Visual Modelling and Validation of Distributed Systems

Kumulierte Habilitation an der Fakultät IV -
Elektrotechnik und Informatik
der Technischen Universität Berlin

Dr. Gabriele Taentzer

Lehrgebiet: Informatik
Eröffnung des Verfahrens: 21. November 2002
Verleihung der Lehrbefähigung: 9. Juli 2003

Habilitationsausschuß:
Vorsitzender: Prof. Dr. G. Hommel
Prof. Dr. H. Ehrig
Prof. Dr. G. Engels (Paderborn)
Prof. Dr. K. Geihs
Prof. Dr. B. Mahr

Gutachter:
Prof. Dr. H. Ehrig
Prof. Dr. G. Engels (Paderborn)
Prof. Dr. M. Goedicke (Essen)
Prof. Dr. F. Parisi-Presicce (Rom)

Berlin, 2003
D 83

Visual Modelling and Validation of Distributed Systems

Survey of Habilitation Thesis

Gabriele Taentzer
Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
E-mail: gabi@cs.tu-berlin.de

September 18, 2003

Abstract

Distributed system technologies are fast developing and the complexity of networked systems increases. For this reason the precise design of distributed systems is necessary, comprising all key aspects. The employment of formal methods is restricted to few aspects such as performance and correctness of distributed algorithms, while the investigation of consistency issues, as they arise e.g. when data is shared between different sites, is still a challenge. In this work, approaches to visual modelling and validation of distributed systems are considered focussing on the Unified Modeling Language (UML) as visual modelling language and graph transformation as formal validation domain. To meet the main requirements for distributed system modelling, UML has been extended by a number of profiles and heavy-weight language extensions. For visual reasoning about certain key aspects of distributed systems, the formal calculus of distributed graph transformation has been developed. This calculus supports the formal validation of distributed systems, especially concerning concurrency and consistency issues.

To precisely defining the syntax and semantics of a visual modelling language like UML, graphs play a central role, since they are well suited to store the multi-dimensional structures behind visual representations. As a consequence, graph grammars are shown to be a promising technique to define visual languages. Theoretical results for graph transformation can be advantageously used to speed up visual parsing and to show the functional behaviour of model translations. Compared to other approaches to visual language definition, graph grammars allow a fully visual approach which handles all structural aspects visually. The presented approach builds the basis for the precise syntax and semantics definition of visual modelling languages for distributed systems.

Zusammenfassung

Die Entwicklung von verschiedensten Technologien für verteilte Systeme ist rasant vorangegangen und hat zu einer stetig wachsenden Komplexität von Softwaresystemen dieser Art geführt. Deshalb ist ein präziser Entwurf von verteilten Systemen, der zumindest die wichtigsten Systemaspekte modelliert, sinnvoll. Die Verwendung von formalen Methoden ist üblicherweise auf wenige Aspekte, wie z.B. Performanz und Korrektheit von verteilten Algorithmen, beschränkt. Konsistenzeigenschaften, wie sie beispielsweise bei der gemeinsamen Datennutzung durch verschiedene Prozesse auftreten, sind noch nicht hinreichend untersucht worden. Diese Arbeit befasst sich mit Modellierungs- und Validationskonzepten für verteilte Systeme, wobei wir uns auf die Unified Modeling Language (UML) als visuelle Modellierungssprache sowie Graphtransformation als formalen Validationskalkül einschränken. Um die wichtigsten Anforderungen an das Modellieren von verteilten Systemen zu erfüllen, wurden eine Reihe von Spracherweiterungen an der UML durchgeführt. Zur visuellen Validation von verteilten Systemaspekten wurde der formale Kalkül der verteilten Graphtransformation entwickelt. Dieser Kalkül unterstützt im wesentlichen den Nachweis von Konsistenz- und Nebenläufigkeitsaussagen.

Graphen spielen eine zentrale Rolle, wenn es um die präzise Definition der Syntax und Semantik von visuellen Modellierungssprachen geht, denn sie eignen sich hervorragend zum Speichern von mehrdimensionalen Strukturen, speziell visuellen Repräsentationsstrukturen. Folglich sind Graphgrammatiken eine vielversprechende Technik zur Definition von visuellen Sprachen. Theoretische Resultate für Graphtransformation können gewinnbringend eingesetzt werden, um den Parsierungsprozess von visuellen Darstellungen zu beschleunigen und um das funktionale Verhalten von Modeltransformationen zu zeigen. Im Vergleich zu anderen Ansätzen zur Definition von visuellen Sprachen erlauben Graphgrammatiken eine vollständig visuelle Definition, die alle strukturellen Aspekte visuell behandelt. Der in der Arbeit vorgestellte Ansatz stellt eine Grundlage für die präzise Syntax- und Semantikdefinition von visuellen Modellierungssprachen für verteilte Systeme dar.

Contents

1 Introduction **1**

2 Requirements for Distributed System Modelling **4**
2.1 Main Aspects of Distributed Systems 4
2.2 Challenges 5
2.3 Examples of Distributed Systems 7
2.4 Summary of Requirements 12

3 Visual Modelling Languages **14**
3.1 UML 15
3.2 Visual Language Definition and Parsing 23

4 Model Validation **25**
4.1 Validation Issues 26
4.2 Validation Techniques for UML 28
4.3 Model Translation by Graph Transformation 30

5 Graph Transformation as Validation Domain **32**
5.1 Structured Graphs and Graph Transformation 33
5.2 Distributed Graph Transformation as Semantic Domain 36
5.3 Validation Support by Graph Transformation 40

6 Summary of the Papers Submitted for the Habilitation Thesis **43**
6.1 A Visual Modelling Framework for Distributed Object Computing 43
6.2 Consistency Checking and Visualization of OCL Constraints . . . 43
6.3 Application of Graph Transformation to Visual Languages 44
6.4 A Combined Reference Model- and View-based Approach to System Specification 45
6.5 Distributed Graphs and Graph Transformation 45
6.6 Distributed Graph Transformation With Application to Visual Design of Distributed Systems 45
6.7 The AGG-Approach: Language and Environment 46

7 Conclusions **47**

1 Introduction

Distributed systems technologies developed fast over the last decades. Especially the Web and other Internet-based applications and services have become of great interest. A range of useful languages, tools and environments have been provided supporting the development of distributed applications. Due to its increasing complexity, a distributed system can hardly be well designed without modelling at least the key aspects of the system. Up to now, resource sharing has been the primary aim for the development of distributed systems. Another proven concept is redundancy of resources which improves the performance and the fault-tolerance of a system. Since redundancy is expensive and increases the complexity of distributed systems, good validation methods which are available already at design time, are highly desirable. So far the main reasons to develop formal models for distributed systems have been performance issues and correctness of distributed algorithms. Consistency issues have not been of interest.

Today resource sharing is taken for granted, but effective data sharing still is a challenge. For example, data sharing plays an important role when considering Distributed Shared Memory (DSM) systems. In DSM systems, data is shared logically between computers that do not share physical memory. The main point of DSM systems is that they appear more high-level to the developer than usual message passing systems, since developers do not have to care about distribution issues for shared data such as marshalling the data. The acceptance of DSM systems is very much dependent on the efficiency with which they can be implemented. Concurrent and conflicting updates may arise and it depends heavily on the underlying consistency model, if a DSM system has an acceptable performance. But occurring consistency issues are not yet well investigated in the literature. Typical formal modelling techniques such as Statecharts, Petri nets, and I/O-automata, are used for process modelling and do not support an attractive integration of process and information modelling on a complete formal basis.

Focussing on information modelling, the Unified Modeling Language (UML) has evolved to the standard visual modelling language for object-oriented systems. For a flexible adaptability to further domains, UML supports a standard extension mechanism by defining profiles. There exists a strong interest of researchers and companies to complete UML by profiles such that it becomes the universal modelling language for any kind of system. Recently, the comprehensive profile for Enterprise Distributed Object Computing (EDOC) has been developed providing a lot of new concepts to model the important aspects of distributed systems. Only few aspects are still missing in UML, e.g. concepts for application-specific views and an adequate visual constraint language.

The definition of the visual modelling language UML is semi-formal, i.e. its syntax and semantics are only partially defined in a precise manner. Compared to textual languages, there does not exist a standard formalism like the Backus-Naur-Form (BNF) and attribute grammars, to define the syntax and semantics of a visual language (VL). For each VL, separate formalisms for precise language definition, parsing and translation to some semantic domain are given. For UML, a constraint-based approach, the so-called metamodel, is used which

provides class diagrams to define all types of model elements and their relations as well as constraints of the Object Constraint Language (OCL) to restrict to allowed structures. Other visual languages like the Specification and Description Language (SDL) are defined by grammar-based approaches. Constraint and grammar-based approaches use textual notions to a large extent which make the multi-dimensional structure of diagrams and graphics difficult to understand and to reason about.

The semantics of UML as specified by a standard of the Object Management Group (OMG), is even more informal. It is described in natural language. A large variety of semantic domains exist for UML which map UML models partly or completely to semantic models focussing on certain system aspects. Especially the behaviour part of UML models has been considered and mapped to any sort of process model. A precise semantic model for investigating consistency issues in distributed systems is not yet focussed in the literature.

Outline of the Habilitation Thesis

The habilitation thesis consists of this survey and seven research papers which present the author's contributions most relevant for the research on visual modelling and validation of distributed systems. The following outline refers directly to the work achieved in these papers. This survey embeds these achievements into a larger context taking a large variety of related work into account.

The central framework for the habilitation thesis is outlined in [1]. After clarifying the main requirements for distributed system modelling, UML is discussed as visual modelling language for distributed systems. For offering a precise, but intuitive approach to key aspects of distributed systems the formal calculus of distributed graph transformation [5] has been developed. Distributed graph transformation combines process and information modelling and can serve as semantic model domain for a visual model [6] where all main aspects of distributed systems can be considered. Moreover, distributed graph transformation offers a validation support concerning issues seldom considered in distributed system models, such as conflicts and dependencies of actions as well as consistency issues. Choosing it as semantic domain for UML models offers the possibility to reason about consistency of shared object structures as well as conflicting and dependent updates. Furthermore, process issues such as an event structure semantics are worthwhile to be investigated. The validation process can be supported by the AGG tool [7] offering an environment which allows to develop, test and analyze graph transformation systems.

UML serves well as visual modelling language for distributed systems, especially since there exist various extensions to adapt it to different problem domains. Concerning distributed system modelling, we found out that most of the interesting concepts have been already captured by some extension. One main extension is OCL, a constraint language used to formulate system invariants, pre and post conditions as well as guards. Unfortunately, OCL is textual and not yet well integrated with UML. Basing OCL on collaborations as done in [2], paves the ground for a much closer integration with UML. Surely, a UML-like graphical notation will increase the attractiveness of OCL consideringly. In

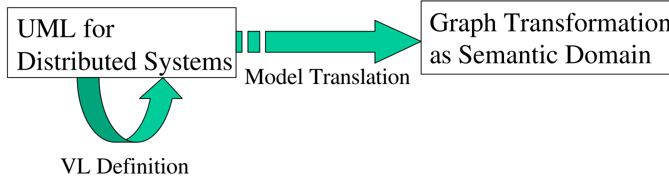

Figure 1: Main topics and interrelations of the thesis

addition, a concept for application-specific views [4] has been developed which is useful to support distributed specifications.

Having considered all language concepts of an adequately extended UML, it has to be precisely defined for further investigations. The UML metamodel provides a definition of all symbols and relations in their abstract form, i.e. visual alphabet of the abstract syntax. Well-formedness rules which are given in form of textual OCL constraints, restrict the set of all possible models to the syntactically meaningful ones. In this approach, graphs are used as a natural means to describe the inherent structure of diagrams. But instead of using OCL as textual formalism for language description, graph grammars are much better suited to define the abstract syntax of a visual language [3]. Similarly to the BNF, the rule-based manipulation of syntactical forms is performed by graph derivations. Adding attributes to store representational data and moreover, semantical information, the graph grammar approach is well-suited to define the concrete and abstract syntax of VLs as well as to translate to some semantic domain. VLs are precisely defined and translations from concrete to abstract syntax as well as to a semantic domain can be formally investigated, e.g. functional behaviour of translations can be proven. Performing the definition of UML sublanguages shows how clear and precise a VL can be defined following the graph grammar approach. These and further examples show that graph transformation has a great potential to become the "BNF for visual languages".

Figure 1 sketches the main topics of this thesis and their interrelations.

Organization of the Survey

This article gives an overview on research activities concerning visual modelling of distributed systems and the validation support which has led to a number of published papers. First the requirements for distributed system modelling are discussed (Section 2). In Section 3, UML as adequate visual modelling language for distributed systems as well as precise definition and parsing of visual languages are considered. Section 4 contains a discussion of the main validation issues, discusses validation techniques for UML and shows how models in some abstract syntax can be translated to a semantic domain by graph transformation. Taking graph transformation as semantic domain, Section 5 first presents the concepts of structured graph transformation which are needed to define the mapping of all main aspects of distributed systems to this semantic domain. Thereafter, available validation techniques are discussed together with their tool support. The presentation style in this survey is informal, formal definitions and further details can be found cited papers, especially in those papers referred to

in bold letters. At the end of this article, a summary of the papers submitted for the habilitation thesis is given.

Acknowledgements

Special thanks go to my supervisor Hartmut Ehrig and to Martin Große-Rhode who both gave me valuable hints, especially concerning the presentation of my work. Furthermore, I thank Rosi Bardohl, Claudia Ermel, Julia Padberg, Anilda Qemali, and Ingo Weinhold for proof reading this thesis.

2 Requirements for Distributed System Modelling

Distributed system modelling is an evolving discipline which mainly has coped with performance issues and quality of service. But there are further aspects of distributed systems to carefully think about. In the following, we first introduce distributed systems and discuss the main design challenges in general. Thereafter, three examples of distributed systems with their special challenges are presented, namely *mobile computing*, *distributed shared memory systems*, and *web applications*. At the end of this section, the requirements for distributed system modelling are summarized.

2.1 Main Aspects of Distributed Systems

A distributed system consists of a number of hardware or software components which are located at networked computers. These components usually proceed concurrently but also have to communicate and interact with each others to achieve productive work within the system. Communication is performed by message passing or logically shared memory which is distributed physically. Each distributed process has its own clock, thus there does not exist a global clock. Distributed processes usually work asynchronously and have to run synchronization protocols to get actions executed simultaneously. They even can fail partially, e.g. one component stops while the others are still running. That means each component in the system has to detect whether other components failed which is not always possible to find out. After a failure happened, the running components should react appropriately.

Typically computer networks are organized in layers: the basic layers are concerned with the network, above those the middleware layers provide a common programming platform, used in the upper layers where distributed applications reside. Well-known examples of distributed systems are the Internet, intranets, and mobile computing. Distributed applications like Web services are distributed systems which only consist of software components belonging all to the application layer. A comprehensive presentation of distributed systems, their concepts and design principles, is given in [41].

Reasons for developing a distributed system are either inherent distribution of its components like the computers in the Internet or sharing resources. A resource is an abstract notion for any system part. It ranges from hardware

components like processors, disks and printers to software-defined entities such as files, databases, and any kind of data object.

2.2 Challenges

In the following, we discuss the main challenges a designer of distributed systems usually meets and has to take into account carefully when developing the system.

Heterogeneity. Developing a distributed application, especially on the Internet, confronts the designer with many different sorts of computers and networks. The developer has not only to deal with different hardware and operating systems, but also with heterogeneous programming languages. Moreover, for larger applications, several developers are engaged following different encoding styles. One main concept to mask the *heterogeneity* of network layers is that of middleware which provides a programming abstraction for distributed applications such that part of the heterogeneity is capsuled.

Openness. The *openness* of a distributed system determines the degree how far the system can be modified and extended by new components. To achieve openness the key interfaces have to be published. This is usually done on the programming level using standards as the Common Object Request Broker Architecture (CORBA) IDL [17] or, more recently, the Simple Object Access Protocol (SOAP) [15] which is based on XML. Moreover, open systems provide a uniform communication mechanism and published interfaces for access to shared resources. In this way, open systems can be constructed from heterogeneous hardware and software, but the conformance of each component to the published standard must be carefully tested and verified to be sure that the system works correctly.

