ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies

Michael Goedicke and Torsten Meyer
Dept. of Mathematics and Computer Science
University of Essen, D - 45117 Essen
{goedicke,tmeyer}@informatik.uni-essen.de

Gabriele Taentzer
Computer Science Deptartment
Technical University of Berlin, D - 10587 Berlin
gabi@cs.tu-berlin.de

Abstract

Software development is a staged and evolutionary process. Multiple stakeholders with different needs and views collaborate to build a system from interoperating and heterogeneous development artifacts. In such a setting, one has to cope with requirements changing dynamically during the entire lifetime of the system. Within this changing world living with inconsistencies is natural. Tool support is needed to tolerate inconsistencies and help developers to use them to drive the development process forward. In this contribution we consider the application of distributed graph transformation to the problem of formalizing the integration of multiple perspectives in software development called 'ViewPoints'. Our work concentrates on requirements engineering. The paper demonstrates how inconsistency management can be used as a tool for requirements analysis by presenting a sample integration of architecture design and performance evaluation views.

1. Introduction and related work

System design is a staged process as for example the spiral model of software development suggests. The various development stages are visited more than once. In such a setting multiple participants collaborate to construct a wide range of development artifacts. In addition, multiple components of the software system under construction must interoperate effectively to achieve the desired behaviour. Especially for large projects various participants with different needs and even conflicting views are involved.

The change of requirements to these development artifacts is an evolutionary process which does not finish before the lifetime of the entire system has expired [6]. New and evolving requirements originate from changes in the world, for example from changes in people's expectations, business practices, social forces, or enabling technology opportunities. Dynamic accommodation of these evolutionary changes is highly desirable.

. Part of the work reported here is funded by the DFG project QUAFOS-2, contract MU1158/2-1.

Within such a setting inconsistencies emerge naturally - not only as consequences of disagreement among stakeholders of a development process but also as effects of constantly changing requirements [5, 6]. Detecting and resolving inconsistencies is only part of the problem: for example a resolved inconsistency might not stay resolved as the system evolves or an inconsistency might resolve itself after a brief delay by other compensating actions.

Traditionally inconsistency between development artifacts is regarded as a state which has to be avoided at all cost. In contrast to this view our approach takes a broader view: within software development the changing of requirements and system properties is a continuous process and inconsistencies arise quite naturally within such a process. Thus it is highly desirable to incorporate inconsistencies in the development process. It is important to develop techniques how to tolerate inconsistencies and use them to drive the development process forward.

It has been shown that ViewPoints can be employed for organizing and describing the software development process in a useful way [13, 14]. Here we take this work as a foundation and provide a formal basis for defining processes and software artifacts more precisely. Distributed graph transformation is a nice formal basis at a problem oriented level and is well suited for execution as well as animation. Furthermore, it serves as a better basis for inconsistency checks and analysis.

The concepts as well as the formal definition of distributed graph transformation are based on the double-pushout approach to algebraic graph transformation [2] where basic concepts from category theory are applied. Distributed graph transformation is introduced formally in [19].

The ViewPoints framework was devised by A. Finkelstein et al. [7] and B. Nuseibeh [13] to describe complex systems. An overview of other approaches related to multiple perspectives in software development can be found in [8]. In [14] a general overview wrt. inconsistency management is given. That work used a logic approach to describe inconsistency check actions between ViewPoints. This approach is extended towards quasi-classical (QC) logics in [11]. In contrast to our approach, QC logic is only used for the definition of ViewPoint check actions. Our approach uses graph transformation to provide the entire ViewPoints framework with formal underlying semantics. Thus not only check actions are formalized, but general benefits of

formal specification are achieved for the entire ViewPoint architecture (cf. chapter 5).

In [6, 5] S. Easterbrook, B. Nuseibeh, A. Finkelstein and J. Kramer emphasize the importance of tolerating inconsistencies and propose a process model for inconsistency management. The current paper uses these ideas a foundation and shows how they can be realized by distributed graph transformation.

This paper shows how distributed graph transformation can be used as a natural formalism serving as the underlying formal semantics for the ViewPoints framework. Hence distributed graph transformation functions as a meta modeling approach to make the various (requirements engineering) methods encapsulated by single ViewPoints comparable. This is necessary to enable Inter-ViewPoint check actions between heterogeneous specifications and does not intend to unify the different views of different stakeholders. A general overview wrt. the services offered by meta modeling environments can be found in [1, 17]. ISO proposes a standard meta modeling framework in [12]. K. Pohl et al. discuss the integration of process model aspects of multiple heterogeneous user viewpoints in [17, 16]. In [21] V. Wiels and S. Easterbrook use category theory in order to integrate specifications and manage dynamic change. Since our graph transformation approach is also based upon category theory, further work would be to identify interfaces between both models. A framework to manage ontological overlaps and to integrate same system aspects in different notations is proposed by G. Spanoudakis et al. in [18]. These ideas can be used to elaborate the correspondence relation within our framework (cf. section 3.4.) in more detail.

