

# WS5. The Eighth International Workshop on Component-Oriented Programming (WCOP 2003)

Jan Bosch<sup>1</sup>, Clemens Szyperski<sup>2</sup>, and Wolfgang Weck<sup>3</sup>

<sup>1</sup> University of Groningen, Netherlands

Jan.Bosch@cs.rug.nl - <http://www.cs.rug.nl/~bosch/>

<sup>2</sup> Microsoft, USA

CSzypers@microsoft.com - <http://research.microsoft.com/~cszypers/>

<sup>3</sup> Oberon microsystems, Switzerland

Weck@oberon.ch

**Abstract.** This report covers the eighth Workshop on Component-Oriented Programming (WCOP). WCOP has been affiliated with ECOOP since its inception in 1996. The report summarizes the contributions made by authors of accepted position papers as well as those made by all attendees of the workshop sessions.

## 1. Introduction

WCOP 2003, held in conjunction with ECOOP 2003 in Darmstadt, Germany, was the eighth workshop in the successful series of workshops on component-oriented programming. The previous workshops were held in conjunction with earlier ECOOP conferences in Linz, Austria; Jyväskylä, Finland; Brussels, Belgium; Lisbon, Portugal; Sophia Antipolis, France; Budapest, Hungary; and Málaga, Spain.

WCOP96 had focused on the principal idea of software components and worked towards definitions of terms. In particular, a high-level definition of what a software component is was formulated. WCOP97 concentrated on compositional aspects, architecture and gluing, substitutability, interface evolution, and non-functional requirements. WCOP98 had a closer look at issues arising in industrial practice and developed a major focus on the issues of adaptation. WCOP'99 moved on to address issues of structured software architecture and component frameworks, especially in the context of large systems. WCOP 2000 focused on component composition, validation and refinement and the use of component technology in the software industry. WCOP 2001 addressed issues associated with containers, dynamic reconfiguration, conformance and quality attributes. WCOP 2002 has an explicit focus on dynamic reconfiguration of component systems, that is, the overlap between COP and dynamic architectures.

WCOP 2003 had been announced as follows:

COP has been described as the natural extension of object-oriented programming to the realm of independently extensible systems. Several important approaches have emerged over the recent years, including CORBA/CCM, COM/COM+, J2EE/EJB, and most recently .NET. After WCOP'96 focused on the fundamental terminology of COP, the subsequent workshops expanded into the many related facets of component software.

WCOP 2003 will emphasize the dynamic composition of component-based systems and component-oriented agile development processes. A typical example of dynamic, i.e. run-time, composition of systems is the notion of web-services, but also in other domains and contexts this trend can be identified. This requires clearly specified and documented contracts, standardized architectures, specifications of functional properties and other quality attributes, and mechanisms for dynamic discovery and binding of services. A service is a running instance (all the way down to supporting hardware and infrastructure) that has specific quality attributes, such as availability, while a component needs to be deployed, installed, loaded, instantiated, composed, and initialized before it can be put to actual use. The commonalities and differences between service and component composition models are interesting and a proposed focus of this workshop.

Agile development processes can benefit from component-based development in that use of existing or of-the-shelf components reduces the amount of required development effort and gives quick results early in the process. This requires deciding early in the process which specific architectures, standards, interfaces, frameworks or even components to use. Unfortunately, such early decisions contradict general agility, as promoted, for instance, by extreme programming, because reconsidering such fundamental decisions later in the development process comes at considerable cost but may be unavoidable at the same time. On the one hand, one may end up with developing a new component instead of deploying an of-the-shelf component as planned earlier. On the other hand, a chosen architecture or interface may prove inadequate later in the process, when additional requirements are considered for the first time. As a source of delay and extra cost this easily puts the entire development at risk. Can these contradictions be dealt with?

COP aims at producing software components for a component market and for late composition. Composers are third parties, possibly the end users, who are not able or willing to change components. This requires standards to allow independently created components to interoperate, and specifications that put the composer into the position to decide what can be composed under which conditions. On these grounds, WCOP'96 led to the following definition:

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

Often discussed in the context of COP are quality attributes (a.k.a. system qualities). A key problem that results from the dual nature of components between technology and markets are the non-technical aspects of components, including marketing, distribution, selection, licensing, and so on. While it is already hard to establish functional properties under free composition of components, non-functional and non-technical aspects tend to emerge from composition and are thus even harder to control. In the context of specific architectures, it remains an open question what can be said about the quality attributes of systems composed according to the architecture's constraints.