Scalability. Distributed systems can be found at many different scales, ranging from two processor machines to the Internet. A system is called *scalable* if it will remain effective when the number of resources and clients significantly increases. The design of scalable distributed systems has the following challenges: It must be possible to add server computers to avoid performance bottlenecks that would arise if a single server had to handle all requests. Considering services which heavily rely on data sets whose size is proportional to the number of users or resources in the system, their main algorithms have to be carefully designed. For example, hierarchical net structures scale better than linear ones. For a system to be scalable, the maximum performance loss should be no worse than the one in hierarchical systems. Moreover, software resources such as Internet addresses should not run out. This goal is very difficult to meet since especially networking software is supposed to run for a long time and designers cannot predict developments over years. In general, algorithms should be decentralized to avoid bottlenecks. Usually, replication mechanisms are used to improve the performance of resources which are heavily used such as certain Web pages. To handle possible scalability problems as early as possible

the developer should be supported in configuration management and should be provided with techniques for early performance tests.

Concurrency. Distributed applications and services usually include a high amount of *concurrency* which might lead to complex control flow difficult to overlook during programming and even more difficult to test by distributed debugging. Often resources are shared by several clients which might lead to simultaneous attempts to access a shared resource. The process which manages a shared resource could take only one client request at a time. Since this approach limits the throughput, services usually allow multiple client requests to be processed concurrently. A programmer has to take care about shared resources and to ensure that operations access them in a synchronized way such that its data remains consistent.

Partial failures. In distributed applications, it may happen that one component fails while others continue to function, i.e. *partial failures* can occur. Due to this fact, the handling of failures is particularly difficult in distributed systems. Usual techniques to handle partial failures are failure detection, e.g. checksums are used to detect corrupt data in messages or files, and masking of failures. For example, messages can be retransmitted when they fail to arrive. Further techniques are tolerating failures, e.g. informing the user when a Web browser cannot contact a Web server and leave the further proceeding to the user. Another technique is recovery from failures meaning to bring server data to a consistent state after a failure occurred. Moreover, redundancy of hardware components is a popular technique to make systems more fault tolerant. A developer has to be aware of all the kinds of failures which can occur in a distributed system and to choose the best practice for each kind of failure.

Security. *Security* issues are of considerable importance in distributed systems. Security for information resources has three aspects: protection against unauthorized access, against alteration or corruption, and against interference with the means to access the resources. A typical example occurs in electronic commerce where users send their credit card numbers across the Internet. Although resource protection is the main security issue, it is not the only one. Another important security mechanism are digital signatures and certificates to trust certain information given by others. A system designer has to choose an adequate security policy meeting all security requirements while keeping the process and management costs at a minimum.

Transparency. Due to its intrinsic complexity another important design issue for a distributed system is the *transparency* (i.e. hiding) of certain aspects to the users. The degree of transparency is much dependent on the design of a distributed system. The scope of transparency can be broad: The access to remote resources can be transparent, i.e. local and remote resources are accessed using identical operations or at least the server location of remote resources is not known like it is when typing the address of a certain web page. Furthermore,

the user might not be aware of concurrent processes, replication of resources, partial failures, movements of resources to other places, or performance-specific reconfigurations.

Availability and Fault Tolerance. Data replication is a key for providing high availability and fault tolerance in distributed systems. The caching of data at clients and servers is by now familiar as a means to performance enhancement. Furthermore, data is sometimes replicated transparently between several servers in the same domain. A general requirement for replicated data is that of *consistency*. Dependent on the application, different forms of consistencies are considerable and the developer has to choose the adequate one.

Having clarified the main aspects of distributed systems and the general challenges when constructing them, we now discuss three examples of distributed systems and look closer at their specific challenges.

2.3 Examples of Distributed Systems

In the following, we present three examples of distributed systems belonging to different system layers. While mobile computing is mainly concerned with a platform for distributed applications, the other two examples belong to the application layer itself describing distributed shared memory systems and web applications. All examples are first introduced, and their main challenges are discussed which results in requirements for their adequate modelling. After this subsection, we draw our conclusion from these sample cases and present a summary with the main requirements for distributed system modelling.

Mobile Computing. Mobile computing is concerned with the performance of computing tasks while the user is on the move, i.e. is changing their environment. Portable devices like mobile phones, laptops, handhelds, etc. together with their ability to connect conveniently to networks in different places, often wireless, makes mobile computing possible. Mobile computing raises important system issues imposing new requirements on system modelling. A mobile device should be easily connectable to a local network. The adaptation of a mobile device to its current environment should be done transparently for the user, e.g. they should not have to type in the name or address of local services to achieve connectivity. The available services in the current environment should be discovered by the devices themselves such that the user does not have to do special configurations, e.g. nowadays DHCP servers [18] are installed in many networks which automatically assign IP addresses to new devices and thus, allow a convenient access to the Internet.

Once connected to a network, local services have to be discovered, e.g. the nearest local printer which is accessible by the device requesting a printer service. It is the purpose of discovery services to accept and store details of services that are available in the network and to respond to queries from a client about them. That means a discovery service offers a registration service which records

details of available services, and a lookup service which accepts queries concerning available services. Jini [19] is a Java-based system that is designed for mobile networking. Among others it provides facilities for service discovery. The discovery-related components in a Jini system are lookup services, Jini services and Jini clients. A crucial point is the bootstrapping connectivity. If a Jini client or service enters the network, it must use the lookup service. But how can it locate the lookup service? Jini uses multicast to a well-known IP multicast address which is known by all instances of Jini software.

In mobile environments, query processing has to be well designed to meet best the typical restrictions in bandwidth and energy. Different types of queries can be distinguished: concerning broadcasted and fast changing data. Broadcasted data addresses a large number of clients. What should be the best organization of broadcasted data so that the energy spent on the client's side is minimized? Which information should be broadcasted and which should be provided "on demand"? How will a contiguous query which is used to keep track of the value of the query in a changing environment over time, be evaluated using broadcasted data? Another kind of queries concerns fast changing, update intensive data. Location management viewed as establishing locations of individual mobile users is one special case of this problem. Precise tracking may be impossible or simply unnecessary. Instead, we may have to store incompletely defined information. As a consequence, queries will be answered in an approximate way. Scale is a very important factor when dealing with the above two categories of queries. The benefit of data broadcasting is proportional to the number of users serviced and the cost of broadcasting does not depend on the number of users. Therefore broadcasting seems to be the method particularly suitable for a large number of clients having the same query. In location management, scalability of solutions is also extremely important: what may work for 100 users in a local area network that may not work for thousands of users in the wide area network, e.g. informing the location server about each user's move will not scale up in large networks.

Security and privacy are important issues in mobile computing although it is largely neglected presently. Here, we concentrate on the software issues and will not discuss the also very important and difficult problem of security on the physical level. Some systems track the moves of users, e.g. "active badge" technology [97], and this may threaten the user's privacy. To protect privacy it is necessary to enable and enforce the specification of personal profiles of users. In such profiles, users should be able to determine whom, when and where authorization is given to reach them, e.g. due to energy restrictions one may want to restrict the list of users who are allowed to "wake up" the mobile unit to send a message to it. Once a profile is created, it has to be managed. I.e. where are such profiles to be stored? Should the profile migrate with the user? A good solution has to cope with an adequate authentication policy for data access.

Since the mobile terminals will often be disconnected from the static part of the network, transactions will often be processed locally on the cached data. The degree of connectivity of mobile devices to the fixed part of the network can vary widely. Mobile users will "check out" data for long periods of time. Thus, new

methods of cache synchronization reflecting different degrees of connectivity will be necessary. Especially, when mobile hosts carry shared data, lock and commit protocols have to be redefined to meet the new requirements. How should integrity constraints which involve data residing on a number of mobile and fixed hosts be efficiently maintained? Here, the main question is how to partition data among static and mobile hosts such that the constraint can be maintained without having to contact mobile hosts all the time.

Distributed Shared Memory Systems Distributed shared memory (DSM) systems are a kind of distributed systems where components that do not share physical memory communicate logically by sharing data. Components access DSM by reads and updates of data which appears to be local but in fact is distributed over a network. An underlying infrastructure has to make sure that the shared data is consistent at all sites. Updates made at one site have to be replayed at all other sites.

The main advantage of DSM is that the software developer does not have to concern of message passing for communication. DSM is usually less appropriate in client-server systems, where clients normally request server-held resources. However, servers can provide DSM that is shared between clients. An example of this kind of DSM system are distributed version management systems where a number of revision archives are distributed over different sites and share large parts of software project documents. Each revision archive cares about the version control at one site where clients can store and access their revisions of documents. Another example are chat systems where clients have to register at a server. Thereafter, the server provides all clients registered with the new chat contributions. One of the first notable examples of a DSM application was the Apollo Domain file system [69] where files are shared between different processes. Originally, DSM systems have been considered in connection with shared memory multiprocessors.

Critical issues of DSM systems are the *efficiency* of their implementation and the degree of *consistency* of shared data to be reached. In message passing systems, all remote data accesses are explicit and therefore the developer is always aware of the expense of communication. Using DSM, however, any particular data access may or may not involve communication costs. A data update, for example, may cause immediate updates of all its replications. Another possibility is a lazy update which is performed first when this data is read again. The update policies in DSM systems are highly dependent of particular applications and the facilities of underlying infrastructures. The performance of the implementation depends directly on the degree of consistency to be guaranteed. Strict consistency which means that written values are instantaneously available at all sites, is usually very expensive. Temporal inconsistencies which allow to delay propagating updates to all peers, can result in considerably faster realizations.

An issue that is closely related to the structure of DSM is the granularity of sharing. Considering e.g. a shared file system it would clearly be very wasteful always to transmit the entire file system as answer to a read or update request.

A more adequate unit of sharing would certainly be a single file or directory. Taking directories as unit for sharing it may be likely that two components try to access the same directory, although two different files within this directory have to be accessed. The phenomenon is known as false sharing and is caused by choosing too large units. This effect does not occur when files are taken as unit of sharing. In this case, however, access of whole directories would cause a considerable overhead of communication costs. The determination of adequate unit sizes is an important factor in developing efficiently working DSM systems.

For the development of DSM applications, middleware such as Linda [34] and its derivate JavaSpaces [9] support the sharing of memory in a platform-neutral way. JavaSpaces realizes tuple spaces of Java objects. The main access operations are *write*, *read*, and *take* to create a new object, read an object and delete an object from the tuple space if it exists. Moreover, an object can register to be notified if a specified object has been written into the tuple space. JavaSpaces is realized on top of JavaRMI [10] which offers the possibility to invoke methods of remote objects.

Web Applications. The World Wide Web (WWW), created by Berners-Lee while at CERN[1], is an evolving system for publishing and accessing documents and services across the Internet. Through commonly available web browsers, users use the Web to retrieve and view documents of any kind, or to interact with an unlimited set of services. A web application is a software application that is accessible using a web browser or HTTP user agent. The Hyper Text Transfer Protocol (HTTP) [12] defines exactly how a browser should format and send a request to a Web server.

The Web is a hypermedia system, because the resources in the system are linked to one another. It is especially this idea of a link which led to the term Web. The structure of links can be arbitrarily complex and the set of resources that can be added is unlimited, the web of links is indeed world-wide.

The Web is an open system, i.e. it can be extended by new components without disturbing its existing functionality. First, communication between Web browsers and Web servers is standardized by the HTTP. Second, the Web is open with respect to the types of resources that can be published and shared on it. The simplest resource is a Web page or some kind of content which is storable in a file, e.g. text in PostScript or PDF format, images, audio streams, etc. Web pages are usually written in HTML, the Hyper Text Markup Language [11], a standardized tag language used to express the content and visual formatting of a Web page. If Web pages also contain parts which accept user input, they are called Web forms. Nowadays the Web moved beyond simple Web pages and encompasses also applications, such as on-line banking, on-line registration, and any kind of E-commerce. Web applications include an application server that enables the system to manage the business logic. The application server often consults or updates a database when processing a request. CGI (Common Gateway Interface) programs are the standard way to allow Web users to run applications on the server. Sometimes the developers of Web ap-

[1] CERN is the European center for nuclear research.

plications require some application-related code to run inside the browser. For this purpose a script language such as JavaScript [8] can be used which has quite limited functionality. Going beyond, applets can be downloaded and run by a client. Applets are Java applications which have restricted functionality due to security issues, described in the so-called sand box model.

Web services are a special kind of Web application. They are self-contained components that can be published, located, and invoked across the Web. Web services perform functions, which can be anything from simple requests to complicated business processes. Web services are services that are made available from a business's Web server for Web users or other Web-connected programs. The accelerating creation and availability of these services is a major Web trend. The basic platform for web services is XML together with HTTP. On top of that the Simple Object Request Protocol (SOAP) [15] is used which provides a protocol specification that defines a uniform way of passing XML-encoded data. For dynamically finding other web services, the Universal Description, Discovery and Integration Service (UDDI) [23] has been introduced. It comprises registry and discovery facilities for web services. The Web Service Definition Language (WSDL)[14] provides a way for service providers to describe the basic format of web service requests over different protocols or encodings. WSDL is used to describe what a web service can do, where it resides, and how to invoke it.

Although the Web has a phenomenal success, there are some problems when designing web applications [36]. First, there are problems concerning the web structure. The hypertext model gets easily "dangling links" when target resources are moved or deleted. Links to these resources still remain which causes frustration for users, since resources cannot be found anymore. Furthermore, information in the Web is semi-structured, often users follow confusingly many links and get "lost in hyperspace". Meta information about Web resources would be helpful to support users in finding what they are looking for.

Another problem which is one of the most common challenges of Web applications is managing the client state on the server. Due to HTTP the communication between client and server is connectionless, i.e. the communication takes place without first establishing a connection. A server does not have an easy job to keep track of the state of each client using the service. Often the relevant information is stored on the client side, e.g. in form of so-called cookies being short strings of characters. Cookies are persistent and can even last beyond the lifetime of a browser's execution. This mechanism is simple but not totally secure, cookies can become risky when one service gets access to another's cookies which might contain personal data like identification information.

Using Web applications requires often the input of various user data which might have a lot of interdependencies, e.g. on-line registration to a conference comprises the input of personal data, conference-specific selections, accommodation data and paying information. These data are highly interdependent, e.g. the conference fee is dependent on the user's profession, the selected events and the date of registration. Submitting input data to the server without previous validation means that the data has to be checked by the server. In case of inconsistencies or missing data an error message is produced and the user has to resubmit the revised data. This causes a high communication traffic and could

distract the user from using the service. A better solution would be to already validate the user's input data on client side. But although Web applications usually control user interfaces by Java applets which have to be downloaded first, application developers are often not aware of all data interdependencies and provide poor user interfaces not guiding the user enough and not validating the input completely.

In the Web, problems of scalability can easily occur. Popular server pages may have many "hits" per second, and as a result the response to users is slow. The practice of caching in browsers and proxy servers is used to reduce communication traffic, but a typical Web client has no efficient means to keep users up to date with the latest versions of pages. Users have to reload pages explicitly to check whether a local copy of a resource is still valid. Especially during longer sessions caching might become problematic. Caching every page could become too expensive, intelligent caching is needed which keeps track of the most important information.

For searching Web services more efficiently, WSDL has been developed. It provides a meta structure to describe interface signatures, but does not support a semantical description of the services, e.g. in form of certain kinds of constraints.

Designing Web applications two major activities are different from designing other (non-distributed) systems: object partitioning between clients and servers and defining Web page user interfaces. Usually, objects for input field validation and specialized user interface widgets reside on the client, while the main business objects will exist on the server. In general, objects should reside where they have easiest access to data and collaborations they require.

2.4 Summary of Requirements

Although distributed applications can be found everywhere, they are seldom backed by a model capturing all the main design decisions concerning architecture, data structure and interaction. A modelling technique for distributed object computing should be able to cope with a variety of design aspects in a flexible and intuitive way and should support validation of essential properties. In the following, we state the modelling requirements which follow from the challenges described above.

Intuitive Method. First of all, the method should support an intuitive appealing representation of all important system aspects. For example, it is natural to represent computer networks as well as web page structures visually by some kind of graph structure. All modelling concepts should be easy to learn and to understand such that the model can be easily communicated to other developers as well as to customers. Moreover, the technique should provide different views to separate system concerns. Views support a convenient approach to the modelled system by different stake holders, such as developers, customers, and quality managers.