In chapter 2 we will introduce briefly the basics of our approach: the ViewPoints framework, distributed graph transformation, and the integration of both. Based upon this we will demonstrate in chapter 3 how distributed graph transformation can be used as underlying formal semantics for a set of sample ViewPoints integrating architecture design and performance evaluation perspectives. Finally, in chapter 4 we will sketch how inconsistency management actions can be represented by distributed graph transformation.

2. Basics

First we summarize the basic principles of the ViewPoints framework (section 2.1.) and distributed graph transformation (section 2.2.). Then we sketch the integration of both (section 2.3.).

2.1. ViewPoints framework

ViewPoints have been successfully used in a wide variety of domains to express different views and plans of participants in a development process. A *ViewPoint* can be thought of as a combination of the idea of an 'actor', 'knowledge source', 'role' or 'agent' in the development process, and the idea of a 'view' or 'perspective' which an actor maintains [5]. In software engineering terms, a ViewPoint is defined to be a locally managed object or agent which encapsulates partial knowledge about the system and its domain. It contains partial knowledge of the design process [7]. The knowledge is specified in a particular, suitable representation scheme. An entire system is described by a set of related, distributable ViewPoints which are loosely coupled.

A single ViewPoint consists of five slots. The *style* slot contains a description of the scheme and notation which is used to describe the knowledge of the ViewPoint. The *domain* slot defines the area of concern addressed by the ViewPoint. The *specification* slot contains the actual specification of a particular part of the system which is described in the notation defined in the style slot. The fourth slot is called *work plan* and encapsulates the set of actions by which the specification can be built as well as a process model to guide application of these actions. Two especially important classes of work plan actions are *In-ViewPoint check actions* and *Inter-ViewPoint check actions* which are used for checking consistency within a single ViewPoint or between multiple ViewPoints. The last slot of a ViewPoint called *work record* contains the development history in terms of the actions given in the work plan slot.

The ViewPoints framework is independent from any particular development method and actively encourages multiple representations. Software development methods and techniques are defined as sets of ViewPoint templates which encapsulate the notations provided as well as the rules how they are used. While these rules may refer to an isolated ViewPoint template, integration of methods (views) is realized by rules referring to multiple ViewPoint templates (ViewPoints).

Within the ViewPoints framework, there is no requirement for changes to one ViewPoint to be consistent with other ViewPoints. Thus inconsistencies are tolerated. Inconsistencies are managed by explicitly representing relations that should hold, within a single ViewPoint or between multiple ViewPoints. Consistency checking is performed by In-ViewPoint and Inter-ViewPoint check actions defined within the work plan slot. This allows consistency to be checked incrementally between ViewPoints at particular stages rather than being enforced as a matter of course. A protocol of development actions including inconsistency management information is recorded within a ViewPoint's work record slot. This is necessary for realizing inconsistency management strategies, e.g., incremental or delayed resolution. In chapter 4 we will investigate our framework for detecting inconsistent views and acting in the presence of inconsistency in detail.

A more detailed description of the ViewPoints framework is given in [7, 5]; examples for ViewPoints and interdependencies between them are presented in the following chapters. In the next section we will sketch the techniques

which are used as formal underlying semantics of the ViewPoints framework: distributed graph transformation.

2.2. Distributed graph transformation

In the following, we introduce a recent specification technique, called *distributed graph transformation*, which is structured graph transformation at two abstraction levels: the *network* and the *local* level. The network level contains the description of a system's network topology by a network graph and its dynamic reconfiguration during runtime by network rules. The application of such a rule to a network graph follows the double-pushout approach to graph transformation as presented in [2].

The combination of the network graph structure and the local object structures is specified by distributed graphs in order to describe distributed object structures. A *distributed graph* consists of one *network graph* where each network node is equipped with a *local graph* describing locally used data and object structures. Each network edge is equipped with a relation on local graphs defining replicated / used data and object structures in that way.

In the early stages of system development, a global view on the entire distributed system is desirable to get an overview. Once the network structure is fixed, the system developers are supposed to take a local view on network nodes and local system parts running on them to facilitate concurrent development. The *local view* of a local system contains this local system together with remote import and export interfaces to which it may have connections. Communication between local systems takes place via export and import interfaces. In *export interfaces*, local systems present data and objects accessible for other local systems and *import interfaces* contain data and objects required from remote export interfaces.