As in previous years, we could identify a trend away from the specifics of individual components and towards the issues associated with composition and integration of components in systems. While the call asked for position papers on the relationship to agile development processes, we did not receive any such papers. An emphasis on anchoring methods in specifications and architecture was noticeable, going beyond the focus on run-time mechanisms in the previous year. Sixteen papers by authors in nine countries were accepted for presentation at the workshop and publication in the workshop proceedings; three submissions were rejected. About 35 participants from around the world participated in the workshop. The workshop was organized into four morning sessions with presentations, one afternoon breakout session with four focus groups, and one final afternoon session gathering reports from the breakout session and discussing future direction.

## 2. Presentations

This section summarizes briefly the contributions of the sixteen presenters, as grouped into four sessions, i.e. Specification and Predictable Assembly, MDA and Adaptation, Separation of Concerns, and, finally, Dynamic Reconfiguration.

### 2.1 Specification and Predictable Assembly

The first paper, presented by Sven Overhage, is a proposal towards a standardized specification framework covering five specification perspectives, following the UDDI example of grouping these into colored specification pages. In particular, the following five colors are proposed: (i) general and commercial information (White Pages), (ii) component classifications (Yellow Pages), (iii) component functionality (Blue Pages), (iv) component interfaces (Green Pages), and (v) implementation-related information (Grey Pages). Each of the last three perspectives is addressed from the static, operations, and dynamic views. Blue pages cover a lightweight ontology over concepts such as entities, tasks, processes, etc. Green pages specify provided and required interfaces, assertions, and use temporal logic for protocols. Grey pages use QML (quality modeling language) to specify extra-functional properties. As future directions, the paper proposes (i) a meta-model to integrate different notations, (ii) standardization, (iii) the implementation of specification tools and repositories, and (iv) the development of a multi-dimensional component type system for compatibility tests.

The second paper on contracts and quality specifications of software components was presented by Ralf Reussner. Starting from a situation of wide-spread interest in quality-of-service (QoS) and work on QoS specification languages (such as QML), the paper identifies two problems: (i) a component has to interrogate the environment for QoS in order to offer QoS (thus a need for requires-interfaces for QoS) and (ii) a component vendor has to select which QoS properties to specify in provides and requires interfaces. (Reussner noted that a requires interface can be seen as a precondition and a provides interface as a postcondition to the component assembly function.) QoS is context dependent: (i) there is no single value for QAs of a component; (ii) a component exposes different quality in different contexts; and (iii) the component vendor cannot specify a QA as a constant. These observations lead to the concept of deployment-time interfaces for components that reflect context properties. Deployment-time evaluation of a parameterised contract can be used to perform static interoperability checks. The authors found that many such parameterised contracts can be generated by code analysis (and guessing transition probabilities in resulting Markov models), still yielding reasonably accurate QoS predictions.

The third paper, presented by Franz Puntigam, discusses the use of state information in statically checked interfaces. The idea is to equip types with state variables and methods with pre- and poststates. The paper includes a series of small examples, including concurrency scenarios such as dining philosophers. The system supports static type checking and separate compilation. Several simultaneous references are supported. Alternative When-clauses can be used to state complex conditions. Dependent state information and exclusive use can be handled. In application to the area of component-based systems, the approach can be used to partially capture non-trivial protocols. A specific application discussed (but not fully resolved) in the paper is to determine safe points and mechanisms for type-state recovery for component hot-swapping. The last paper of this session, presented by Nicolae Dumitrascu, presents a methodology for predicting the performance of component-based applications. As a primary motivator, the authors identify the cost benefits of detecting problems early in a development effort. Their concrete examples are set in the context of .NET. Performance profiles of the used components are used to predict the performance of a component assembly (not of a .NET assembly); a difficulty is to properly account for run-time behavioral interaction. The presented approach predicts performance of an application taking into account: application design, resource design, assembly performance profile, and connection performance profiles. Extended queuing models are used as performance models.