Abstraction Levels. An adequate model has to offer different abstraction levels, concentrating on the main aspects first, and taking details into account later. In the beginning, one may ask what are the main entities of the system and how do they interact? What are the characteristics of their behaviour? The model should help to make explicit all the relevant assumptions about the system to be modelled and to discuss possible generalizations when designing structures and algorithms, given by those assumptions. Despite all possible heterogeneity the key design should be independent of that. Concrete systems, languages, and protocols are relevant first on a more detailed design level.

Components. The distributed computing model should support a component-based approach. Components with clear interaction only through interfaces are the main feature to handle complex (distributed) systems. Clearly they are the basis to facilitate open systems like the Web which can consist of heterogeneous parts. To add a new system part, only the interfaces of existing components are needed. Moreover, a component-based approach supports the distributed development of a system by different teams. Altogether, components are the very means for flexible distributed system architectures and thus, essential in every modelling technique for distributed computing.

Interaction Modelling. Reasoning on concurrent execution of processes is one of the main purposes for developing a distributed computing model. The modelling technique should support the natural description of concurrent control flow as well as pre and post conditions of distributed operations. [74] is a comprehensive work on distributed algorithms considering key problems for concurrent and distributed processes such as mutual exclusion or consensus problems. Furthermore, caching and replication mechanisms should be adequately describable. Considering the kinds of failures which can occur, a distributed computing model should also give the possibility to express them all and should support the modelling of techniques how to deal with failures such that a high availability of the whole system can be guaranteed. Moreover, there should be the possibility to model and reason about security strategies such as protection against unauthorized users, manipulation and corruption of resources, and accessibility of resources.

Formal Semantics. Concurrent process execution can easily become too complex to follow, thus one of the main requirements of a distributed computing model is a clear formal background for reasoning about interesting process properties like deadlocks, progress, and termination. Moreover, a distributed computing model should offer means to describe precisely all interesting forms of consistency as they occur when using distributed shared memory, and should offer means to validate a modelled system concerning the chosen consistency constraints. Considering web applications, we noticed that input data validation is important which leads to a kind of domain-specific consistency constraints. A precise constraint modelling with validation support is highly desirable to improve design decisions such as the adequate consistency model and unit of

sharing for DSM.

Reconfiguration Facilities. A distributed computing model should offer the possibility to model the increase of resources and clients, i.e. it should have reconfiguration facilities. Here, testing of scalability, i.e. the performance when the system scales up, should be supported. To achieve transparency, views should be definable supporting aspect- and component-oriented selection of information. A special modelling requirement comes from mobile computing: Due to continuously changing environments, concepts for modelling system evolution of network and data structures have to be provided.

Standardized Technique. To specify and document open interfaces on a high level - as needed for web services - distributed computing models could help to clear the understanding. For this purpose a proven technique should be standardized.

3 Visual Modelling Languages

Nowadays visual languages and visual environments emerge in an increasing frequency. This development is driven by the expectation to achieve a considerable productivity improvement when using visual languages and tools. Visual languages are developed for very different tasks. They are most successful for system modelling and building graphical user interfaces. Furtheron, visual programming languages, and visual database query languages have been developed, but with less user acceptance.

In the following subsection, we consider the visual modelling language per se, the Unified Modeling Language (UML) [95] being the quasi standard language for modelling software systems. Since we focus in this work especially on modelling distributed systems, those extensions of UML are chosen for presentation which concern distributed system issues. The main basis for long living open heterogeneous systems is a well-suited component concept. UML offers two main approaches to components which are both presented and compared. Furtheron, there are extensions concerning failures, security, and evolution. Even for web applications as special kind of distributed systems, the Web Application Extension (WAE) is available. But certain aspects are missing: an application-specific view concept and a visual constraint language. A clear concept of application-specific views is needed, especially when large systems are developed where engineers with different skills and background specify system parts in a distributed manner. As previous discussions of requirements for distributed system modelling manifested, a powerful constraint language is needed to state system invariants expressing consistency constraints, to express pre and postconditions of operations and services such as web services, and to build consistent user interfaces such as input validating web forms. Although OCL [98] has been designed as constraint language for UML, it is seldom used, since it is textual and quite complicated to use. A visualization of OCL is much more attractive for users. The visualization of OCL presented here relies on

UML collaborations. In this way, it integrates much better with UML itself than the textual OCL.

Although UML is the visual modelling language as such, we have to mention two further visual languages for distributed system modelling, especially used to model reactive systems. These are the Specification and Description Language (SDL) [13] developed for modelling telecommunication systems, and Petri nets [82, 84] which can advantageously be used to visualize concurrent control flow and which are especially useful to reason about distributed algorithms [85]. In the following, however, we concentrate on UML as the most comprehensive and promising visual modelling language for distributed systems.

3.1 UML

The Unified Modeling Language is used for visual modelling of software systems following an object-oriented design methodology. UML is an accepted standard of the Object Management Group (OMG).

UML and one of its extensions, the profile for Enterprise Distributed Object Computing (EDOC) [44], are often used for architecture modelling where component diagrams describe the logical system structure and deployment diagrams handle its mapping to underlying hardware resources. The EDOC profile defines process components, ports, connections, and roles, while a kind of Statecharts is chosen to model the behaviour of each component. Statecharts are also used in other contexts for the behaviour description of concurrent processes.

UML defines a number of diagram types which describe the static, dynamic, and management aspects of a system. The twelve kinds of diagrams defined by the UML standard, are grouped into three classes:

1. *Static Structure Diagrams.* A *class diagram* is a graph of classifier elements connected by their various static relationships. A classifier is one of many static system constructs such as classes, interfaces, packages, relationships, etc. An *object diagram* is an instance of a class diagram indicating the static relationships between object instances, frozen in time.

   Component diagrams are meant to document the overall structure of a system itself. *Deployment diagrams* show configurations of real processing systems. Elements contained in deployment diagrams include computers, network nodes, processes, objects, connections, etc.

2. *Behaviour Diagrams.* A *use-case diagram* shows the relationship among actors (active entities including software elements and human users) and use cases (usage scenarios) in a system. It is mostly used to formulate requirements. A *Statechart diagram* shows the sequences of states that an object goes through during its life-cycle in response to stimuli. It also documents the object's responses and actions. An *activity diagram* is a variation of a state machine in which the states are "activities" representing the execution of operations. By the completion of an operation, transitions may be triggered. A *sequence diagram* represents an object interaction in which messages are exchanged among a set of objects to effect

a desired result. A *collaboration diagram* shows interaction among objects as well, additionally the object's relationships are presented. Unlike a sequence diagram, time is not visualized in a collaboration diagram, i.e. the sequence of messages and concurrency must be indicated by sequence numbers.

3. *Model Management Diagrams.* Model management diagrams include *packages*, *subsystems*, and *models*.

UML also provides a number of general mechanisms for annotating and extending system specifications. The following light-weight constructs are provided: stereotypes, constraints and tagged values. A *stereotype* is a new class of model element that is introduced at modelling time by specializing existing model elements. Generally, a stereotype indicates a usage or semantic extension. A *constraint* is a semantic relationship among model elements that specifies invariant conditions and propositions. Any value attached to a model element (attributes, associations, tagged values, etc.) is a property. A *tagged value* is a keyword value pair that can be attached to any model element. Tagged values permit arbitrary annotation of models and model elements. Such an annotation is considered as extension when the tagged values are precisely defined. All these constructs are light-weight extensions of UML in contrast to a change of the UML metamodel itself which is considered as heavy-weight extension.

Several extensions have been elaborated to adapt UML to the requirements for distributed system modelling as stated in Subsection 2.4: Most important for distributed system modelling is a well-suited component concept. Two main proposals are discussed and compared below. Thereafter, further extensions concerning web applications, failures, security, evolution, and application-specific views are presented. Moreover, a visualization of OCL based on collaborations is presented. Each extension concerns a certain system aspect which is orthogonal to the others, i.e. to meet all requirements all extensions have to be integrated with each other.

Components. The main aspect when developing distributed systems, is to design a good component structure, with as small interfaces as possible to reduce communication to the essentials. The basis for a good architecture supporting open heterogeneous systems which might evolve over life time, is a well-suited component concept.

One approach to modelling component systems can be found in the UML specification itself [95]. Here, component diagrams are proposed for this purpose. UML components can be considered as concrete software pieces that realize abstract interfaces. A component diagram is meant to show the dependency structure between components. Dependencies between components are usually defined through interfaces. UML component diagrams show the classical use relation between two components where provided functionality is presented in interfaces. However, it is not possible to state required functionality for a use relation between components stating what has to be imported.

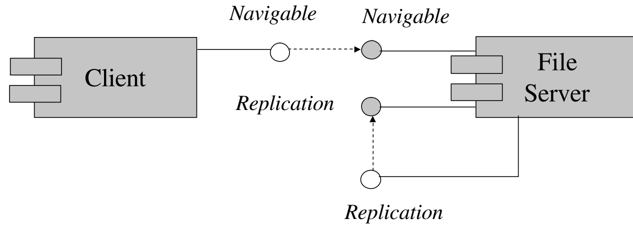

Figure 2: Modified component diagram with *required* and *provided* ports

In [73], ports have been introduced which are distinguished in two kinds, *required* and *provided* ports. Consider Fig. 2 which shows an extended form of component diagram using *required* and *provided* ports. The required ports are white, while the provided ports are black. A use relation from a required to a provided one means that the provided port meets all requirements of the using component. Similarly as in the component collaboration architecture described below, ports could be stereotypes of classes which reside on certain components.

A component diagram has been used to show the logical structure of components in principal. That's why they only occurred in type form, and not in instance form. In the newest version of UML, version 1.4, the type and instance level have been explicitly distinguished for components. Another important step towards a sufficient component description in UML has been done. For the instance form a deployment diagram has to be used, a possibly degenerated one without nodes if the system is not distributed. Otherwise, the actual distribution of component instances is described by nodes within deployment diagrams. In principle, a component might have multiple instantiations of a particular type of interface. However, it remains unclear if this is valid also for UML components.

Furthermore, especially when considering open distributed systems, reconfiguration of the architecture during runtime becomes an important issue. Since UML deployment diagrams deal with component instances, they are also able to capture dynamics, although this is not explicitly presented in [95]. Execution constraints can be defined on instances which describe the life time. These constraints are described by tags {new}, {destroyed} and {transient} and can advantageously used as notation for reconfiguration. In Fig. 3 two simple reconfigurations are shown creating a new "Client" and "FileServer" instance, resp. Moreover, the migration of an instance can be expressed by a <<become>> relationship between the instance at its old place and at its new place.

The connectors between components are described rather rudimentary in component diagrams. Either components are in a dependency relation or they contain each other. A more detailed description of these relationships is left to future work.

In [44] the UML profile for Enterprise Distributed Object Computing (EDOC) is specified which contains another approach to component modelling. This profile uses the Reference Model for Open Distributed Processing (RM-ODP) [80] as conceptual basis. RM-ODP follows the "separation of concern"-principle and provides five viewpoints for system specification: the enterprise

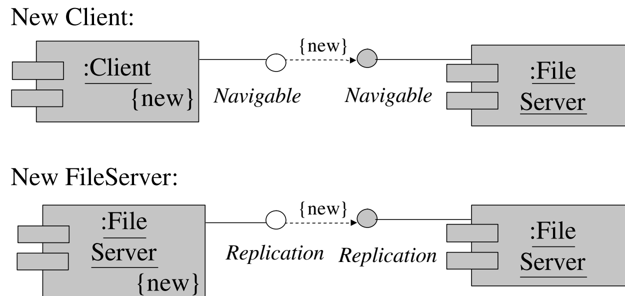

Figure 3: Two simple architecture reconfigurations

viewpoint which contains the business model, the computational viewpoint which describes the system through objects and their communications, the information viewpoint which specifies the information structures and processing, the engineering viewpoint which describes distribution relevant design decisions, and the technology viewpoint describing the concrete technology (hardware and software) to be used. The UML profile for EDOC is structured in the Component Collaboration Architecture (CCA) and four further profiles for events, entities, relationships and processes. The CCA as well as the events, entities, and process profiles are used for the enterprise and the computational specification. The information specification uses the entities and relationships profiles.

CCA is a component model which incorporates concepts adapted from the Object-Oriented Role Analysis Method (OORAM) and from Real-Time Object-Oriented Modeling (ROOM) [89]. Its notion of components has the same basis as UML components in [35]. CCA relies on processing components which provide a composition operation. Components can implement an arbitrary number of protocols which are needed to specify collaboration with other components using messages. Protocols are provided at ports which are the connection points between components. Compositions define how components are used. They are used to build composed components out of other components. **ProcessComponents** are stereotypes of classifiers whereas **Component Usages** which can be considered as templates of instances, are classifier roles. Although this choice seems to be quite natural, it is not perfect, since a component instance might define additional structure not specified in its type. However, in UML an object being the instance of some classifier role, cannot include parts that its classifier does not also define. A **Port** is a stereotyped class which is linked to its owner component. This solution allows ports to store some kind of state which is not possible when ports would be stereotyped interfaces. But there is no longer a clear semantic distinction between ports and components. **Connections** between components are stereotyped association roles. Since components are stereotyped classifiers, this approach is straight forward and well suited to basically describe the relationship between components. But instead of associations, connections are supposed to be modelled independently of classifiers storing additional information concerning e.g. states and roles. **Compositions** describe whole component systems and are stereotyped collaborations. Since

component usages are classifier roles and connections are association roles, this choice looks natural, but there is a semantic mismatch. While collaborations describe a representative interaction between objects, an architectural configuration is meant to capture a complete description at some level of abstraction.

Summarizing up, two main approaches for a UML component concept exist, UML component diagrams and the Component Collaboration Architecture. In [53], Garlan et.al. compare several approaches to component modelling in UML capturing the predecessors of the approaches discussed here as well. Although a bit out-of-date, this article is still useful to understand precisely the characteristics of each approach: While UML component diagrams are a natural approach capturing the main, but not all aspects of components, the CCA approach is comprehensive, but contains some semantic mismatches. This is due to the decision to provide a profile and to avoid any heavy-weight extension of UML. A combination of both approaches taking the concepts of CCA, but allowing an adequate extension of the UML metamodel seems to be most promising. In this way, semantic mismatches are avoided and all relevant component concepts are considered.

Web Application Extension (WAE). The WAE [36] is a light-weight extension which defines a set of stereotypes, tagged values, and constraints that enable us to model Web applications with UML. It mirrors the approach to model Web pages by classes or even components, while different types of Web pages are expressed by various stereotypes of these two basic model elements. The principal model element specific to Web applications is the **Web page**. It is a stereotype of **Component** which contains pages like **Server Page**, **Client Page**, **Form**, and **Frameset**, all being stereotypes of **Class**. Furthermore, special relationships between pages are defined such as **Link**, **Frame Content**, and **Submit**, all being stereotypes of **Association**. Moreover, new stereotypes of **Attribute** are defined, e.g. **Input Element** being used to input text to a Web form. All new stereotypes of **Class** and **Component** come along with new graphical notations to facilitate the reading of Web models.

Failures and Security. To model failures and security issues several profiles have been developed for UML. E.g. in [100], a UML profile is presented which models and assesses reliability. Failures are invented as stereotypes of events to report on faults. Faults are classified concerning the name, origin, and persistence, and cause error states of the system. Several UML extensions have been found to model security issues, mainly the role-based control access (RBCA) mechanism [101]. Users, roles, permissions, sessions, and constraints are introduced as new stereotypes to model an access mechanism where users get permissions to access certain resources dependent on their roles. The structure and behaviour of RBDA is then modelled by UML using these new stereotypes.

Evolution. UML extensions for evolution rely on profiles as well. In [52], new stereotypes for dependency relationships are introduced expressing evolving structures. Moreover, inference diagrams are introduced which show trans-

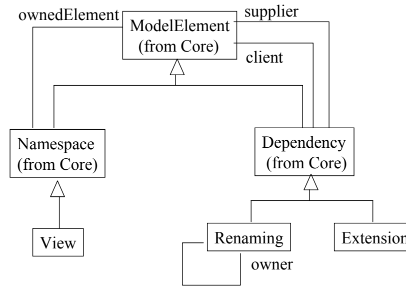

Figure 4: Views-related extension of the UML metamodel

formations of system structures such as class diagrams or some interaction diagram. In [77] evolution contracts are added as new model elements which are defined between name spaces and which can be further stereotyped to primitive contracts like addition, connection, removal and disconnection, and complex contracts.