Each node and edge in a distributed graph may be typed and attributed by further data types. Usually, each local system contains different sets of node, edge and attribute types. The interfaces may contain any subsets of types for communication (cf. [19] for distributed graph attribution).

Network nodes being the source of a network edge may be interpreted as interfaces. Figure 1 shows an example of a distributed graph. Compare Figure 1 and Figure 2 for sample network graphs. Figure 3 and Figure 4 show sample distributed graphs where import and export graphs are distinguished by different gray shades of local graph elements. In all these examples, replication / use relations are unambiguously presented by obeying compatibility with attributes.

Graph rewrite rules are used to transform graphs in a rule based manner: *if* a certain structure exists *then* it may be transformed into a new one. At an abstract level a graph rewrite rule consists of a *left hand side rule graph L* and a *right hand side rule graph R*. Within a graph rewrite step the occurrence of L in a working graph is replaced with R. Distributed graph rules are useful to describe dynamic network reconfiguration by a network rule as well as dynamic

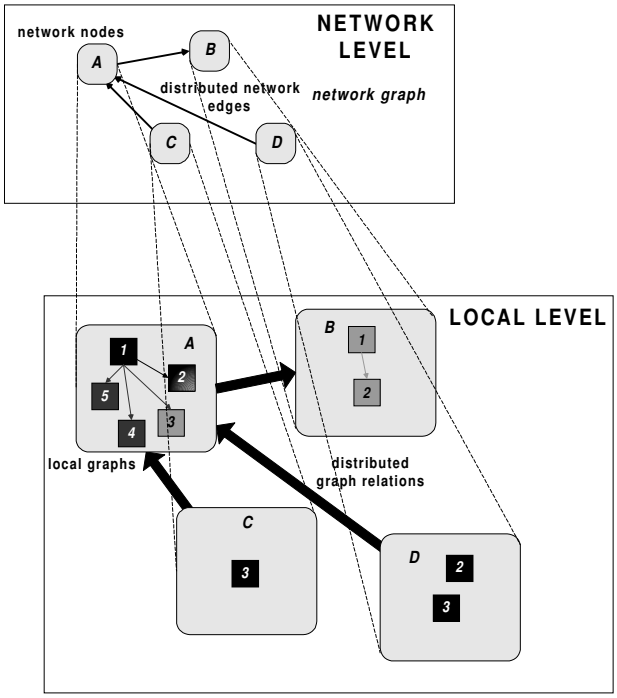

Figure 1 This example of a distributed graph shows a local view of the system *A*. In this and in the following distributed graphs body elements of local graphs are depicted in ***medium gray***, exported graph elements are depicted in *light gray*, imported graph elements are depicted in ***black***, and graph elements which are both imported and exported are depicted in mixed *light gray* / ***black***. Relations between local graph elements are not explicitly depicted, we assume a mapping between graph elements with identical labels.

local data structures in each component by a set of local rules. Furthermore, remote interactions can be described by distributed rules. In this case, all network nodes where parts of the interaction should take place, are incorporated into the network graphs of the rule. A *distributed graph rule* consists of a *network rule* and *local rules* for all network nodes which are preserved by the network rule. Simple examples of network graph rewrite rules are sketched in 4.2. A detailed description of the distributed graph transformation based on the double pushout approach can be found in [19, 2].

2.3. Integration into the ViewPoints framework

While application-internal states and behaviour of a single ViewPoint can be described by distributed graph transformation at the local level, distributed graph transformation at the network level is well suited for coordinating a distributed ViewPoint configuration.

In detail, the five slots of a single ViewPoint are formalized by the following aspects of distributed graph transformation:

- The style slot is represented by a local graph transformation system (i.e. a start graph and a set of rules).
- The domain slot is described by a local graph (in most

cases a single node attached with a domain label suffices).

- The specification slot is represented by a local graph. Please note that *formalizing rules* for transferring the specification in the original notation to its graph-based formalization as well as for transferring the graph-based formalization back to the original notation have to be specified by the developer of the ViewPoint template. However, these rules are in most cases easy to develop, since most requirements engineering methods are based upon diagrammatic notations. This aspect in investigated in detail in [10].
- The rules and actions of the work plan slot are formalized by distributed graph rewrite rules for accessing multiple ViewPoints, network graph rewrite rules for coordinating a ViewPoint configuration and local graph rewrite rules for accessing a single isolated ViewPoint.
- The work record slot is represented by a local tree the edges of which are labelled with applied development actions (i.e. applied graph rewrite rules defined in the work plan). Links have to be established from the tree nodes to nodes of the specification graph for tracking the effects of development actions.

Since our work concentrates on inconsistency management, this paper focuses on formalizing the specification slot as well as parts of the work plan slot (In-ViewPoint and Inter-ViewPoint check actions). The formalization of the other slots is beyond the scope of this paper. In the next chapter we will now present a non-trivial application of our framework.

