEM)* (2002).

[128] Spitznagel, B., and Garlan, D. A compositional formalization of connector wrappers. In *Proccedings of the 2003 International Conference on Software Engineering (ICSE'03)* (2003).

[129] Spivey, J. M. *Understanding Z: A Specification Language and its Formal Semantics*, vol. 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Jan. 1988.

[130] Sreedhar, V. C. Mixin'up components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ACM Press, pp. 198–207.

[131] Süss, J. G., Leicher, A., Weber, H., and Kutsche, R.-D. Model-centric engineering with the evolution and validation en vironment. In *UML 2003 - The Unified Modeling Language: Modeling Lan guages and Applications, 6th International Conference, San Francisco, CA, USA* (2003), P. Stevens, J. Whittle, and G. Booch, Eds., vol. 2863 of *Lecture Notes in Computer Science (LNCS)*, Springer, pp. 31 – 43.

[132] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[133] Süss, J., Leicher, A., and Busse, S. OCLPrime - Environment and Language for Model Query, Views, and Transformations. In *OCL 2.0 - Industry standard or scientific playground?, Workshop on the 6th Int. Conf. UML 2003* (2003).

[134] Tai, S., and Rouvellou, I. Strategies for integrating messaging and distributed object transactions. In *Middleware 2000, IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing* (2000), J. Sventek and G. Coulson, Eds., no. 1795 in Lecture Notes in Computer Science (LNCS), Springer.

[135] Tannenbaum, A., and Van Steen, M. *Distributed Systems: Principles and Paradigms*. Pearson, 2002.

[136] Taylor, R. N., Medvidovic, N., Anderson, K. M., Jr., E. J. W., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering 22*, 6 (1996).

[137] Thompson, J. Avoiding a middleware muddle. *IEEE Software 14*, 6 (November/December 1997).

[138] Vallecillo, A., Hernández, J., and Troya, J. Component interoperability. Tech. Rep. ITI-2000-37, Dept. Lenguajes y Ciencias de la Computación, University of Málaga, July 2000.

[139] van Glabbeek, R. J. What is branching time semantics and why to use it? In *The Concurrency Column*, M. Nielsen, Ed. *Bulletin of the EATCS* 53, 1994, pp. 190–198. Also available as Report STAN-CS-93-1486, Stanford University, 1993, at `http://theory.stanford.edu/branching/`, and in G. Paun, G. Rozenberg & A. Salomaa, editors: *Current Trends in Theoretical Computer Science; Entering the 21st Century*, World Scientific, 2001.

[140] van Glabbeek, R. J., and Weijland, W. P. Branching time and abstraction in bisimulation semantics. *Journal of the ACM 43*, 3 (1996), 555–600.

[141] van Lamsweerde, A. Formal specification: a roadmap. In *Proceedings of the conference on The future of Software engineering* (2000), ACM Press, pp. 147–159.

[142] Vitharana, P., Zahedi, F., and Jain, H. Design, retrieval, and assembly in component-based software development. *Communications of the ACM 46*, 11 (2003).

[143] Warmer, J. B., and Kleppe, A. G. *The object constraint language: precise modeling with UML*. Addison-Wesley, 1999.

[144] Weber, H., and Ehrig, H. Specification of concurrently executable modules and distributed modular systems, September 1988. Workshop on the Future Trends of Distributed Computing Systems in the 1990s.

[145] Wegner, P. Interoperability. *ACM Computing Surveys (CSUR) 28*, 1 (1996), 285–287.

[146] Weissbach, M. Analyzing user requirements for customizable pim-to-psm transformations. Master's thesis, TU Berlin, 2005.

[147] Wiederhold, G. Mediators in the Architecture of Future Information Systems. In *Readings in Agents*, M. N. Huhns and M. P. Singh, Eds. Morgan Kaufmann, San Francisco, CA, USA, 1997, pp. 185 – 196.

[148] Wing, J. M. A specifier's introduction to formal methods. *IEEE Computer 23*, 9 (September 1990), 8 – 23.

[149] World Wide Web Consortium (W3C). *RDF Primer: W3C Recommendation 10 February 2004*, 2004. http://www.w3.org/TR/rdf-primer/,21 June 2005.

[150] World Wide Web Consortium (W3C). *RDF Vocabulary Description Language 1.0: RDF Schema: W3C Recommendation 10 February 2004*, 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/,21 June 2005.

[151] Yakimovich, D., Bieman, J. M., and Basili, V. R. Software architecture classification for estimating the cost of cots integration. In *Proceedings of the 21st international conference on Software engineering* (1999), IEEE Computer Society Press, pp. 296–302.

[152] Zaremski, A. M., and Wing, J. M. Specification matching of software components. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 1995), ACM Press, pp. 6–17.

Index