Application-Specific Views. Views in UML are aspect-oriented and manifest themselves in a variety of diagram types.[2] In this way, views are completely independent from applications to be modelled by UML. But when for example, a team of application engineers with different skills and backgrounds is splitted into subgroups to specify only that aspect of a system which is later seen and used by a certain type of user, application-specific views should be definable. When integrating different views later on, name conflicts, i.e. the same name for different concepts and two different names for the same concept, can arise. Moreover, two views can overlap in their behaviour specification which has to be synchronized then. A detailed motivation as well as the semantic concepts of the view concepts are presented in [**4**]. Here, we concentrate our discussion on the abstract syntax of views in form of a UML metamodel extension. In chapter 5, we take up the view discussion again to combine view and component concepts semantically.

To adequately handle application-specific views the following extension of the UML metamodel is proposed: Views are introduced as new model elements. In the UML metamodel, **View** is supposed to be a stereotype of **Namespace** and can contain any kind of model element. Two kinds of relationships are basically for views, renamings and extensions. Thus, two stereotypes of **Dependency** are proposed, **Renaming** and **Extension** meaning that one view renames or extends another view, resp. See Figure 4 for this views-related extension of the UML metamodel. Views are supposed to have no own graphical notation. They would influence the graphical representation of a UML model within a tool using tree or form-like notations.

Visualization of OCL. Negotiation about component interfaces becomes a crucial issue in open distributed systems. Stating only the signature is often

[2]However, aspect-oriented views do not occur explicitly in the UML metamodel.

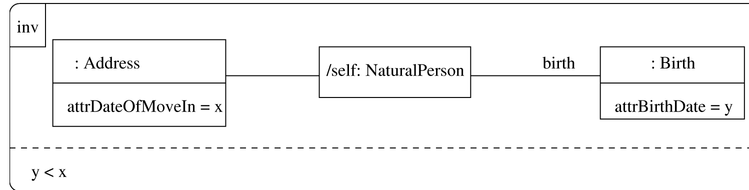

Figure 5: OCL constraint: The birth date comes before the date of moving into an apartment.

too loose, additional constraints in form of invariants or pre and postconditions for methods and operations help fixing important properties which have to be fulfilled by an interface. Originally, there was no such constraint language for UML. This lack has been realized and OCL, the Object Constraint Language [98] has been proposed for this purpose. Unfortunately, OCL is a textual language and does not look very attractive to software developers. Moreover, it does not fit well to UML and can be much better integrated relying on collaborations. In **[2]** and **[31]**, we visualize OCL on the basis of collaborations and integrate the UML and OCL metamodel to a larger extent than in the current OCL specification [20].

In **[31]**, a visualization of OCL is presented which departs from UML collaborations. The visualization of navigation paths in object structures uses that of classifier and association roles in collaboration diagrams. The visualization of navigation paths helps the developer to maintain an overview without forcing a reformulation of a visually given object structure in a completely different syntax, as done in OCL's textual notation. Thus, this visualization adapts the notation of UML core elements. An alternative proposal to visualize OCL are constraint diagrams [66] and their evolution, spider diagrams [65], which are not based on collaborations but extend Venn diagrams and Euler circles. However, this approach is not based on graphical elements already present in the UML core.

The following example show a sample constraint taken from the 'eGovernment' project in Berlin, Germany. The project objective is to replace the software used in the residents' registration offices in Berlin by a new one exploiting Internet technologies. Citizens and authorities shall have the possibility to perform typical business processes like the registration of inhabitants using web forms. Especially in this kind of applications the correctness of inputs plays an important role. A visual constraint language is meant to help software developers to easily formulate input constraints unambiguously. Figure 5 shows a visualized OCL constraint on object properties. The constraint is stated in natural language by the caption and given in textual OCL in the following.

```
context NaturalPerson inv:
self.birth.attrBirthDate < self.address.attrDateOfMoveIn
```

A number of visual shortcuts are introduced for pre-defined operations to support a compact notation of OCL constraints. OCL has built-in types being primitive like integer, boolean and strings, and collections like sequences, sets and bags.

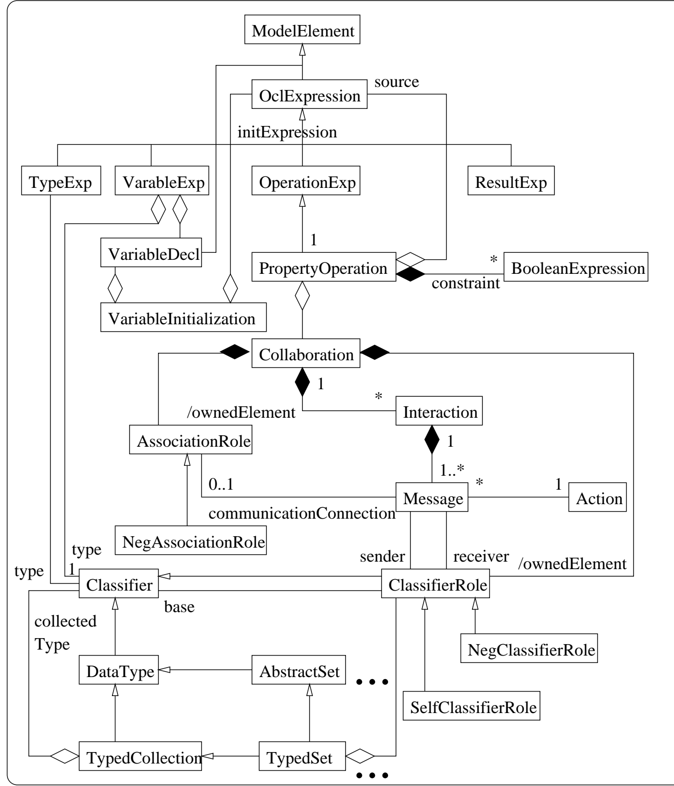

Figure 6: A metamodel for OCL based on collaborations

For textual OCL, a metamodel has been developed building up on the metamodel for UML. This OCL metamodel can be adapted also to the visual OCL. Thus, the visual and textual notations for OCL can be used in parallel. Nevertheless, the visualization based on collaborations shows that the two metamodels for UML and OCL can be even further integrated, where property operation expressions are described by collaborations. Figure 6 contains the main model elements of an OCL metamodel which is based on collaborations. In addition, two new steoretypes of **ClassifierRole** and **AssociationRole** are proposed, namely **NegClassifierRole** and **NegAssociationRole**. They can be used to express the non-existence of certain collaboration parts.

Semantics. Due to its status as quasi standard modelling language for software systems, an enormous number of extensions have been elaborated for UML to make it well-suited for system modelling. In this article, we focus on promising extensions for distributed system modelling. As we have seen, all main aspects of distributed systems can be modelled. However for validation, UML does not yet provide a formal basis. The UML metamodel offers a semi-formal approach which concentrates on the definition of the abstract syntax of UML. Hundreds of researchers have proposed formal semantics relying on various formalisms, mostly for restricted parts of UML. A kernel of those researchers is organized in the "precise UML (pUML)" group [22]. One main aim of this group is to define a standard semantics for UML which is precise. In Section 5 we discuss

how distributed graph transformation can serve as a precise semantic domain for UML focussing on distributed system issues.

3.2 Visual Language Definition and Parsing

The abstract syntax of the UML is defined by the metamodel being a class diagram which describes the model elements and their relationships being available in principle. Moreover, OCL is used to formulate well-formedness rules on the abstract syntax structure, i.e. to restrict the allowed structures to meaningful ones. Consequently, the abstract syntax of a concrete UML model is an instance of the metamodel, i.e. a diagram containing model element instances. The concrete syntax of the UML is given by a document called "UML notation" describing all concepts in natural language and showing their diagrammatic representations. The interrelation between concrete and abstract syntax is left vague.

Visual languages like the UML are considered with more rigor in visual language (VL) theory. A number of questions and problems arise with visual languages which are well answered for textual languages, but not for VLs. While visual languages are used for many different tasks, it is still not clear how to define the term "visual language". Surely, visual languages have to possess a multi-dimensional representation of language elements. Diagrams and graphics, so-called *visual sentences*, consist of a set of visual symbols such as graphic primitives, icons or pictures, and a set of relations such as 'above' or 'inside'. There are textual as well as visual notions for the abstract syntax of visual sentences. In textual notations, multi-dimensional representations have to be coded into one-dimensional strings which is not convenient. A visual notation such as UML metamodel instances being a sort of graph, looks much more natural. Information concerning concrete representation is often added by attributes.

Visual languages can formally be defined by a grammar-based approach, e.g. picture layout grammars, constraint multiset grammars, and positional grammars, or by some kind of constraints and logics. The UML metamodel belongs to the constraint-based approaches. A standard formalism as the Backus-Naur-Form for textual languages, is missing to define VL syntax and semantics. Nevertheless, the theory of visual languages [75], especially concerning a language hierarchy and parsing algorithms, is very much oriented at its pendant for textual languages. A main subclass of VLs, i.e. context-free VLs, is distinguished which allows more efficient parsing.

While constraint-based formalisms provide a declarative approach to VL definition, grammars are more constructive, i.e. closer to the implementation. For the special case of OCL constraints, in **[2]** we presented how they can be translated to a certain graph grammar approach to support constructive constraint checking. Due to its appealing visual form, graph grammars can also directly be used as high-level and visual meta modelling mechanism. Considering visual sentences as graphs, a graph grammar defines the language grammar. Its graph language determines then the whole visual language. In **[3]**, we comprehensively present the application of graph transformation to visual language definition and parsing.

In the following, we consider the graph transformation approach to VL definition more closely and discuss its main properties. But first, we present a short introduction to graph transformation in general.

Graph Transformation. Graph transformation is a computational paradigm which combines the potentials and advantages of both, *graphs* and *rules*. Graphs are a well-known means to represent any kind of structure like system states, object structures, computer networks, entity-relationship diagrams, flow charts, etc. Rules are a clear and abstract concept to describe structure changes in general. Computations in logic or functional programming, language definitions, term rewriting, and concurrent processes are prominent example for the use of rules. Graph transformation combines both concepts, graphs and rules, providing the rule-based manipulation of graphs. [88] gives an overview on all the main approaches to graph transformation. They are formalized on the basis of set theory, algebra, logics, and category theory. The articles in [45, 46] present a comprehensive collection of graph transformation applications, particularly to concurrent and distributed system specification. In **[26]**, an introduction to graph transformation is given, independently of a particular graph transformation approach, i.e. independently of a certain kind of graphs and rules and a specific way of rule application. Hence, this article is particularly interesting for readers who want to know how graph transformation works and how it is applied in principle. Moreover, the concept of *transformation units* has been introduced in this article. It offers a structuring principle for building large graph transformation specifications out of smaller ones. Transformation units encapsulate mainly rules with their control conditions. Semantically, the application of rules within one transformation unit may be interleaved with other transformation units used.

Throughout this article, graph transformation will occur again and again to serve for different purposes. Especially as meta modelling paradigm, graph transformation is promising. We will use it for visual language definition and parsing as well as for translating a visual language into some semantic domain to enable model validation (described in Section 4.3). Even as semantic domain itself graph transformation is well suited, compare Section 5.

Defining the Syntax of Visual Languages by Graph Grammars. Sentences of visual languages may be regarded as assemblies of pictorial objects with spatial relationships, i.e. their structures are a kind of graph. We distinguish *spatial relationship graphs* (SRG) which describe the structure of a visual sentence seen as a picture, and a more abstract graph, the so-called *abstract syntax graph* (ASG). An ASG provides the information about the syntax of a visual sentence in a more succinct form. It is oriented towards the interpretation or compilation of the regarded sentence. The distinction between SRGs and ASGs - as introduced in [86] - has been inspired by the Model-View-Controller concept of Smalltalk and the traditional distinction between abstract and concrete syntax of textual languages.

As both, SRGs and ASGs are graphs, *graph grammars* are a natural means

VL concept	graph grammars	UML metamodel
visual sentence (concrete syntax)	SRG	–
visual sentence (abstract syntax)	ASG	metamodel instance
interrelation (concrete - abstract)	graph translation	–
abstract VL definition	graph grammar	UML metamodel

Table 1: Comparison of graph grammar and UML metamodel concepts

for defining the concrete and abstract syntax of visual languages. In contrast to other approaches to VL definition, graph grammar approaches use nodes and edges to describe objects and any kind of their interrelationships. Additional attributes are used only to describe further information, i.e. layout of visual objects and relations as well as their semantics. In [**3**] two different approaches to VL definition are presented: DiaGen, relying on hypergraph grammars, and GenGED which is based on attributed graph transformation. In both approaches, the graphs describe the graphic's structures while the concrete layout is specified by graphical constraints or graph algorithms. Graph grammars are used to define the translation from SRG to ASG (low-level parsing) as well as high-level parsing on the ASG solving the membership problem. A concrete parsing approach based on graph transformation is given in [**33**]. Following this approach, it is possible to formally define the concrete and the abstract syntax of a VL as well as their interrelation (by graph translation). In Table 1, a comparison of the graph grammar and UML metamodel concepts for VL definition is presented. For some VLs it happens that the structures of SRGs and ASGs are very similar, as e.g. for class diagrams. In this case, it is more convenient to work with only one graph structure which might differ in the sets of attributes for concrete and abstract syntax definition. Parts of UML have been defined in DiaGen as well as in GenGED, i.e. class diagrams [27, 78], statecharts [28, 78], etc. showing that the graph grammar approach is promising to follow.

Following a graph grammar approach to VL definition like in DiaGen and GenGED, the corresponding tools comprise general parsing algorithms to check the membership problem for a concrete VL sentence. Changing the VL graph grammar, automatically adjusts the parsing process for the changed VL. Moreover, theoretical results concerning e.g. independence of graph rule applications can profitably be applied to efficiently parsing context-free as well as less restricted languages [**3**],[**33**].

4 Model Validation

One general requirement to a modelling technique for distributed systems is a clear formal background to have the possibility to reason about interesting properties. For example, *fault tolerance* and *trace properties* like deadlocks and termination, play an important role in distributed systems. The most interesting validation issue for distributed systems has been *performance*, since one of the main motivations to build distributed systems has been the sharing of

resources. Today resource sharing is taken for granted and good performance analysers have been developed. But data sharing is still a challenge and leads to new kinds of problems. The possibility of concurrent updates can easily lead to inconsistencies. On the other hand, important data should be replicated to guarantee a fast access. But data replication leads also to problems of inconsistency which have to be solved adequately. *Consistency issues* are less well investigated in the literature. Adequate techniques to guarantee consistency are a central part of the author's contributions concerning model validation. Syntactic consistency checks for UML are considered in Section 4.2, while semantic consistency is discussed in Section 5.

UML is a semi-formal modelling language: The UML metamodel defines the abstract syntax of the UML by providing a class diagram containing all model elements and their interrelations as well as well-formedness rules to restrict the set of allowed models. Consistencies in aspect-oriented views and inter-viewpoint consistency conditions are mainly captured by those well-formedness rules. Distribution or domain-specific consistencies can be formulated using OCL, but the UML standard [95] describes its semantics quite vaguely in natural language. There are various approaches to define a precise semantics for UML or some of its sublanguages. In general, the semantic domain chosen depends heavily on the validation issues which are taken into account. This is natural, since the semantics can be more abstract when concentrating on the validation of specific properties. This means in particular that the semantic domain can be changed according to the property to be analysed. In Section 4.2, selected validation techniques for UML are mentioned putting the main emphasis on distribution issues.

As pointed out there is an increasing need for model translation into semantic domains. In the literature a number of approaches can be found following a grammar or logic-based approach such as [40], [48], **[60]**, [25]. A promising approach for automatic model translation is given by graph transformation. Abstract syntax graphs are translated to semantic graphs or expressions by applying graph transformation rules. In contrast to other approaches, translation rules can be completely visually developed. Since the author is partly concerned with the development of such translation concepts, this approach is presented more closely in Section 4.3.