## 2.2 MDA and Adaptation

In the first presentation of this session, Seung-Yun Lee presented their paper on an MDA-based COP approach. In their previous work, the authors had worked on CBD support: (i) generating EJBs from specifications, extracting EJBs from Java applications, extracting EJBs from databases; (ii) adapting and composing

EJBs; (iii) assembling components based on architecture diagrams; (iv) deploying assemblies; and (v) testing interfaces, running applications, and measuring performance. However, they identified the problem that CBD does not directly support interoperability and integration across different platforms. This leads to the viewpoint of model-driven architecture (MDA) as an evolutionary refinement of CBD. MDA works over the core features of a model repository, a set of model transformations, and a meta editor. Further features are required to support CBD: component identification from business modeling results and an architecture-centric approach. To enable this integration, the presented research project aims to deliver a full stack of MDA-based CBD tools.

In the second presentation Olivier Nano looked into ways to using MDA to integrate services in component platforms. Most current component platforms offer services such as persistency, transactions, or authentication. However, components are frequently required to target used services explicitly, which requires knowledge as to where to call a service and what arguments to pass. The claim is that the “join points” where service calls need to be inserted are independent of any component platform. To introduce systematic join points, a detailed meta-model of the invocation process is used with four interception points on the forward path (send, accept, receive, execute) and two on the return path (send-Back, return). Service integration is then described as interceptions along these points defined by the behavioral meta-model. Merging is performed by proved merging rules that are commutative and associative. The mapping to a concrete component platform is not a problem where such a mapping is surjective. In a next step, Nano proposed to try the construction of the model as a UML2 profile and to implement the transformation with QVT.

The third presentation was delivered by Abdelmajid Ketfi, focusing on dynamic interface adaptability (<http://www-adele.imag.fr/~ketfi>). The approach is to introduce dynamic adaptation frameworks for EJB and OSGi. Strategies and Monitoring (of material resources) form inputs to an adaptation manager, which modifies non-functional services that interact with a components functional services; both functional and non-functional services provide feedback to the adaptation manager. An abstract application model is used (i) to extract meta-level information from an application and (ii) to describe an application at the meta-level. A prototype for Java has been developed. Dynamic interface adaptability poses a number of issues, including The need to handle syntactic dependencies and dynamic availability. Ketfi proposes to reify connections, that is, to introduce connectors as first-class entities (“aware connectors”). As future work, Ketfi named the dynamic adaptation of extra-functional services. After the presentation, he answered the question as to how dynamic Adaptation can be in his system by stating that he can adapt in a running system in principle, but that this depends on the concrete component platform used.

The fourth and final presentation of this session, presented by Gustavo Bobeff, looked into the “molding” of components using program-specialization techniques. The context of this approach is component development (where the stages of production, assembly/configuration, and execution are distinguished). Bobeff

observes that today adaptation is limited to the superficial level of adapting interfaces to the context of use, where implementations remain unchanged. His goal is to adapt the implementation to the context of use, where the idea is to rely on program transformation, using one of two particular techniques: partial evaluation or slicing. To be successful, Bobeff claims that the trick is to capture specialization opportunities at production time, moving deliverables from components to component generators that carry self-contained specialization knowledge. The component-specialization process then works by invoking component generator with context arguments at deployment time. The result is that deployed components have an adapted interface as well as implementation (“deep adaptation”). The question from the audience as to how specializing components would differ from specializing programs was addressed by noting that the individual suppliers of components do not communicate, but that the specialization should yet support the integration scenarios where multiple such components meet. (The created knowledge gap is closed by specializing at deployment and not at production time.)

### 2.3 Separation of Concerns

The first presentation of this session was by Paolo Falcarin. It dealt with technology supporting dynamic architectural changes for distributed services, in particular replacement of the middleware and dynamic evolution. For this, source code should be checked against an architectural description. In JADDA architecture is described in an XML-based language. At runtime, applications register with a system administrator component and receive such an architecture description in return. Connections to components are made using a name server. To replace the middleware, the administrator will distribute new architecture files and bindings are reestablished calling the name server again. When applications reconnect, is under their individual control and they can make sure to be in a consistent state at the time.