3. A sample system: integration of software architecture and performance evaluation

In chapter 2 we have introduced the ViewPoints framework and in presented distributed graph transformation as its underlying formal semantics. Now we will sketch a sample application of our approach where architecture and performance views are integrated.

3.1. Combining architecture design and performance analysis

As a non-trivial application of our framework we present the work within the project QUAFOS (QUantitative Analysis of FOrmally specified distributed Systems). The goal of the QUAFOS project is the integration of the architecture description language Π [9] with the Queuing Specification and Description Language QSDL, an extension to the ITU's Specification and Description Language SDL for non-functional system evaluation [3]. Thus a performance model can be derived from a design model and maintained in parallel in order to analyze the architecture's performance-related properties.

Using the Π language, the functional behaviour of distributed components and their connections can be described as well as performance-related attributes of this architecture. For functional as well as performance-related evaluation, we use QSDL. The transformation of a QSDL-specification to an executable program for simulation and validation of the specified system is performed automatically by the tool QUEST [3].

We have identified ViewPoint templates for both Π and QSDL; they are defined in detail in [4, 9]. Relations between ViewPoints generated from the Π and QSDL ViewPoint templates are expressed by *Inter-ViewPoint check actions* in the work action slots of the Π and QSDL ViewPoint templates, respectively. In this contribution we will emphasize two kinds of Inter-ViewPoint relations:

- *Inter-ViewPoint use relations* describe a dependency relation between ViewPoints, such that if ViewPoint *A* uses ViewPoint *B*, some elements of *B* are necessary for *A* to function correctly.
- *Inter-ViewPoint correspondence relations* identify relating ViewPoint elements in a Π and a QSDL ViewPoint describing the same system part. These relations are used to express consistency / inconsistency checks and handling between the architecture and the performance view.

In the next section we will introduce a simple example how the ViewPoints framework is used to describe a software system from multiple views.

3.2. System representation

Let us consider developing a multimedia application globally distributed over the Internet. Figure 2 shows a snapshot of a system part after some development actions: it depicts a sample ViewPoint system representing a web browser connected to a cache component. Figure 2 comprises two Π ViewPoints representing distributed software components realizing the browser and the cache as well as a Π ViewPoint representing the remote use relation connecting both components. The system contains two ViewPoints encapsulating QSDL-systems representing performance models for the browser component and the cache component. The QSDL representation of the remote use relation is modeled by three ViewPoints encapsulating QSDL-systems for the medium as well as for a browser protocol instance and a cache protocol instance.

In the next two sections we will investigate the different kinds of Inter-ViewPoint relations and how they are represented by distributed graph transformation techniques.

3.3. Representation of Inter-ViewPoint use relations

While within distributed graph transformation interfaces are described explicitly, within the ViewPoint world interfaces are denoted implicitly. At the graph transformation

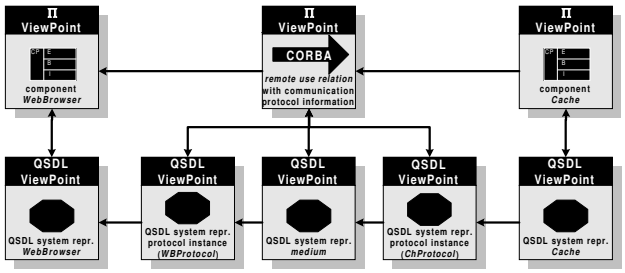

Figure 2 This network graph represents a ViewPoint system comprising architecture and performance related views of a sample system part. Squares representing *components* and an iconized arrow representing a *remote use relation* denote local graphs describing Π specifications. Accordingly, hexagons representing *QSDL systems* denote local graphs containing QSDL specifications. All horizontal network edges depict *Inter-ViewPoint use relations* and all vertical arrows depict *Inter-ViewPoint correspondence relations*.

level, interface graph elements in a local graph are specified explicitly by different grey shades (cf. Figure 1). At the ViewPoint level all elements involved with an Inter-Viewpoint relation are marked implicitly as interface elements.

In the following examples we will only depict the distributed graph representation of a ViewPoint's specification slot and work plan slot (In- and Inter-ViewPoint check actions, in particular). The formalization of all slots as sketched in section 2.3. is beyond the scope of this paper.

Let us look at a single Inter-ViewPoint use relation between two of the QSDL ViewPoints depicted in Figure 2 in detail. In order to establish a connection between the browser and its cache, the communication initiation has to be negotiated by the browser and its protocol instance. Thus the ViewPoint encapsulating the specification of the protocol instance imports a connection request signal which has to be provided by the ViewPoint containing the specification of the browser. Figure 3 shows the protocol's view of the established use relation.