4.1 Validation Issues

Performance and Quality of Service. Performance analysis has been one of the primary interests when modelling distributed systems. Performance issues arise from limited processing and communication capacities of computers and networks. They comprise responsiveness, *throughput* and *balancing of computational loads*. Using e.g. a remote service, a fast and consistent response is required. But the response time is dependent of many factors, i.e. the load and performance of any server needed and of the network as well as any delay caused by some software component. Especially when considering Web services, fast responses can be better achieved when intelligent caching mechanisms are used.

A traditional measure of performance for computer systems is the through-

put - the rate at which computational work is done. In distributed systems, throughput is measured by the processing speeds at clients and servers as well as by data transfer rates. Discovering that the throughput differs heavily for different users, tasks or hosts, the computational loads have to be balanced. Load balancing exploits the available resources better and thus, enables a better service. E.g. a heavily used Web service such as a search engine, has to be hosted by several computers to provide an acceptable service.

The notion 'Quality of Service' (QoS) has been formed to refer to the ability of a system to meet time limits, especially when transmitting real-time multimedia data. But QoS comprises reliability, security and performance as well. Recently, adaptability has been added, meaning the ability to meet changing environments which has recently been recognized as a further important aspect of service quality. E.g. in mobile computing, load balancing mechanisms have to be much more flexible than in traditional intranets to meet fast changing requirements.

Consistency. Distributed systems are modelled by different views using UML as well as extensions such as application-specific views. Each view concentrates on a subset of system aspects. Especially when distributed teams are developing their own system views, inconsistencies between different views, i.e. *Inter-ViewPoint inconsistencies* [49], can easily occur. But also inside of one view, so-called *In-ViewPoint inconsistencies* have to be taken into account.

Data consistency is mainly application-specific, but there are also distribution-specific issues. A typical technique for performance enhancement is replication of hardware and software resource, including *data replication*. Moreover, consistent data replication is the key aspect when considering a distributed shared memory model. The degree of consistency varies widely and depends heavily on the use of data replication. E.g. authentication data has to be always consistent to achieve a high security level, especially in the area of Web applications such as E-commerce applications. Here, each data change has to be immediately propagated to all replication servers, before subsequent actions are performed. On the other hand in mobile computing, changes on location data are not supposed to be handled promptly, since discovery services are also allowed to respond suboptimal to restrict communication traffic.

Several notions of consistency have been formed (compare [41]): *Linearizable processes* on replicated data meet the correctness criteria and real times of executions as corresponding processes on non-replicated data. Since distributed systems cannot always synchronize clocks, the accuracy of linearizable processes is difficult to meet. A weaker correctness condition is *sequential consistency* which captures the requirement on the same order of action processing instead, without appealing to real time. Since sequentially consistent data structures are costly to implement, even weaker forms of consistency have been developed. One of them is *coherence* which means that processes agree on the order of data manipulations to the same part of data, but not necessarily agree on the order of data manipulations to different data parts. The weakest form is *weak consistency* of data which is possible if the knowledge of synchronization is

exploited. This model relaxes consistency temporarily, while appearing to the user to implement sequential consistency. E.g. when data is locked, it can be inconsistent as long as the lock exists, since no other user can access it. All replications of a locked data part have to become consistent, before the lock is released.

Typically for distributed systems are distributed data structures which are accessed concurrently. Shared data is usually accessed by distributed transactions implementing a weak consistency.

Trace Properties and Fault Tolerance. Basic trace properties are concerned with *action conflicts* and *dependencies*. Two actions are in conflict if the execution of one action prohibits the execution of the other one. On the other hand, an action is dependent on another one if it cannot be executed without the other one being executed first.

In general, trace properties are described by sets of action sequences, distinguished in *safety* and *liveness properties*. A safety property asserts that nothing bad happens during execution, i.e. the process never reaches a bad state. While a liveness property asserts that something good will eventually happen, i.e. the process eventually reaches a good state.

For sequential processes, the most important safety property is correctness (i.e. the process' final state is correct); whilst the most important liveness property is termination (i.e. the process eventually terminates). However, with distributed systems, we are often dealing with systems that may partially fail and may not be supposed to terminate. Furtheron, there are many more safety and liveness properties of interest.

Additional important examples of safety properties are *mutual exclusion* and the *absence of deadlocks and livelocks*. Mutual exclusion is of great interest when resources are shared. No matter if the resource is a printer or some data which should be changed, there should be at most one process accessing the resource at a time. When coordinating mutual exclusion and also during other activities, process executions may come into a deadlock meaning that none of the processes can continue its execution, or into a livelock where processes do continue their executions but without really getting their work done.

An additional important kind of liveness properties are progress properties. They allow us to assert that, no matter what state the system is in, it will always be the case that a specified action will eventually occur. Progress properties are essential, considering e.g. mutual exclusion, processes should be coordinated such that a process accesses the shared resource finally.

Trace properties are most interesting when process or communication failures can occur. Even if some process or communication failed, the whole system is supposed to continue correctly. In this case, it is called *fault tolerant*.

4.2 Validation Techniques for UML

After introducing all the validation properties which play an important role in distributed system validation, we discuss now selected validation techniques developed for UML. There exist various approaches to validate UML models, in

the literature. We selected those approaches being most promising to support validation of distributed system properties.

Abstract Syntax Checks. Consistency problems are two-fold: While *syntactic consistency* ensures that a UML model is correct wrt. the metamodel, *semantic consistency* presumes the model translation into a semantic domain. The consistency condition has to be formulated in the same domain to be checkable. Semantic consistency will be considered in the next paragraph.

Syntactic *language-specific consistencies* are formulated as well-formedness rules which are OCL constraints on metamodel instances. For each constraint a context has to be defined being a specific model element. In [**2**], we showed that these language-specific constraints can be constructively checked by translating them first to graph rules and graph transformation units (see Section 3.2), resp. Then, the language consistency of a UML model is checked by testing the applicability of the resulting rules and transformation units to the abstract syntax graph given showing an instance of the UML metamodel. If all rules and transformation units are applicable to all instances of their context model elements, a UML model is consistent. Otherwise, the model element instance to which a specific rule or transformation unit is non-applicable, is reported to be inconsistent to the corresponding OCL constraint. Another approach towards a formal semantics for OCL is presented in [87]. This work relies on algebraic specifications and is under discussion to define the standard semantics of OCL. Moreover, it comes up with a nice tool, the USE tool, supporting the constraint checking of UML models. Unfortunately, the OCL metamodel proposed in [87] is only loosely coupled with the UML metamodel and can be much better integrated using collaborations (see [**31**]).

A promising approach where distributed graph transformation is used to formulate *In- and Inter-ViewPoint consistency* checks is presented in [**54**]. Again consistency checking is performed by applying graph rules to abstract syntax graphs. But here, each view is modelled separately and Inter-ViewPoint checks result in synchronized graph transformations on several interrelated abstract syntax graphs. Another approach, *xlinkit* [79], where the consistency of distributed specifications can be checked automatically, is based on XML technologies.

Semantic Models. Hundreds of approaches exist to give UML a formal semantics. On the one hand, we can gain insight into the precise meaning of UML notations and diagrams. On the other hand, a formal semantics is a necessary prerequisite to check important properties of a UML model. Especially algebraic specification techniques have been chosen to build up a formal reference model for UML, e.g. [55, 64]. It is the declared aim of the precise UML group to come up with a precise standard semantics of the whole language UML and to use it for verification purposes then. Up to now, there are various approaches to formalize each a certain aspect of UML by some formal technique and to use this precision for formal reasoning. Especially behaviour diagrams have been considered and translated to process algebras [48, 51], abstract state

machines [56] or labelled transition systems (e.g. UMLAUT [63] and vUML [70]) for model checking purposes. UML behaviour diagrams are also used to translate to stochastic and timed Petri nets and processes showing quality of service [71, 72].

Mostly, the provided formalizations comprise only a subset of model aspects, enough to show specific properties. The formal semantics is stated, but a rigorous translation from the abstract syntax to the semantic domain is often missing. If this is the case, nothing can be said about the correctness of a given translation. Considering an automatic translation, we can at least reason about its functional behaviour. In the following subsection, we argue that graph transformation is well suited to serve for this purpose.

4.3 Model Translation by Graph Transformation

Model translation by graph transformation is a natural and rather popular approach, done by a number of researchers [48, 30, 42, 96] and **[60]**. Here, graph transformation is applied on a meta level taking an abstract syntax graph as input and producing a semantic domain model as output. The translation approaches vary in input and output formats dependent on the visual language and its semantic domain chosen. Moreover, different approaches to graph transformation are used dependent on concepts and theoretical results needed as well as personal preferences of the researchers. All the translation approaches referred to above, are illustrated and proven at concrete translation scenarios. The approach by Baresi and Pezze [30] has been examined at several kinds of diagrams, i.e. Structured Analysis, IEC Function Block Diagrams, a subset of UML, Control Nets and LEMMA, a language for medical model analysis. The semantic domain for all these diagram types is always the same: High-Level Timed Petri Nets. Lara and Vangheluwe report on the translation of Non-deterministic Finite Automata to deterministic ones in [42] and statecharts to some sort of Petri nets in [43]. Varro presents the translation of Message Sequence Charts to a partial order of events in [96] and refers to the translation of UML Statecharts to Extended Hierarchical Automata.

In [48] and **[60]**, the algebraic graph transformation approach is chosen. It is illustrated by a subset of UML state diagrams that is translated to CSP. We will consider this approach more closely in the following.

In all related approaches some formal definition of graph transformation is used to define the translation rules, but none of them exploits the theory behind. Often, some informal or semi-formal language is translated into a formal domain, giving a formal semantics to a language in that way. It is useless to reason about correctness of such a VL translation, in the sense of model equivalence for these VLs. However, it makes sense to ask for functional behaviour of a model translation, i.e. to show confluence and termination of translations in any case. Otherwise, repeated model translation might lead to different results, or model translation might not give any result, since it is not terminating. In **[60]**, we consider the critical pair analysis to show confluence of attributed graph transformation systems and apply it to the translation of UML Statecharts to CSP. Critical pair analysis computes all minimal conflicting situations, minimal

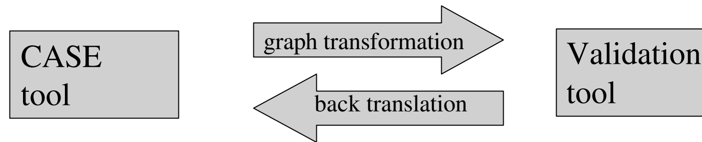

Figure 7: Model translation for validation purposes

in the sense that there is no unnecessary context. If we can show that all critical pairs can be joined, i.e. further rule applications lead to a common result graph again, the graph transformation system is confluent.

Model translation is useful mainly for two reasons: validation of models and code generation. Considering model translation for validation purposes, first a model has to be created by some CASE tool. At a certain point the model developer might ask for important model properties going beyond pure syntax checks. This is the starting point for translating the model to some semantic domain dependent on the properties to be examined. After translation, a (separate) validation tool is started and gets the translated model as input. The validation result produced has to be translated back to be useful for the developer. This general procedure is depicted in Figure 7. Code generation is a similar procedure, but easier to perform, since back translation is not necessary.

To support a wide range of model translations, common exchange formats for CASE tools, graph transformation engines and validation tools are needed. For CASE tools, especially UML CASE tools, the XML-based format XMI (XML Metadata Interchange) [24] has become a quasi-standard. For graph transformation engines and other graph-based tools, an initiative has been started to develop a common exchange format for graphs, called GXL [99], also based on XML. Validation tools do not yet have a common exchange format, but that has been taken under consideration, e.g. in the ETI initiative for an electronic tool integration platform [90] (`http://www.eti-service.org`).

Common Exchange Formats for Graph-Based Tools. Since graphs are a very general data structure used in various fields in computer science, also a variety of graph-based tools exist. To increase their interoperability, GXL, a common exchange format for graphs, has been developed on the basis of XML [99], **[92]**. GXL allows to store the logical structure of nearly all kinds of graphs, e.g. typed and attributed graphs, with hyperedges, and also hierarchical graphs. The layout of graphs can be stored by e.g. SVG [21], an XML format for scalable vector graphics.

Discussing the tool support for model transformation by graph transformation, GXL will play a key role. It can be considered as the central format when translating a model. Assuming a CASE tool producing XMI output, there has to be some XML Stylesheet (XSL) transformation [16] to produce GXL input for the model translator. The XSL transformation from any kind of XML format to GXL can be straight forward, since the underlying structure is a tree. In this sense, tags are interpreted as nodes and subtags are translated to nodes connected to their corresponding parent node. Interpreting tag references also as edges leads to a proper graph structure. But considering formats like XMI,

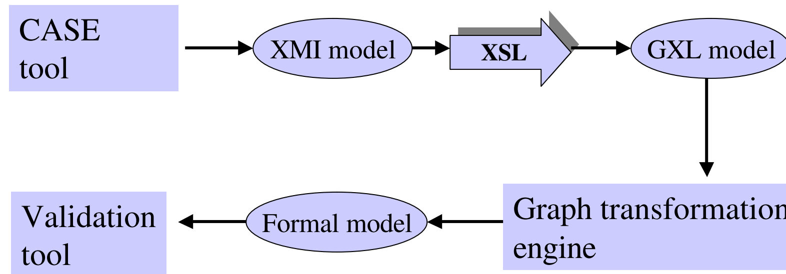

Figure 8: Refined model translation using XML

they are too verbose to be transformed in this simple way. The stylesheet has additionally to compress the information adequately. After model translation, a formal model in the input format of the corresponding validation tool should have been produced which can be used to compute the actual validation. This validation result has to be back translated into an input format of the CASE tool. How this back translation looks like is very much dependent on the kind of result. The whole tool chain is illustrated in Figure 8.

On top of GXL, the core exchange format for graph transformation systems, GTXL [**92**], has been developed for the interchange of graph transformation systems. Discussing graph transformation as a semantic model domain, as in the next section, GTXL is supposed to be used as target format for model translations and thus, as input format for graph transformation tools performing validations.

5 Graph Transformation as Validation Domain

On the one hand, graphs are a very general means to formalize any kind of system structures, on the other hand, they have a clear visualization and thus, are well suited to intuitively present system structures. Using graph transformation to describe a system model, graphs are usually taken to describe static system structures such as component and object structures as well as deployments. System behaviour expressed by state changes is modelled by rule-based graph manipulations, i.e. graph transformation. The rules describe pre and postconditions of single transformation steps. The order of steps describes the temporal or causal dependency of actions. The graph transformation paradigm is formal and intuitive at the same time, thus an attractive framework to reason about system properties. In Section 5.1, additional structuring concepts are discussed for graph transformation yielding a comprehensive formalism well designed to define all important system concepts and properties (see Section 5.2). Several validation techniques have been developed for graph transformation which can be applied to a formal semantic model to show conflicts and dependencies of actions as well as structural consistencies (Section 5.3).

5.1 Structured Graphs and Graph Transformation

For modelling complex systems such as distributed systems usually are, a variety of structuring principles has been developed within the graph transformation paradigm. Parallel and distributed graph transformation both structure rule applications, in temporal as well as spatial dimensions. Distributed graphs contain an additional structure on graphs. On top of these basic structuring concepts for rules and graphs, component concepts define the partitioning of graph transformation systems into subsystems. Last but not least, views are helpful to structure a system according to different aspects and localities.

Parallelism and Distribution. An excellent introduction to the double-pushout (DPO) approach to graph transformation is given in [39], worthwhile to get a comprehensive overview on all important results concerning this approach. Introducing the basic concepts of graphs and rule-based transformation of graphs, *graph transformation systems* (GTSs) are defined by tuples of a type graph, a start graph, and a set of rules. In the basic form of graph transformation, each rule is applied sequentially. Considering parallel transformation, there are different approaches. If we stick to sequential execution, *parallel transformations* have to be modelled by interleaving arbitrarily their atomic actions. This interleaving leads to the same result, if the atomic actions are independent of each other. Simultaneous execution of actions can be modelled, if a parallel rule is composed from the actions. Atomic actions which are not independent of each other, but synchronize in a common subaction, can be composed to a so-called amalgamated rule which models the concurrent execution of the atomic actions where their subaction is executed only once. Amalgamated transformations generalize parallel transformations by such an additional synchronization mechanism. In [39], parallel graph transformation steps are restricted to two rule applications in parallel. In **[91]**, the author extended the theory to an arbitrary number of parallel rule applications and showed that they can be amalgamated to one rule application leading back to a basic graph transformation step.