The second presentation, given by David Lorenz, discussed how technology of aspect-oriented programming can be used to em unweave legacy systems. Components can be em unplugged by using an em around advice that redirects calls to certain methods and handles them as an aspect rather than a component call. The still challenging part of this work is aspect interaction as it is with the usual forward AOP.

Third, William Thomas talked about the need to improve verification techniques so that they extend to creating systems from existing components. Some policies must be enforced to create verifiable systems, but these policies often will depend on the application domain. Considerations are in particular when to activate mediators, the level of access to the applications state, and actions to be performed by the mediator. Also, mechanisms for detection and resolutions of policy conflicts need to be investigated.

Finally, Marek Prochazka presented a paper about the interaction of components and transactions. Transactions are a key service to components but each component framework and application server handles things differently. To

open the middle ground between component-controlled and container-controlled transactions, the proposed Jironde framework introduces separate controllers that can be associated with components. Controllers decide about transactional functionality.

## 2.4 Dynamic Reconfiguration

**Dynamic Reconfiguration** The fourth and final presentation session was concerned with the *dynamic reconfiguration* of component-based systems. The challenge of dynamic reconfiguration is to balance the freedom allowed after the deployment of the system versus the functionality and properties of the system that should remain available in the face of reconfigurations.

As the other sessions, also this session consisted of four papers. The first paper was presented by Kris Gybels and presents an approach to enforce feature set correctness. The paper starts out from a generative programming approach and extends this towards dynamic generative programming. It views a software system as consisting of a set of features and a set of composition rules. During reconfiguration, the existing set of composition rules should not be violated. The author identifies that, due to the interaction between the base language and the meta language describing the composition rules, one requires a *linguistic symbiosis* approach. In the presentation, the author presented a new version of the SOUL language that achieves a level of linguistic symbiosis with Smalltalk.

The second paper was presented by Vania Marangozova and discusses the use of replication as a means to achieve high quality component-based systems, especially from an availability perspective. The motivation is that one, on the one hand, would like to reuse functionality of different applications and, on the other hand, would like to use these applications in different execution contexts. A reusable and effective approach to high availability needs, thus, to be separated from the application code. The approach proposed by the author uses a simple component model for business applications, an application independent protocol design model and a composition model that facilitates the composition of the business application and replication protocol. The protocol controls the application, but the configuration is generated at system deployment. The advantages of the approach are the aspect separation between business and replication code, a component model of replication management and non intrusive approach to replication configuration. The disadvantages are the negative impact on performance and the, currently unclear, interaction with other system services.

The third presentation, by Mircea Trofin, presents a self-optimizing container design for enterprise Java beans applications. The paper addresses the problem that applications that run on a server cluster need to be modularized in order to support their evolution. On the other hand, especially fine-grained modularisation has, in most component interconnection standard implementations, a substantial negative impact on performance. The approach proposed in the paper focusses on optimizing containers. It uses call path and deployment

descriptor information, predetermines the services that are required and predetermines the state management requirements. Based on this, the approach is able to determine and define safe optimizations. The current implementation shows promising results in that response times are reduced with 20% to 40%.

The final paper in the dynamic reconfiguration session was presented by Jasminka Matevska-Meyer. The paper presents an approach to determine runtime dependency graphs and using this information to speed up runtime reconfiguration of component-based systems. The problem addressed by the paper is that static dependency graphs are overly negative in most situations, limiting the set of possible reconfigurations. The approach to enabling reconfiguration of component-based systems at runtime consists of three main steps. First, the running system is analyzed for a particular time interval with respect to the estimated reconfiguration duration. Second, it creates change-request specific runtime dependency graphs, based on specified (or derived) architecture and component protocols. Finally, it observes the running system at a particular time interval for the actual reconfiguration duration and then performs the reconfiguration. The key benefit of the approach are that the set of components affected by the reconfiguration is smaller, resulting in a larger set of services to be available during reconfiguration. This allows one to minimize the down-time of the system at reconfiguration.

### 3. Break-out Session

During the afternoon, the attendees were organized into break-out groups. The breakout groups addressed a number of topics, i.e. interface specification and MDA, connection of components, dynamic reconfiguration, and separation of concerns. At the end of the day, the workshop was concluded with a reporting session where the results of the break-out sessions were presented. Below, the results of the sessions are summarized briefly.