3.4. Representation of Inter-ViewPoint correspondence relations

In Π various views upon a software component can be specified. Within the *interaction view* of a client component performance requirements wrt. remote use relations and remote server components can be stated (e.g., a value for maximum response time, etc.). Accordingly, within remote use relations a communication protocol with performance attributes can be selected. A remote use relation is only valid for a client component, if its performance attributes satisfy the component's performance requirements. As this distribution information is integrated into the Π language for evaluating architectures, the formalisms for noting performance attributes and requirements are based upon the QSDL sensor concept. A *sensor* can be placed anywhere in the QSDL-system and collects information about system events during the simulation of the QSDL-system (e.g., a counter for signal throughput of a signalroute). The interaction view's performance require-

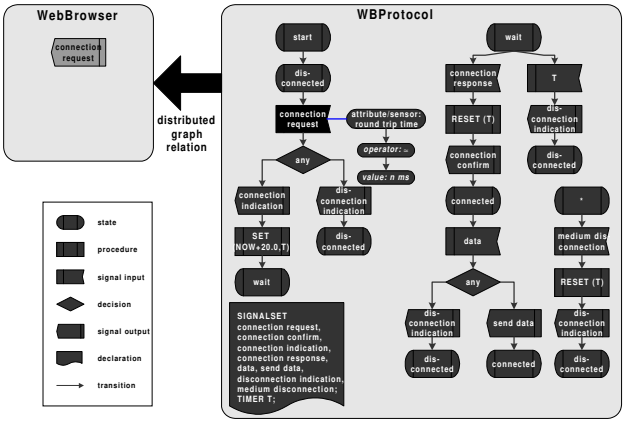

Figure 3 This distributed graph shows the established use relation from the view of the protocol instance. The right local graph represents a part of the specification slot of the ViewPoint encapsulating the protocol. The extended finite state machine [3] depicted here is constructed by the graph elements sketched within the legend. A connection request signal is imported and a distributed graph relation associates this node to an exported connection request node of the browser's local graph. Important for understanding this example is the issueing of a connection request, represented by the connection of signal output and signal input nodes labeled with a connection request signal.

ments are sensors extended by a compare operator and a concrete value (e.g., *response time* $\prec$ *10 ms*).

Bidirectional relations wrt. performance-related system properties between Π and QSDL ViewPoints can be identified: performance attributes grasped in the architecture design can be evaluated and also the results of the evaluation may feedback to new insights wrt. architecture requirements. This indicates that in addition to use relations (which represent unidirectional relations) we need correspondence relations between Π and QSDL elements (which represent relations in both directions).

Coming back to our sample architecture, a performance requirement of the web browser component may be an actual value for the round trip time of a connection request signal to the cache component. Figure 4 shows a distributed graph representing ViewPoints containing partial specifications of the Π remote use relation and the QSDL protocol instance. The correspondence relation is realized as a bidirectional graph relation, i.e. two use relations is both directions.

In this chapter we have shown how a sample ViewPoint system can be formalized by distributed graph transformation. Now we will investigate how distributed graph transformation helps to manage inconsistencies.

4. Dynamic inconsistency management

In highly interoperable and evolving systems based on heterogeneous collaborating ViewPoints developed by a variety of stakeholders with different needs and attitudes, the occurrence of inconsistent views is natural. We propose

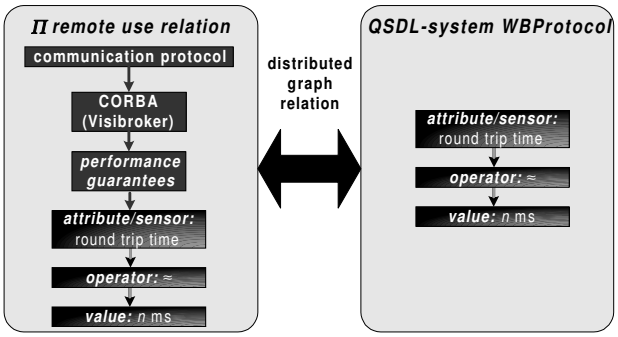

Figure 4 This distributed graph depicts a correspondence relation between the Π ViewPoint encapsulating a specification part of the remote use relation and the QSDL ViewPoint encapsulating a specification part of the protocol instance from the view of the Π ViewPoint. Thus a Π performance attribute is associated with a QSDL sensor. The *attribute/sensor* node, the *operator* node and the *value* node are colored as import/export nodes. Please note that the round trip time node represents a performance attribute within the Π context and a sensor within the QSDL context. The variable *n* represents a value for the round trip time.

a framework for inconsistency management which emphasizes the policy of *tolerating and living with inconsistencies*. Thus the information associated with inconsistent views is exploited to push forward the development process. Indeed, prohibiting a system change because it might lead to an inconsistency in cooperation with other system parts is in many cases counter-productive and might prevent the system from evolving.