Moreover, *distributed graph transformation* is discussed in [39] following a simple approach. Graphs are allowed to be split into local graphs and after local transformations, local graphs are joined again to one global graph. In **[5]**, this theory is extended by the author in several directions: First, in [39] the splitting into two graphs together with a common interface graph is considered only. In **[5]**, more general topologies are allowed, actually all kinds of network structures are possible. The network graph is the abstract part of a distributed graph which is a structured graph on two levels. Each network node is refined by a local graph, while each network edge is refined by a local graph morphism. Formally, a distributed graph is a diagram in the category **GRAPH**. Second, splitting and joining of graphs is not needed to change the distribution structure. Network transformations which are usual graph transformations on the network level, can be used for arbitrary network changes. Third, complex synchronizations between an arbitrary number of local graphs can be described and fourth, local transformations can be parallel transformations.

Parallel graph transformation as in **[91]** is a special case of distributed graph

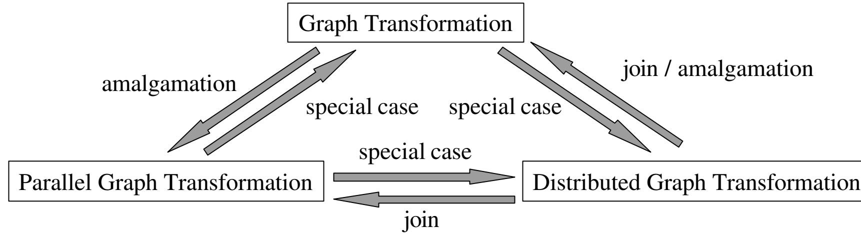

Figure 9: Interrelation between graph transformation concepts

transformation where the host graph is non-distributed. The other way round, the distributed host graph can be joined to one graph leading to a parallel transformation. If additionally, the distributed rule and match are amalgamated to one rule and one match we end up at a simple graph transformation step again. If we want to keep the distribution structure, each distributed graph can be considered as a simple graph if its network graph and its refinement are flattened to simple graph parts of special types. Similarly, distributed graph rules can be flattened to usual graph rules. This kind of reducing distributed graph transformation to non-distributed graph transformation is interesting when applying validation techniques for graph transformation not directly available for the distributed case. Vice versa, it is obvious that simple graph transformation is a special case of distributed transformation where graphs and rules are non-distributed. All graph transformation interrelations are summarized in Figure 9.

In [**6**], the theory of distributed graph transformation has been extended by *attributes* at nodes and edges as well as *application conditions* for distributed rules. These two extensions are necessary to provide a sound formal basis for reasoning on distributed systems.

Components. During the last years, a variety of structuring concepts for graph transformation systems has been developed. An overview and classification of most of these concepts is given in [**62**]. Structuring concepts for graph transformation systems differ heavily in features and notions. They are called *modules*, *packages* or *components* and all support some kind of information hiding. At least, an export interface is declared which provides some structural and behavioural information about the body which hides the remaining parts. Connections between such entities are realized by using the export interface of other entities to implement the own import interface.

In the following, a component concept for graph transformation systems is discussed which follows a distributed semantics for components. Semantically, components are active processes and their semantics is restricted with each new interconnection to other components. The component concept, presented in [**1**] has its origin in the DIEGO approach presented in [**93**] and relies directly on distributed graph transformation.

A *graph transformation system (GTS) component* C consists of one *body GTS* B, a set of *import GTS* $\bigcup_{1<i\leq n} I_i$ as well as a set of *export GTS* $\bigcup_{1<j\leq m} E_j$, each with an *embedding morphism* into B. Embedding morphisms relate the

type and start graphs as well as the rules. Note that not all body rules must have a correspondence in each import/export GTS.

For the *composition of GTS components*, embedding morphisms are added between import and export GTS leading to an import/export relation. Such an embedding morphism is a usual GTS morphism. All import GTSs form the import of the composed GTS, all export GTSs are still export interfaces. Component rules related by embedding morphisms are applied simultaneously. The composition details are described in [**50**] where GTS components together with their connections to other component interfaces define *local views.*

Views. Two view concepts have been developed for graph transformation: the *type-oriented* and the *instance-oriented* approach. The type-oriented approach is followed in [**4**] to describe a system according to different aspects. The approach is based on typed graph transformation which allows to describe a set of graphs by a type graph. According to its types, each view specifies only partially the system's states and behaviour. It may happen that an action performed in one view has to be concurrently coupled with actions of other views to ensure a consistent state manipulation. Thus, each view specifies what at least has to happen on a system's state, i.e. its semantics is loose. Actions in other views may be needed to complete the whole system's behaviour. If several views are developed independently of each other, inconsistencies, e.g. concerning names, might occur which have to be solved before the view can be integrated. In [**4**], an automatic integration algorithm is applied presuming a common reference model which underlies all views and specifies correspondences between model elements of different views. Two prerequisites are needed for automatic view integration: It must be possible to rename views such that different names can be used for the same concepts in different views. Furthermore, the extension of views has to be defined componentwise for types and actions such that type sets can be extended and a subaction relation can be established.

The instance-oriented approach to views has been considered in [**6**] and [**50**] where different local systems are distinguished. Each local system is described by a GTS component which defines a local view. The body, all import and export interfaces, as well as all remote interfaces connected are comprised in one local view. Thus, local views overlap in interfaces and their interrelations. Each local view shows only that system part described by its component. Considering a local action, actions of other components might be needed to model a synchronized system behaviour. The synchronization is done by amalgamating local view rules by common interface rules. For each synchronization, local view rules are synchronized as much as possible such that the resulting amalgamated rule does not have unbound interface rules. During the lifetime of a system, new local views can show up or old local views can vanish if corresponding components are created or deleted. Since local views directly rely on distributed GTS components, they also have a formal semantics based on distributed graph transformation.

Both kinds of views are orthogonal to each other and can be easily combined considering aspect-oriented views on distributed systems. In this case, the

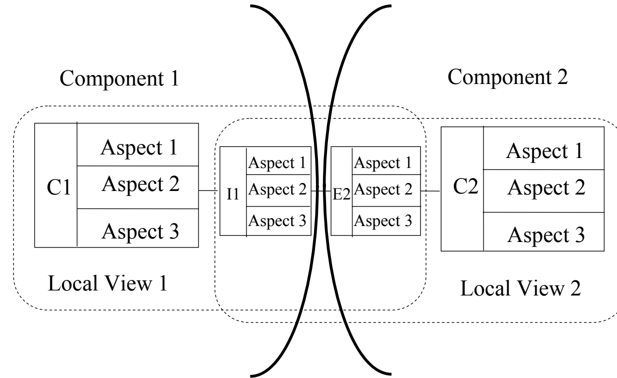

Figure 10: Schematic example of combined view concepts

aspect-oriented view concepts presented in [**4**] have to be based on distributed graph transformation which defines a straight forward extension. In this way, each part of a local view, i.e. the body, import and export interfaces, can be presented aspect-oriented. For a schematic example of this view combination see Figure 10.

5.2 Distributed Graph Transformation as Semantic Domain

In the following, we discuss all key concepts of distributed object systems and show how they can be mapped to corresponding graph transformation concepts. We also draw connections to corresponding model elements in UML. This discussion shows that graph transformation, and moreover, distributed graph transformation seems to be an expressive and adequate formal framework for visual modelling of distributed object systems, powerful enough to give a semantics to all key aspects with adequate abstraction. It also yields a concise basis for the comparison with other semantic reference models such as [55]. However, the priority of a complete and precise mapping of a VL syntax such as the UML metamodel, to a semantic domain is very dependent on the validation techniques available in that domain. In Section 5.3, we consider validation techniques based on graph transformation which need only partial model translations.

Intuitive and Formal Semantics. Graph transformation is one of the few precise formalisms which come up with an intuitive appealing graphical representation of structures. States and state changes can be visualized and thus provide an easy access to the semantics discussion. Type- and instance-oriented view concepts which are available for graph transformation allow to concentrate on certain system parts and concerns at a time.

Abstraction Levels. Two abstraction levels are offered by distributed graph transformation to concentrate on the overall system architecture first and refine to the object level then. The system architecture and its reconfiguration is

described on the more abstract network level. Network nodes are distinguished into component and interface nodes containing a refinement of the corresponding system part each. Each refinement belonging to the object level, models some kind of object structures and their manipulation. This abstraction hierarchy is clearly a structure-oriented one which suits well to the UML. It can be complemented by behaviour refinement as defined by transformation units [**26**]. A combination of transformation units with distributed graph transformation is discussed in [**32**]. It could be advantageously used to formalize UML use cases and their main interdependencies being subuse case relations. In [**58**], we discuss the refinement of use cases by activity diagrams and their formalization by graph rules.

Components and Reconfiguration. GTS components have been developed as the central concept to describe subsystems of (distributed) systems. A GTS component consists of a body GTS and two sets of interface GTSs which define views on the body GTS. Considering CCA being part of the UML profile for EDOC (see Subsection 3.1) processing components would be modelled by GTS components where the interface GTSs provide the connection points between components, i.e. they play the role of ports. Mapping ports to interface GTSs allows ports to have a semantic state as well. Simple connections between components can be mapped to embedding morphisms between import and export GTSs. More complex connections defining whole protocols with different roles, have to be mapped to separate GTS components semantically.

CCA distinguishes components on the type and on the instance level. Component instances, so-called component usages, are mapped to a template of distributed graphs, so-called component graphs, semantically, each describing a component usage in a certain state. Compositions can be considered as a template of distributed graphs each being glued from single component graphs which overlap in interfaces in the same state.

Component reconfigurations, in UML described by collaborations, can be translated into network rules. Restricting to simple collaborations containing only one action each, the translation can be done as follows: The left-hand rule side contains all those parts of a reconfiguration not annotated by $\{new\}$ and the right-hand side all those not annotated by $\{destroyed\}$. The partial graph morphism between the left and right-hand side relates that part which is neither $\{destroyed\}$ nor $\{new\}$. More complex reconfigurations have to be modelled by several network rules applied in a certain order of control. Using rules for reconfiguration, concurrency issues are well addressed, i.e. conflicts and dependencies can be naturally expressed. As a special form of reconfiguration, scaling can be modelled by applying creating network rules. In this way, a new basis for performance tests is laid using the graph transformation paradigm. In the case of real-time constraints, time attributes have to be added to certain node types as done in [57].

Views. Two kinds of view concepts have been distinguished for graph transformation systems: type-oriented views to concentrate on certain aspects and

instance-oriented views for local views of components. Type-oriented views are very common for visual languages. They are usually expressed by different kinds of diagrams, e.g. UML has twelve kinds of diagrams to model software systems from different perspectives. Additionally, application-specific views as presented in Section 3.1, can semantically be mapped to type-oriented views. In the UML model, all different views are integrated in one model by gluing them at common model element instances.

Instance-oriented views play an important role when approaching the system's implementation. Concrete instances of components, packages, classes, and objects are distributed to certain network nodes and interconnected. UML offers deployment diagrams to model this kind of distribution.

Classes and Objects. The abstract syntax of a class diagram can be considered as graph structure. It describes the types of objects and links occurring in the object system. Therefore, we map class diagrams to type graphs. Classes are mapped to nodes while associations (except association classes) are mapped to edges in the type graph. An association class can be represented by a node together with arcs to all class nodes where the association relates to or an hyperedge with tentacles to all those class nodes. A generalization cannot directly be mapped but its inheritance hierarchy has to be flattened first, i.e. associations to a parent class and its features have to be prolonged to its child classes. Class nodes may be further attributed, where each attribute is specified by a type, a name and a possible default value. Multiplicities at associations and further constraints result in additional graph constraints which have to be checked on each instance graph being type graph compatible.

Object structures are expressed by templates of instance graphs, each representing the object structure in a certain state. An object structure is consistent with its class structure if all corresponding instance graphs are type compatible with the corresponding type graph and satisfy the additional graph constraints.

Object Interaction. There are various kinds of behaviour models for (distributed) systems. UML alone has five types of diagrams to describe different aspects of behaviour. In UML a certain subset of common behaviour concepts is distinguished which forms a basis to reason about behaviour. First of all, behaviour is considered on the instance level, based on objects and links. Correspondingly, instance graphs are used semantically. In the graph transformation paradigm, the only concept to express dynamics are rules which formulate the preconditions and the effects of an action. Dependent on certain events which can be considered as triggers for rule application, state transitions are performed by graph transformations.

Object collaborations can be formalized similarly to component reconfigurations. Collaborations restricted to one action are mapped to one rule where the left-hand side of a rule contains all those parts of a collaboration not annotated by {*new*} and not being the target of a <<becomes>>-relation. The right-hand side contains all those not annotated by {*destroyed*} and not being the source of a <<becomes>>-relation. The partial graph morphism between left

and right-hand side relates both copies of that part which is neither $\{destroyed\}$ nor $\{new\}$ as well as the object nodes related by the relation <<becomes>>. Complex collaborations are mapped to several rules according to the control flow specified. If parallel actions are modelled within a collaboration, they are mapped to parallel rules.

Graph transformation can explicitly model failures which is presented in [**1**]. For this purpose, special rule triggers such as failure events can be considered, which cause the application of certain error actions like stop actions. A stop action changes the state of a local system in a way that no further action can be performed. Furthermore, security issues resulting in a set of new stereotypes in UML, can be based semantically on graph transformation leading to new types of objects and dependencies. Recently, the well-known role-based security mechanism RBCA has been specified with graph transformation [68].

Altogether, the rule concept of graph transformation provides one clear concept to define system behaviour. Especially for modelling the intrinsic concurrency of actions, graph rules are an adequate means, because they explicate all structural interdependencies. Moreover, additional control flow for rule application can be defined by control conditions in transformation units which however, do not support a nice visualization of control flow structures so far.

Constraints. Constraints occur at different places in system models. Simple ones can be translated to graph constraints, while more complex ones have to be translated to rules and transformation units. Graph constraints and their ensurance have originally be presented in [61]. We consider constraints in class diagrams to describe the object structures allowed. Furthermore, they can occur as any kind of invariant, pre and postcondition for operations, guards in state transitions or collaborations, etc. OCL has been defined as extension of UML providing a constraint language for object systems. Although the following explanations are formulated for a translation of OCL, please note that the basic ideas of this approach are independent of the concrete language OCL, but apply to any constraint language.

In [**2**], we showed how OCL constraints can be translated to graph rules and transformation units. Each constraint is defined concerning a certain model element which is specified as context. Invariants, pre and post conditions are special kinds of constraints which specify when a translated constraint should be checked, i.e. when its corresponding rules and transformation units are tried to be applied to a certain instance graph. That means for an invariant that if corresponding graph rules or transformation units are applicable to all instances nodes of the context model element, the corresponding constraint is semantically consistent, otherwise an inconsistency has been found.

The approach to base OCL heavily on collaborations suits well to its semantic interpretation. As described above, collaborations can be semantically explained as rules or transformation units. Pre-defined types and operations such as collections and operations like forall, select, exist, etc. are mapped to special object types and pre-defined transformation units.

Evolution. Model evolution can be viewed as a process in which transformations are applied successively to a system model. Semantically, we describe a distributed system by a distributed graph transformation system (DGTS). Thus, model evolution means a DGTS transformation. Such a transformation can be defined component-wise and can contain simple transformation concepts as well as more complex ones. Evolution of static structures like classes structures is easily to describe. So-called inference diagrams on e.g. class diagrams as they are presented in [52], can directly be interpreted as graph transformations. Evolution of behaviour is more difficult to formalize in the graph transformation domain. Here, we have to follow the ideas of transforming graph grammars presented in [81] which means applying rules to rules in particular. This approach to describe behaviour evolution seems to be promising and worthwhile to be elaborated in future work.