#### 3.1 Interface specification formalisms and MDA

The first breakout group discussed issues around interface specification and model-driven architecture (MDA). Arne Berre served as the scribe for this group and contributed the material for this subsection.

The group structured the topics to be discussed into three areas: (i) component specification frameworks, (ii) the interface model, and (iii) domain-specific standards.

**Component specification frameworks** The group considered it an advantage if component specifications could be based on (extended from) standardized specification approaches such as the standards from OMG: UML 2.0 with its new component model (with a conceptual basis in the CCM), UML Quality/Fault-tolerance profile standard for QoS specifications, and classification models, such as the UDDI white/yellow/green pages. Specifications should be goal driven,

meaning that there should be a clear intention by each part of a specification to support its intended usage.

Three main usages were identified:

*Component retrieval* (to identify components – or candidates, respectively - based on component classifications like the yellow pages from UDDI) – In so doing, there is a need for similarity metrics on components, which needs more research.

*Component Interoperability* (for checking if two components can work together) – There is also a need for defining the semantics/meaning of concepts and method names, (ref. later on blue pages).

*Component adaptation, configuration and generation* – There is also a need to have enough information to be able to create adapters, to reconfigure components and to generate platform specific models. A question is which information needs to be attached to the interface.

**The interface model** An interface model for business components from a working group of the German Society of Computer Science (GI) was accepted by the group as a principal basis for structuring component specifications. It builds upon the UDDI specification approach for (web) services and uses green pages to specify interface definitions and protocols, white pages for general business information, and yellow pages for classification information. The two additional “color” dimensions are “blue” pages for domain/business information and “gray” pages for quality information.

Component composition and logical structure/architecture is also important to be able to integrate into the specification framework, but this is the focus of another Working Group, so it was not discussed further.

A graphical overview and further information can be found from the works of the WG on Business Information Systems/GI (<http://www.fachkomponenten.de> (also in English)).

The MDA approach (From OMG Model Driven Architecture) could build upon these by capturing the color aspects in UML, with appropriate stereotypes/UML profiles. Subtyping should be possible in each of these (colors) dimensions. More research into static checking of protocol subtypes is required. The new UML 2.0 is viewed as a good basis for this, but further standardization is needed here, such as a platform independent type system, links between the Component model and Activity model, and profiles for the various “color” dimensions. The before suggested Interface model for business components (Fachkomponenten), might not yet cover all the aspects and have the precision required from MDA-based model transformation and code generation.

Specifications must be usable from tools, through a representation in XML/XMI. Hopefully the new version of XMI (version 2) will improve on the compatibility aspects that has been a problem so far.

Component specifications are dependent on the deployment/run-time context, and thus requires parameterized contracts, and the handling of dependency between required and provided interfaces. A component will typically have mul-

multiple interfaces/roles (deployment time interfaces) for the various contracts it is responsible for/involved in.

**Domain-specific standards** In the context of discussing specification formalisms there was also a discussion on the conditions for such formalisms to come into practical usage.

It is not possible to mix and match components arbitrarily, it is necessary to have a common foundation in the understanding of the semantics/meaning of the concepts involved in the component interfaces and protocols. This can only be done by creating domain specific models/ontologies of concepts that can be referred to in the interface specifications.

An example that was discussed is the General Ledger domain, which has been addressed by the GI working group and was suggested as an OMG component standard (though unsuccessful so far). In the specification of the semantics it is for instance important to state whether a balancing operation is done according to US GAAP or IAS accounting principles.

Emerging component markets not only depend on technical standards such as specification formalisms, but also on market conditions that encourage companies to be involved in the agreement on such domain models.

For the companies that currently have a major presence in the market it can be difficult to find the incitement to take part in the promotion of open standards/ open models in their domain. This is more obvious as an opportunity for second generation companies.

This also relates to the more general problem of encouraging reuse, even within companies, with the NIH (Not invented here) syndrome etc.