While usually inconsistencies are defined as logical contradictions, we will adopt a more general definition [6]: we regard an inconsistency as any situation in which two parts of a specification do not obey some relationship that should hold between them. This notion of inconsistency complies naturally with the ViewPoints framework: firstly, we are able to handle inconsistencies between specifications described in different notations and secondly, inconsistencies are described explicitly in terms of relations between ViewPoints.

4.1. Inconsistency management rules by distributed graph transformation

According to the inconsistency management model presented in [15, 6] inconsistency management consists of three activities: *inconsistency detection*, *inconsistency classification*, and *inconsistency handling*.

As we have defined inconsistencies explicitly in terms of relations that should hold, an inconsistency is *detected* if a relationship which should hold between specification elements is found not to hold. Two types of graph rewrite rules are used for inconsistency detection:

- *consistency check*: the left hand side rule graph models a relation that should hold while the right hand side rule graph is identical to the left one,
- *inconsistency check*: the left hand side rule graph models a relation that should not hold while the right hand side rule graph is identical to the left one.

If a consistency check cannot be applied (i.e. no graph match from the left hand side rule graph into an actual distributed graph representing a configuration of ViewPoints can be found) or an inconsistency check can be applied, an inconsistency in the actual ViewPoint configuration is unveiled. Both types of checks can be realized as In-ViewPoint checks and Inter-ViewPoint checks.

Inconsistency classification focuses on identifying the kind of inconsistency that has been detected. Generally, the inconsistency management model proposes to categorize inconsistencies according to their cause (e.g., syntactic or semantic inconsistencies). It also leaves room for user-defined inconsistency kinds prescribed by a method engineer or tool designer. Within the scope of our sample application integrating architecture design and performance evaluation, we found the distinction between two kinds of inconsistencies useful: *minor inconsistencies* more at the level of an omission or typo which can be handled automatically in many cases and *major inconsistencies* which represent severe design errors or even conflicts between stakeholders.

Inconsistency handling focuses on acting in the presence of inconsistencies. The following five strategies to handle a detected inconsistency can be identified: *ignore*, *delay*, *circumvent*, *ameliorate*, and *resolve* (cf. [6] for details). Two classes of graph rewrite rules to represent ameliorate and resolve actions can be distinguished:

- *consistency check actions*: a consistency check combined with a different right hand side rule graph (e.g., development actions within the evolution of the ViewPoint configuration),
- *inconsistency check actions*: an inconsistency check combined with a different right hand side rule graph (e.g., actual resolve actions).

In [10] a detailed classification of (in)consistency checks and actions can be found. A process model for controlling delay and ameliorate actions with the help of the work record slot is ongoing work. We will now present some examples for check actions within the scope of the sample application introduced in chapter 3.

4.2. Minor and major inconsistencies

Coming back to our sample web browser application, an inconsistency wrt. the correspondence relation depicted in Figure 4 may be a missing QSDL representation of the distribution aspect between the web browser and the cache component. Since the QSDL counterpart of a remote use relation has a rather static nature (a QSDL-system representing the medium and a set of QSDL-systems representing protocol instances), this inconsistency can be fixed instantly at the network level. Figure 5 shows a sample inconsistency check action. However, as Figure 5 only

depicts a network graph rewrite rule, it only realizes an incremental inconsistency resolution, and more rules adjusting the local structures of the involved ViewPoints will be necessary. As both Π and QSDL possess textual as well as graphical specification styles, these graph rewrite rules can be designed as easily as the rule shown in Figure 5

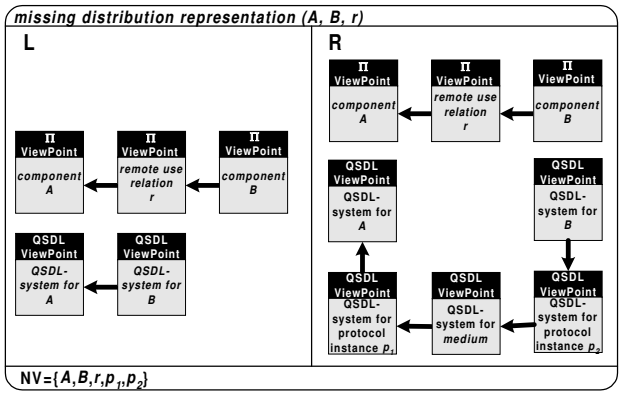

Figure 5 This network graph rewrite rule represents a minor inconsistency check action. Variables for node attributes are given by the set NV, and the variables considered as rule parameters are listed in the rule's head after its name. The rule can be applied to a sample network graph similar to the one depicted in Figure 2 (without the QSDL ViewPoints *WBProtocol*, *medium*, *ChProtocol* and without considering correspondence relations).