Automatic Model Translation. So far, we discussed the semantic mapping of a distributed object computing model, a UML metamodel instance in particular, to distributed graph transformation. In the previous section, we proposed to use graph transformation to formulate an automatic model translation. Of course, this approach can also be followed if the semantic domain is distributed graph transformation. But there is one point to pay attention to: The formulation of translation rules functions only if the target structure, being a distributed graph transformation system, is considered as a graph. But this not a big problem, since any kind of structure can be considered as a graph. In the next subsection, we consider the validation techniques offered by graph transformation which can be advantageously used for distributed system validation. In this way, we get a clear view which partial model translations are worthwhile to be further elaborated by supplying a translating graph transformation system.

5.3 Validation Support by Graph Transformation

In the following, we consider validation techniques available for graph transformation. Although these techniques are elaborated for non-distributed graph transformation, they can be used also for distributed graph transformation in a slightly restricted way. If all distributed graphs are joined and all distributed rules are amalgamated, the resulting graph transformation systems can be used for validation. Positive statements on consistency and independencies are valid also in the original distributed graph transformation system. Inconsistencies, conflicts and sequential dependencies can be transferred if the distribution structure is flattened for validation.

Conflicts and Dependencies. The first validation technique is based on the notion of independence of graph transformations which captures the idea that, in a given situation, two transformations are neither in conflict nor causally dependent. We distinguish parallel independence (absence of conflicts) and sequential independence (absence of causal dependencies). For both notions there

exists a weak, asymmetric, and a strong, symmetric version (see, e.g., [39] for a recent survey).

If two transformations are mutually independent, they can be applied in any order yielding the same result. In this case we speak of *parallel independence*. Otherwise, if one of two alternative transformations is not independent of the second, the second will disable the first. In this case, the two steps are *in conflict*. However, we lay here the focus on static analysis of *potential* conflicts and dependencies, rather than on run-time analysis. Therefore, the above notions have to be lifted to the level of rules.

The computation of potential conflicts and dependencies is based on the idea of *critical pair analysis* which is known from term rewriting. Usually, this technique is used to check whether a rewriting system has a functional behaviour, especially if it is locally confluent. Critical pairs have been generalized to graph rewriting in [83] and further extended to attributed graph transformation in **[59]**. They formalize the idea of a minimal example of a conflicting situation. From the set of all critical pairs we can extract the objects and links which cause conflicts or dependencies. In **[58]**, critical pair analysis has been used to detect conflicting functional requirements in a use case-driven approach.

Graph transformation is an ideal framework to reason about concurrent computations. In [38], *graph processes* have been defined containing a partial order of rule applications which results from causal dependency analysis. Moreover, an *event structure semantics* has been proposed for somehow restricted graph transformation in [37] abstracting from concrete transformations. It just models a collection of events with two relations, namely *causal dependency* and *conflict*. However, the kind of graph transformation considered is restricted in a way that only reactive behaviour instead of pro-active one can be modelled.

Considering for example distributed version control, we have to ensure that configurations of documents are checked in in a synchronized way. For this purpose, a lock has to be created in the beginning prohibiting the application of rules such that another check-in can be started. This means applications of rules which start a check-in, can be in conflict. Performing one of these rule applications prohibits the other one as long as the lock is released. A critical pair analysis finds all these conflicts which have to be analysed by the system designer afterwards. If a mutual exclusion on resources is intended, corresponding critical pairs are desired to show conflicts. Otherwise, critical pairs can give information on conflicting situations which are not desired, e.g. conflicts on check-out rules.

Structural Consistency Checking. *Graph constraints* [61] are used to describe basic consistency conditions on graphs such as existence or non-existence as well as uniqueness of certain graph parts. For example, multiplicity constraints as they occur in UML class diagrams can be translated to graph constraints (compare [94]). A graph transformation system is consistent wrt. a set of graph constraints if the start graph satisfies the constraints and all rules preserve them. In [61], an algorithm has been developed which checks whether a rule preserves all constraints. If a constraint is not preserved, new preconditions

are generated for this rule restricting its applicability such that consistency is always ensured. This procedure has been extended to attributed graph transformation in [76].

More complex consistency conditions such as the existence of paths or cycles of arbitrary length or other typical graph properties, have to be formulated by a set of transformation units which encode a constructive approach to consistency ensurance. Transformation units allow the definition of an initial graph class which can be advantageously used to formulate when consistency checking should be triggered. For example, consistency checking is useful when a new instance of a certain node or edge type has been created whose existence is bound to a number of conditions.

Considering application-specific consistency, distribution issues are irrelevant, thus, consistency checking on the amalgamated distributed graph transformation system would provide a simple, but satisfactory procedure for this kind of consistency. A consistency condition like the uniqueness of all document names in a revision archive is of this type. It can be formulated by a simple graph constraint not allowing graphs with two document nodes, both having the same name.

Distribution-specific consistency instead, deals directly with replication and sharing issues. For example, we postulate that document configurations have always be replicated completely between revision archives. Furthermore, we have to be sure that a document is not replicated several times to one and the same archive. To reason about distribution-specific consistency, we do not have to translate a model completely into the domain of distributed graph transformation, but it is enough to translate the replication structures and their modifications. Application-specific attributes are not needed for this kind of validation. Distributed graph transformation implicitly supports coherence (compare Section 4.1) due to its rule-based character of behaviour specification. Rules operating on different graph parts do not disturb each other, i.e. they are independent, and can be applied in any order. Stronger forms of consistency like sequential consistency, can be realized by additional graph structure enforcing a deterministic order of rule applications. Weak consistency as it occurs in distributed transactions has been modelled with distributed graph transformation in [67].

Tool Support by AGG. AGG is a general tool environment for algebraic graph transformation which supports visual editing and simulation of graph transformation systems as well as their validation. Its special power comes from a very flexible attribution concept. AGG graphs are allowed to be attributed by any kind of Java objects. Graph transformations can be equipped with arbitrary computations on these Java objects described by Java expressions. The AGG environment consists of a graphical user interface comprising several visual editors, an interpreter, and a set of validation tools. These are a critical pair analyser, a graph parser, and a consistency checker for graph constraints. The implementation of AGG follows the well-known model-view-control concept, i.e. its kernel can be flexibly used by its graphical user interface or by an

API when AGG is incorporated into a graph transformation-based application. The kernel concepts of AGG are comprehensively described in [**7**]. Recently, it has been extended by several validation techniques which are described in [**29**],[**58**], and [76]. To use AGG mainly as validation tool, its XML interface is important to connect AGG with a CASE tool as pointed out in Section 4.3. The current version of AGG can be found at: `http://tfs.cs.tu-berlin.de/agg`.

6 Summary of the Papers Submitted for the Habilitation Thesis

Up to now, this survey has presented the state-of-the-art in visual modeling and validation of distributed systems, focussing on UML as visual language and graph transformation as semantic domain for validation. The description of the main challenges when developing a distributed system and resulting requirements for distributed system modelling, the presentation of UML together with a variety of extensions as well as the description of validation issues and techniques and graph transformation as semantic domain have drawn a wide picture of research activities in this field. This section gives a summary of the author's achievements most relevant for the research on visual modelling and validation of distributed systems. Each of the following subsections summarizes one of the papers submitted for the habilitation thesis. These papers cover the most relevant contributions of the author concerning visual modelling and validation of distributed systems.

6.1 A Visual Modelling Framework for Distributed Object Computing

In [**1**], the author outlines the central framework for the habilitation thesis. After stating the main requirements for distributed system modelling a fragment of a slightly extended UML is presented, focussing on architecture and interaction modelling. Thereafter, distributed graph transformation is introduced as a semantic domain where the semantics of the architecture, object and interaction models are given by network graphs and transformations as well as object graphs and their transformations. Graph transformation system components are introduced to describe distributed components and their interaction. They are lateron related to I/O-automata [74] which are used as simple formal models for reasoning about distributed algorithms. In this sense, the theory of I/O-automata becomes applicable to (restricted) graph transformation systems useful to reason about concurrency issues of distributed systems.

6.2 Consistency Checking and Visualization of OCL Constraints

The constraint language OCL is an important extension of UML to formulate invariants, pre and post conditions as well as guards. Unfortunately, it has been given in a textual, quite unintuitive form and without a precise semantics in the beginning. In [**2**], G. Taentzer developed the main concepts to visualize

OCL on the basis of collaborations and discussed her work with the other three authors. She suggested a closer integration of the UML and the OCL metamodels which is possible when relying on collaborations. The visualization has been further elaborated in [**31**] and captures all main concepts of OCL. Moreover, the authors consider consistency checking of OCL constraints by translating OCL constraints to graph rules and transformation units. To check the consistency of a UML model the applicability of resulting graph rules and transformation units to the abstract syntax graph of the model is tested. If all rules and transformation units are applicable to all instances of constrained model elements, the UML model is consistent. Otherwise, an inconsistency can be reported.

6.3 Application of Graph Transformation to Visual Languages

In [**3**], we show how graph transformation can be applied to visual languages: On the metamodel level, graph transformation can be used to define visual languages. Graphs are well suited to describe the multi-dimensional structure of visual sentences such as diagrams and graphics. Similarly to textual languages, a grammar defines a visual language in terms of its graph language. The general concept of visual language definition distinguishes Spatial Relations Graphs (SRGs) which describes the structure of a visual sentence seen as a picture and Abstract Syntax Graphs (ASGs). Graph grammars are used to define the translation from SRG to ASG (low-level parsing) and for high-level parsing on the ASG solving the membership problem. Considering the GenGED approach to visual language definition which has been developed by R. Bardohl under the guidance of G. Taentzer, a generalized form of graphs, i.e. graph structures, is used to describe visual sentences. The graphical layout of visual sentences is described by graph attributes and graphical constraints. The GenGED environment supports the visual definition of a visual language and generates a corresponding visual editor and a visual simulator. The editor offers a syntax-directed editing mode as well as more free-hand editing with parsing facilities. GenGED is based on the graph transformation engine AGG which has been developed under the leadership of the author. In future work, the GenGED approach can advantageously be used to define e.g. a visual OCL according to the concepts described in [**31**]. In this way, a visual editor and a consistency checker could be generated.

On the other hand, graph transformation languages themselves provide a visual modelling or high-level programming language where the underlying data model is a graph. Graph rules and more sophisticated control structures for rule application are used to describe data manipulation. Two general purpose languages PROGRES and AGG are presented by A. Schürr and G. Taentzer. Furthermore, two languages for visual language definition, DiaGen and GenGED, are presented by M. Minas and R. Bardohl. All four languages have been discussed and compared to each other by all authors.

6.4 A Combined Reference Model- and View-based Approach to System Specification

Based on graph transformation, a specification technique is presented in [**4**] which combines a reference and view-based approach. A formal notion of views and view relations is developed and the behaviour of views is described by a loose semantics. Moreover, the integration of views which are based on a common reference model, is considered by R. Heckel and G. Taentzer. For the view integration, dependencies between views not given by the reference model are determined first. Then, the reference model is extended by the model manager. When the views and their common reference model are consistent, the actual view integration can be performed automatically. In addition, several extensions of this basic integration scenario are discussed. Throughout this paper, R. Heckel and G. Taentzer illustrated all concepts at a running example being a banking system. Based on this work, application-specific views in UML can be defined as presented in Section 3.1. Semantically, views are defined as graph transformation systems, compare Section 5.1.

6.5 Distributed Graphs and Graph Transformation

In [**5**], the author presents the formal framework of distributed graphs and graph transformation. It defines structured graph transformation on two abstraction levels: the network and the local level. The network level contains the description of the topological structure of a system. The local level covers the description of states and their transitions. In this way, the state of a distributed system is represented by a number of distributed states, partly dependent of each other. State transitions can be local or distributed and might be synchronized by common subtransitions. Modelling distributed systems in this way offers a clear and appealing description due to its visual form and the few number of concepts which allow to describe all main concepts of distributed systems such as complex distributed data structures, dynamic networks, distributed actions as well as communication and synchronization. The formalization is based on category theory, namely the category DISTR(GRAPH) of distributed graphs and graph morphisms. A distributed graph transformation step is characterized by a double-pushout within this category which is constructed component-wise for each local graph to reflect distributed computations best. Component-wise transformations exist if certain conditions, so-called distributed gluing conditions are satisfied, and lead always to a unique result.

6.6 Distributed Graph Transformation With Application to Visual Design of Distributed Systems

Having the formal framework of distributed graph transformation at hand, it can be used to visually describe the key design of a distributed system. To support precise modelling of network, data and interaction issues, the basic formalism given in [**5**] has been extended in [**6**] by attributes and application conditions for rules. This work has been done by M. Koch under the guidance of G. Taentzer. In this approach, network graphs are not attributed, but

labelled and network morphisms are injective in rules and occurrences to facilitate component-wise rule application.

The whole resulting framework is informally introduced and illustrated at examples by G. Taentzer to show how main aspects of distributed systems can be expressed by distributed graph transformation. A UML-like notation is employed in the examples such that software developers familiar with object-oriented modelling techniques can easily understand how distributed graph transformation can be used to visually model distributed systems. Moreover, I. Fischer and V. Volle developed a larger reference application in [**6**] under the guidance of G. Taentzer. The application is a distributed version management system where several revision archives are distributed over a network and build a distributed shared memory system. Each revision archive contains those documents relevant in its own site. It provides the service of versioning for local workspaces and supports replication between sites. The system is designed as open system where the number and the connections between revision archives may evolve.

6.7 The AGG-Approach: Language and Environment

In [**7**], AGG is presented which can be considered as visual modelling or very high-level programming language modelling the kernel data structures of software systems as graphs. Since AGG graphs can be attributed by arbitrary Java objects, its attribution concept is extremely flexible. It allows to use graphs on very different abstraction levels: Control graphs coordinating processes would be high-level whereas graphs as data model is a more low-level application. The application's behaviour is described by graph rules which can contain Java expressions to describe attribute computations. The development of the AGG environment in Java has been started by G. Taentzer in 1995. It comprises now visual editors, an interpreter, debugger and graph parser as well as validation tools. In [**7**], the kernel concepts of the AGG language and environment are described by G. Taentzer. Moreover, C. Ermel worked out a case example solving the shortest path problem by AGG. AGG environment and implementation issues being described by M. Rudolf, mirror the AGG version of the year 1999. In the meantime, AGG has been updated and extended in several directions, under the guidance of G. Taentzer: After having implemented the graph transformation engine, all the validation techniques which have been developed for attributed graph transformation, shall be gradually implemented in AGG. Up to now, a graph parser, critical pair analysis, and a constraint checker have been realized. Furthermore, the controlling of rule applications by rule layers and attribute conditions as well as the concept of type graphs has been added. The component concept for graph transformation described in Section 5.1 has been realized prototypical, its full realization in AGG is future work.

AGG has developed to one of the standard environments in the graph transformation community. The current version and its documentation are available at: `http://tfs.cs.tu-berlin.de/agg`.

7 Conclusions

This survey describes the visual modelling and validation of distributed systems focussing on the visual modelling language UML and graph transformation as formal validation domain. UML has been extended in various directions and meets now all main requirements for distributed system modelling. For a precise definition of the syntax and semantics of a visual modelling language like UML, graph transformation has been shown to be a promising technique. It allows a visual language definition which handles all structural aspects visually. For the visual reasoning on all key aspects of distributed systems, the formal calculus of distributed graph transformation has been developed which serves as a formal semantic domain for distributed system models. The main advantage of this calculus is a support for the formal validation of consistency issues, rarely investigated in distributed system models so far. Summarizing, main contributions have been achieved concerning the following four subjects:

- A conceptual framework for the visual modelling of distributed systems [**1**] where distributed graph transformation [**5**] is used as semantic domain [**6**].
- An application-specific view concept for UML [**4**] and visualization concepts for OCL [**2**].
- Definition of visual languages by graph transformation [**3**].
- Tool support concerning validation and visual language definition given by AGG [**7**].

Main parts of this work have been developed in a project called "Application of graph transformation to the visual design of distributed systems", supported by the Deutsche Forschungsgemeinschaft (DFG). This project was leaded by the author with regard to the realization of project activities. Main ideas concerning distributed graph transformation as semantic domain for distributed systems, a considerable tool support by AGG, and the GenGED approach to visual language definition have been developed within this project. It is now continued by the project "Application of graph transformation to visual modelling languages" and concentrates on the usage of graph transformation for syntax and semantic definition of visual modelling languages. Elaborated concepts are applied to concrete languages such as UML and extensions, Petri net-based languages as well as domain-specific languages. This research on visual modeling techniques is flanked Europe-wide by the new European Research Training Network on "Syntactic and Semantic Integration of Visual Modeling Techniques" (SeGraVis) which started in October 2002.

After laying the basis for the precise syntax and semantics definition of visual modelling languages for distributed systems, future activities are planned to integrate UML and graph transformation-based techniques with other techniques for modelling and validation to come up with adequate domain-specific solutions. Furthermore, the developments of the last years produced an increasing number of attractive modelling and validation tools, especially for UML.

Time is mature also for graph and graph transformation-based tools which play an important role to draw the attention of practitioners to concepts and solutions as presented in this thesis. To enlarge the acceptance of these new kinds of tools, their integration with established solutions has to be pushed. The development of common exchange formats such as GXL for graphs and GTXL for graph transformation systems, is one important step into this direction. These formats facilitate the usage of graph-based tools in various concrete solutions which yields to a wide spread of numerous concepts and results for graphs and graph transformation into different application fields such as software engineering, visual languages and distributed systems.

References

Publications Submitted as Part of this Thesis

[1] G. Taentzer. A Visual Modeling Framework for Distributed Object Computing. In B. Jacobs and A. Rensink, editors, *Proc. Formal Methods for Open Object-based Distributed Systems V (FMOODS'02)*. Kluwer Academic Publishers, 2002.

[2] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency Checking and Visualization of OCL Constraints. In A. Evans and S. Kent, editors, *UML 2000 - The Unified Modeling Language*, volume 1939 of *LNCS*. Springer, 2000.

[3] R. Bardohl, M. Minas, A. Schürr, and G. Taentzer. Application of Graph Transformation to Visual Languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[4] G. Engels, H. Ehrig, R. Heckel, and G. Taentzer. A combined reference model- and view-based approach to system specification. *Int. Journal of Software and Knowledge Engineering*, 7(4):457–477, 1997.

[5] G. Taentzer. Distributed Graphs and Graph Transformation. *Applied Categorical Structures*, 7(4):431–462, December 1999.

[6] I. Fischer, M. Koch, G. Taentzer, and V. Volle. Distributed Graph Transformation with Application to Visual Design of Distributed Systems. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*, pages 269–340. World Scientific, 1999.

[7] C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999. available at: `http://tfs.cs.tu-berlin.de/agg`**.**

Other References

[8] *JavaScript 1.1 Language Specification* *http://wp.netscape.com/eng/javascript/*.

[9] *JavaScript Homepage* *http://java.sun.com/products/javaspaces*.

[10] *Java Remote Method Invocation* *http://java.sun.com/products/jdk/rmi*.

[11] *Hyper Text Markup Language - 4.01 Specification* *http://www.w3.org/TR/html4/*, 1999.

[12] *Hyper Text Transfer Protocol* *http://www.w3.org/Protocols/*, 2000.

[13] *Z.100 - Specification and Description Language (SDL)*, 2000. ITU-T Recommendation.

[14] *Web Service Definition Language* *http://www.w3.org/TR/wsdl*, 2001.

[15] *Simple Object Access Protocol (SOAP) 1.1* *http://www.w3.org/TR/SOAP*, 2002.

[16] *The Extensible Stylesheet Language (XSL)* *http://www.w3.org/Style/XSL/*, 2002.

[17] *CORBA/IIOP Specification* *http://www.omg.org/technology/documents/formal/corba_iiop.htm*, 2002.

[18] *Dynamic Host Configuration Protocol* *http://www.dhcp.org/*, 2002.

[19] *Jini[tm] Network Technology* *http://wwws.sun.com/software/jini/*, 2002.

[20] *OCL 2.0* *http://cgi.omg.org/cgi-bin/doc?ad/00-09-03*, 2002. OMG.

[21] *Scalable Vector Graphics* *http://www.w3.org/TR/SVG/*, 2002.

[22] *The precise UML group* *http://www.cs.york.ac.uk/puml/*, 2002.

[23] *Universal Description, Discovery and Integration of Business for the Web* *http://www.uddi.org/*, 2002.

[24] *XML Metadata Interchange* `http://www.omg.org/technology/documents/formal/xmi.htm`, 2002.

[25] David H. Akehurst. Model Translation: A UML-based specification technique and active implementation approach. PhD Thesis, University of Kent, Dept. of Computer Science, 2000.

[26] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph Transformation for Specification and Programming. *Science of Computer Programming*, 34:1–54, 1999.

[27] R. Bardohl. GenGEd – *Visual Definition of Visual Languages based on Algebraic Graph Transformation.* Verlag Dr. Kovac, 2000. PhD thesis, Technical University of Berlin, Dept. of Computer Science, 1999.

[28] R. Bardohl. A Visual Environment for Visual Languages. *Science of Computer Programming (SCP)*, 44(2):181–203, 2002.

[29] R. Bardohl, T. Schultzke, and G. Taentzer. Visual Language Parsing in GenGEd. *Electronic Notes of Theoretical Computer Science*, 50(3), June 12–13 2001.

[30] L. Baresi and M. Pezze. A Toolbox for Automating Visual Software Engineering. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *LNCS*, pages 189 – 202. Springer, 2002.

[31] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A Visualization of OCL using Collaborations. In *UML 2001 – The Unified Modeling Language*, LNCS 2185, pages 257 – 271. Springer, 2001.

[32] P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Constraint Distributed Diagram Transformation for Software Evolution. In R. Heckel and T. Mens, editors, *Workshop "Software Evolution Through Transformations"*, volume 72.4. Electronic Notes in Theoretical Computer Science, 2002. to appear.

[33] P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. Techn. report no. si-2000-06, University of Rome, 2000.

[34] N. Carriero and D. Gelernter. Linda in Context. *Communications in ACM*, 32(4):444 – 458, 1989.

[35] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software.* Addison–Wesley, October 2000.

[36] J. Conallen. *Building Web Applications with UML.* Addison-Wesley, 2000.

[37] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. An Event Structure Semantics for Consumptive Graph Grammars with Parallel Productions. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 1073 of *LNCS*, pages 240 – 256. Springer, 1996.

[38] A. Corradini, U. Montanari, and F. Rossi. Graph Processes. *Special Issue of Fundamenta Informaticae*, 26(3,4):241–266, 1996.

[39] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter 3. World Scientific, 1997.

[40] G. Costagliola,A. De Lucia, S. Orefice and G.Tortora. Automatic Generation of Visual Programming Environments. *IEEE Computer,28[3]:56 – 66, 1995.*

[41] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, 2001.

[42] J. de Lara and H. Vangheluwe. ATOM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling. In R. Kutsche and H. Weber, editors, *Proc. Fundamental Approaches to Software Engineering (FASE'02), Grenoble, April 2002*, pages 174 – 188. Springer LNCS 2306, 2002.

[43] J. de Lara and H. Vangheluwe. Computer Aided Multi-paradigm Modelling to Process Petri-Nets and Statecharts. In A. Corradini, H. Ehrig, H.-J. Kreowski and Rozenberg. G., editors, *Proc. of 1st Int. Conference on Graph Transformation.*, LNCS 2505 Springer Verlag, pages 239 – 253.2002.

[44] *UML Profile for Enterprise Distributed Object Computing Specification*, 2002. Available at `http://cgi.omg.org/docs/ptc/02-02-05.pdf`.

[45] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools.* World Scientific, 1999.

[46] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution.* World Scientific, 1999.

[47] H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors. *Advances in Petri Nets: Petri Net Technologies for Modeling Communication Based Systems.* LNCS. Springer, 2002. To appear.

[48] G. Engels, R. Heckel, and J. M. Küster. Rule-based Specification of Behavioral Consistency Based on the UML Meta-model. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools., 4th International Conference, Toronto,*

Canada, October 1-5, 2001, Proceedings, volume 2185 of *LNCS*, pages 272–287. Springer, 2001.

[49] A. Finkelstein, D. Gabbay, H. Hunter, J. Kramer and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. In *IEEE Transactionson Software Engineering.* 20(8):569–578, 1994.

[50] I. Fischer, M. Koch, and G Taentzer. Local Views on Distributed Systems and their Communications. H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, Theory and Application of Graph Transformations, pages 164–178. LNCS 1764 Springer, 2000.

[51] Formal Systems Europe (Ltd). *Failures-Divergence-Refinement: FDR2 User Manual*, 1997.

[52] R. France and J. Biemann. Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software. In *Proc. of Int. Conf. on Software Maintenance.* IEEE Computer Society, 2001.

[53] D. Garlan and A. Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language*, volume 1939, pages 498 – 512. Springer LNCS, 2000.

[54] M. Goedicke, T. Meyer, and G. Taentzer. ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies. In *Proc. 4th IEEE Int. Symposium on Requirements Engineering (RE'99), June 7-11, 1999, University of Limerick, Ireland.* IEEE Computer Society, 1999. ISBN 0-7695-0188-5.

[55] M. Grosse-Rhode. *Semantic Integration of of Heterogenous Software Models.* Habilitation thesis, Technical University of Berlin, Dep. of Comp. Sci., 2002.

[56] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[57] S. Gyapay, R. Heckel, and D. Varro. Graph Transformation with Time: Causality and Logical Clocks. In *In Proceedings of 1st International Conference on Graph Transformation.* LNCS 2505 Springer, pages 120 – 134, 2002.

[58] J.H. Hausmann, R. Heckel, and G. Taentzer. Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In *Proc. of Int. Conference on Software Engineering 2002*, Orlando, USA, IEEE Society, 2002.

[59] **R. Heckel, J. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation with Constraints. In A. Corradini, H. Ehrig, H.-J. Kreowski, and Rozenberg. G., editors, *Proc. of 1st Int. Conference on Graph Transformation.* LNCS 2505 Springer Verlag, pages 161 –176, 2002.**

[60] **R. Heckel, J. Küster, and G. Taentzer. Towards Automatic Translation of UML Models into Semantic Domains. In H.-J. Kreowski, editor, *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002)*, pages 11 – 22, 2002.**

[61] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Grammars – A Constructive Approach. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[62] **Reiko Heckel, Hartmut Ehrig, Gregor Engels, and Gabriele Taentzer. Classification and Comparison of Modularity Concepts for Graph Transformation Systems. In H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, pages 669 – 690. World Scientific, 1999.**

[63] W. Ho, J.-M. Jezequel, A. Guennec, and F. Pennaneac'h. UMLAUT: An Extendible UML Transformation Framework. In *Proc. Automated Software Engineering, ASE'99, Florida*. IEEE Computer Society, 1999.

[64] W. Ho, F. Pennaneac'h, and N. Plouzeau. UMLAUT: A framework for weaving UML-based aspect-oriented Designs. In *Technology of object-oriented languages and systems (TOOLS Europe)*, pages 324 – 334. IEEE Computer Society, 2000.

[65] J. Howse, F. Molina, J. Taylor, S. Kent, and J. Gil. Spider Diagrams: A Diagrammatic Reasoning System. *Journal of Visual Languages and Computing*, 12(3):299 – 324, 2001.

[66] S. Kent. Constraint diagrams: Visualising invariants in object oriented models. In *Proceedings of OOPSLA'97*. ACM Press, 1997.

[67] M. Koch. Bedingte verteilte Graphtransformation und ihre Anwendung auf verteilte Transaktionen. Technical Report 97-11, Technical University of Berlin, Dep. of Comp. Sci., 1997.

[68] M. Koch, L. Mancini, and F. Parisi-Presicce. A Graph Based Formalism for RBAC. *ACM Transactions Information and System Security (TISSEC)*, 5(3):332 – 365, 2002.

[69] P.J. Leach, P.H. Levine, B.P. Douros, AQ.J. Hamilton, J.A. Nelson, and L.B. Stumpf. The Architecture of an Integrated Local Network. *IEEE J. Selected Areas in Communications*, SAC-1(5):842 – 856, 1983.

[70] J. Lilius and I. Porres Paltor. vUML: A Tool for Verifying UML Models. In *Proc. Automated Software Engineering, ASE'99, Florida*. IEEE Computer Society, 1999.

[71] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. Wiley and Sons, 1998.

[72] C. Lindemann, A. Thümmler, A. Klemm, M. Lohmann, and O. Waldhorst. Performance Analysis of Time-Enhanced UML Diagrams Based on Stochastic Processes. In *Proc. of 3rd Int. Workshop on Software and Performance (WOSP), Rome, Italy*, pages 25 – 34, 2002.

[73] C. Lüer and E. Rosenblum. UML Component Diagrams and Software Architecture - Experiences from the Wren Project. In *Proc. of 'Describing Software Architecture with UML', a workshop at the 23rd Int. Conf. on Software Engineering, Toronto, Canada*, 2001.

[74] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[75] K. Marriott and B. Meyer. *Visual Language Theory*. Springer, 1998.

[76] M. Matz. Konzeption und Implementierung eines Konsistenznachweisverfahrens für attributierte Graphtransformation. Master's thesis, TU Berlin, 2002.

[77] T. Mens and T. D'Hondt. Automating Support for Software Evolution in UML. *Automated Software Engineering*, 7:39–59, 2000.

[78] M. Minas. Concepts and Realization of a Diagram Editor Generator based on Hypergraph Transformation. *Science of Computer Programming*, 44(3):157 – 180, 2002.

[79] C. Nentwich, W. Emmerich and A. Finkelstein. Static Consistency Checking for Distributed Specifications. *Int. Conference on Automated Software Engineering* Coronado Bay, CA, 2001.

[80] ISO/IEC International Standard 10746, ITU–T recommendation X.901–X.904: Reference model of open distributed processing – Parts 1–4.

[81] F. Parisi-Presicce. Transformation of Graph Grammars. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, 1996.

[82] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[83] Detlef Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In M.R Sleep, M.J. Plasmeijer, and M. C.J.D. van Eekelen, editors, *Term Graph Rewriting*, pages 201–214. Wiley, 1993.

[84] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[85] Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, 1998.

[86] J. Rekers and A. Schürr. A Graph Based Framework for the Implementation of Visual Environments. In *Proc. IEEE Symposium on Visual Languages (VL'96)*, pages 148–155, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[87] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. Monographs of the Bremen Institute of Safe Systems. University of Bremen, 2002. PhD thesis at the University of Bremen.

[88] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[89] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. Wiley & Sons, Inc., 1994.

[90] B. Steffen, T. Margaria, and V. Braun. The Electronic Tool Integration platform: concepts and design, *Software Tools for Technology Transfer*, Vol. 1, 1997. Springer, 9 - 30.

[91] G. Taentzer. Parallel high-level replacement systems. *TCS*, 186, November 1997.

[92] G. Taentzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. In J. Padberg, editor, *Proc. Uniform Approaches to Graphical Process Specification Techniques (UNIGRA01)*, volume 47 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2001.

[93] G. Taentzer and A. Schürr. DIEGO, Another Step Towards a Module Concept for Graph Transformation Systems. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[94] A. Tsiolakis and H. Ehrig. Consistency Analysis between UML Class and Sequence Diagrams using Attributed Graph Gammars. In H. Ehrig and G. Taentzer, editors, *Proc. GRATRA'2000 - Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 77–86. Technische Universität Berlin, March 25-27 2000.

[95] *Unified Modeling Language – version 1.4*, 2002. Available at `http://www.omg.org/technology/documents/formal/uml.htm`.

[96] D. Varro. Automated Program Generation for and by Model Transformation Systems. In H.-J. Kreowski and P. Knirsch, editors, *Applied Graph Transformation (AGT'02)*, pages 161 – 174. Satellite Workshop of 'European Joint Conferences on Theory and Practice of Software, 2002.

[97] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10:91–102, 1992.

[98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[99] A. Winter. Exchanging Graphs with GXL. In P. Mutzel, editor, *Graph Drawing - 9th Interational Symposium, GD 2001, Vienna, September 23-26, 2001,Mathematics and Vizualization*. Springer, 2001.

[100] A. Zarras and V. Issarny. UML-Based Modeling of Software Reliability. In *in Proc. of 'Methods and Techniques for Software Architecture, Review and Assessment', a workshop at the 24rd Int. Conf. on Software Engineering, Orlando, USA*, 2002.

[101] L. Zhang and E. Shin. UML-based Representation of Role-based Access Control. In *In Proceedings of 5th IEEE International Workshop on Enterprise Security (WETICE 2000), NIST, MD*, 2000.