However, not all inconsistencies allow automated handling. There may exist major ones representing severe design errors the resolution of which require a manually planned strategy.

The goal of creating and simulating a performance model is to gain performance-related information about distributed components and remote use relations of the architecture design. Thus an obvious consistency check may be to determine actual performance values of a remote use relation by simulating a corresponding QSDL-system and to examine if these performance values satisfy the client component's performance requirements. A sample consistency check referring to the round trip time can be realized by a graph rewrite rule containing the graph depicted in Figure 4 as its left and right hand side rule graphs, assuming that the actual round trip time matches with the required one. If no graph match into the actual ViewPoint configuration satisfying the required conditions wrt. the variable n for the round trip time value can be found, an inconsistency is unveiled.

There are several ways to resolve such an inconsistency. One is to introduce faster protocols and / or media. If such an option is not available in such systems usually additional caching schemes are adopted or proxy servers are introduced. To choose from the various alternatives is a conscious design decision and cannot be automated. We consider the alternative to introduce a remote use relation comprising a different communication protocol into the architecture (cf. Figure 6). Of course, the performance model has to be maintained in parallel by appropriate rules and a substitution of the communication protocol may lead to further inconsistencies.

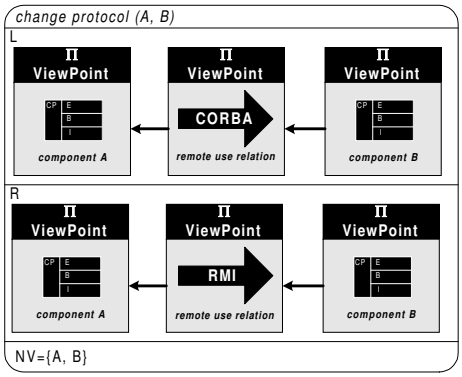

Figure 6 A network graph rewrite rule initiating a major inconsistency check action: the remote use relation with the CORBA protocol is replaced by a remote use relation realized with Java RMI. This rule can be applied to a sample network graph similar to the one depicted in Figure 2 (without considering correspondence relations).

5. Conclusion and further work

In this contribution we have shown that distributed graph transformation is a natural formalism to serve as underlying semantics of the ViewPoints framework. We have emphasized the various kinds of interactions between related distributed ViewPoints, thus leading to analysis of inconsistent views by explicitly representing In- and Inter-ViewPoint relations. Our framework for inconsistency management proposes the policy of tolerating and living with inconsistencies. Since our work is focused on requirements engineering we have sketched how inconsistency management can be used for analyzing and evaluating requirements. We have introduced a sample application of our framework investigating the integration of architecture design and performance prediction, thus concentrating on non-functional requirements. This example shows nicely that both ViewPoint specification as well as inconsistency management can be realized homogeneously by a single formalism: distributed graph transformation.

Our approach provides several benefits:

- In order to allow for tool support and rigorous mathematical reasoning about correctness / compliance, a formalization of the ViewPoints framework is necessary.
- Since most methods in requirements engineering are graphics-based, a graph-based formalization is natural. Moreover, text-based formalisms have problems with representing methods which make intensively use of graphical / structural information.
- Graphs combine intuitive usability with a formal basis. The level of formality visual to the user is highly scalable. The method designer creating a ViewPoint template may work at a more detailed level using the features of our approach to its full extend while the method user (requirement engineer) operating on ViewPoint instances

may work at a more abstract and symbolic level.

- Graph transformation is well suited to represent the structure of an entire ViewPoint, thus a single formalism suffices to represent all ViewPoint slots (cf. section 2.3.).
- The distinction of network and local level within distributed graph transformation furthers a clear and vivid visual impression. For example, graph transformation at the network level can be employed to describe dynamic reconfiguration of distributed ViewPoint configurations, while graph transformation at the local level can be used to model evolving data within single ViewPoints.
- Distributed graph transformation provides a modularization concept for graphs and graph transformation systems (cf section 2.2.). This makes it possible to describe the interfaces of a ViewPoint and the various Inter-ViewPoint relations in a structured way. Further, it helps to manage the complexity of large graphs which will be common in realistic and industry-relevant projects. Wrt. tool support, the module concept also reduces the complexity of the graph matching problem.
- Inconsistencies can be defined in terms of relations that should be handled by distributed graph rewrite rules. Rules are very natural for this task, because they support an if-then style of formulating inconsistency handling. If a certain inconsistent situation occurs, then the following action should be performed. Rules do not prescribe a certain control flow, but the their application order is dependent of the ViewPoint development. This suits the policy of handling inconsistencies (cf section 4.).

In this paper we have elucidated only a single aspect of the ViewPoint framework: we have formalized the specification slot as well as part of the work plan slot and we have sketched how inconsistency management can be realized with the same formalism on top of this. A formalization of the entire ViewPoints framework by distributed graph transformation including a greater case study is ongoing work. Currently we are working on an implementation of our ideas which will be based on the current AGG - machine [20].

6. References

[1] Baumeister, M., "Attribute grouping: emulating meta models without instantiation", *Proc. OOIS'96*, London, UK, 1996.

[2] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., and Löwe, M., "Algebraic Approaches to Graph Transformation", in Rozenberg, G. (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 1 Foundations, pp 163 - 245, World Scientific, Singapore, 1997.

[3] Diefenbruch M., Hintelmann, J., and Müller-Clostermann, B., "The QUEST-Approach for the Performance Evaluation of SDL-Systems", *Proc. IFIP TC6/6.1 Int. Conf. on Formal Description Techniques IX*, Kaiserslautern, Germany, 1996.

[4] Diefenbruch, M. and Meyer, T., "On Formal Modelling and Verifying Performance Requirements of Distributed Systems," *Proc. 2nd World Conference on Integrated Design and Process Technology*, Austin, USA, 1996.

[5] Easterbrook, S., Finkelstein, A., Kramer, J., and Nuseibeh, B., "Coordinating Distributed ViewPoints: the anatomy of a consistency check", *Concurrent Engineering: Research & Applications*, Vol. 2, CERA Institute, USA, 1994.

[6] Easterbrook, S. and Nuseibeh, B., "Using ViewPoints for Inconsistency Management", *BCS/IEE Software Engineering Journal*, pp. 31-43, 1996.

[7] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M., "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development", *Int. Journal of Software Engineering & Knowledge Engineering*, vol. 2(1), 1992.

[8] Finkelstein, A. and Sommerville, I., "The Viewpoints FAQ", *Software Engineering Journal*, vol. 11 (1), pp. 2-4, 1996.

[9] Goedicke, M. and Meyer, T., "Formal Design and Performance Evaluation of Parallel and Distributed Software Systems", *Proc. International Symposium on Software Engineering for Parallel and Distributed Systems*, Kyoto, Japan, 1998.

[10] Goedicke, M., Meyer, T., and Piwetz, C., "A High Level Design Framework for Software Systems", *Journal of the SDPS*, Vol. 2/98, Society for Design and Process Science, 1998.

[11] Hunter, A. and Nuseibeh, B., "Managing Inconsistent Specifications: Reasoning, Analysis and Action", *ACM Transaction on Software Engineering and Methodology*, ACM Press, 1998.

[12] ISO/IEC, "ISO/IEC 10027 Information Technology - Information Resource Dictionary System (IRDS) - Framework", *ISO/IEC International Standard*, 1990.

[13] Nuseibeh, B., "A Multi-Perspective Framework for Method Integration", *PhD. Thesis*, Imperial College, London, 1994.

[14] Nuseibeh, B., "To Be and Not to Be: On Managing Inconsistency in Software Development", *Proc. 8th International Workshop on Software Specification and Design*, Schloss Velen, Germany, IEEE Computer Society Press, pp. 164-169, 1996.

[15] Nuseibeh, B., "Computer-Aided Inconsistency Management in Software Development", *technical report*, Department of Computing, Imperial College, London, UK, 1994.

[16] Pohl, K., "Process-Centered Requirements Engineering", Wiley, 1996.

[17] Pohl, K., Nuseibeh, B., Finkelstein, A., and Kramer, J., "Interoperability: a system specification perspective", in Krämer, B. and Papazoglou, M. (eds.), *Information Systems Operability*, Wiley, 1997.

[18] Spanoudakis, G., Finkelstein, A., and Till, D., "Overlaps in Requirements Engineering", *to appear*.

[19] Taentzer, G., Fischer, I., , Koch, M., and Volle, V., "Distributed Graph Transformation with Application to Visual Design of Distributed Systems", to appear in Rozenberg, G. (ed.), *Graph Grammar Handbook 3: Concurrency & Distribution*, World Scientific,1999.

[20] Taentzer, G., Ermel, C., and Rudolf, C., "AGG-Approach: Language and Tool Environment", to appear in Rozenberg, G. (ed.), *Graph Grammar Handbook 2: Specification & Programming*, World Scientific, 1999.

[21] Wiels, V. and Easterbrook, S., "Management of evolving specifications using category theory", *Proc. 13th IEEE Int. Conf. on Automated Software Engineering*, Honolulu,USA, IEEE, 1998.