### 3.2 How to Connect Components?

The second breakout group discussed issues regarding the definition of the connection of components prior to their deployment. The group's scribe was Markus Lumpe, who contributed the material for this subsection. In addition, the following attendees participated in this group: Selma Maliugui, Jorgen Steengaard-Madsen, Jacques Noyé, Seung-Yun Lee, and Gustavo Bobeff. The main subjects discussed were: (a) soundness of component connection, (b) architectural and behavioral aspects of connector specification, and (c) language support.

Component connection means that one connects required with provided services of components. When using component connectors, it was argued that it is necessary to check the “soundness of connection”, which guarantees that (a) all involved types match, (b) the behavioral characteristics match, and (c) any additional constraint (e.g. real-time demands) is satisfied. Soundness checks must be done at both compile-time and runtime. However, a complete soundness check might not be feasible. Therefore, the breakout group members opted for an approach that allows for “partially erroneous connector specification” – a scheme that requires additional runtime checks, but allows a relaxed specification of component connections.

In order to address architectural and behavioral aspects of component connections, the members of the breakout group proposed to represent connectors as components. This will allow for a homogenous view of a component-based system and will facilitate reasoning about composition. In fact, one can think of a component-based system as a triangle. At the top node a component-based system consists only of one component that actually represents the whole architecture of the corresponding application. A zoom operation can then be used to decompose the application, that is, architectural views will incrementally be replaced by behavioral views. At the bottom only behavioral aspects remain.

To facilitate the specification of component connections, the members of the breakout group finally agreed on needing special language support. More precisely, there is a need for both a component language and a composition language. While the former should provide abstractions to specify components, the latter has to provide abstractions to specify component connections (i.e., component composition). Ideally, one would like to have a unified language that incorporates all aspects of component-based software development.

In the succeeding discussion in the plenum, it was noted that a soundness check of component connection may not be computable. A solution to this problem could be an approach that uses a combination of configurable compile-time and runtime checks.

### 3.3 Dynamic Reconfiguration of Systems (Post-deployment)

The third breakout group covered dynamic reconfiguration of component systems in the post-deployment phase. The group scribe was Jean-Guy Schneider, who contributed the material for this subsection. In addition, the following attendees participated in this group: Robert Bialek, Paolo Falcarin, Abdelmadjid Kefti, and Jasminka Matevska-Meyer.

To facilitate discussion of issues, the group first defined some terms: (i) a component is a state-full first class entity that has a well-defined interface (consisting of both provided and required services) and needs to be instantiated before use and (ii) a configuration to describe both the connections between provided and required services as well as the interaction protocols used for intercomponent communication.

The following questions were addressed during the breakout session:

- What is dynamic reconfiguration? What kinds of reconfiguration exist?
- Why is dynamic reconfiguration needed? Are there any examples which clearly indicate the need for dynamic reconfiguration?
- When is dynamic reconfiguration performed?
- Who does dynamic reconfiguration?
- How is dynamic reconfiguration performed?

*What is dynamic reconfiguration?* In essence, the workout group agreed that (i) a change of connections and bindings between components and (ii) a change of the respective communication protocols are the two main reconfiguration mechanisms. The group also observed that a strict separation between application

logic and (re)-configuration logic is necessary. Taking this into consideration, "patching" a component (i.e., adding new or changing existing behavior) should not be considered as a reconfiguration, although it will probably have an effect on the behavior of the overall system.

*Why is reconfiguration of a running system needed?* Various reasons were brought forward: (i) a change of some third-party component may trigger a reconfiguration in the remaining system, (ii) a change in the deployment environment requires an adaptation of communication protocols (e.g., change in network protocols), (iii) improving the overall performance of a system requires a different configuration, (iv) adding fault tolerance to a system, and (v) monitoring a running application.

*What are characteristic scenarios that require dynamic reconfiguration and cannot be implemented using "traditional" approaches?* The following list was defined during the discussion: (i) service adaptation in distributed systems, (ii) mobile-phone networks (in particular exchanging one base station for another), (iii) embedded systems, (iv) fault-tolerant systems (changing servers due to system load etc.), and (v) evolution of web-applications. What about personalization of (web-)services? Taking the separation of configuration and application logic into account, it was agreed that this should not be considered as an example of reconfiguration, but more as a case of application "patching."

*Who initiates dynamic reconfiguration?* It can either be an external entity or the system itself. In the latter case, access to the system environment is needed (e.g., to find out about loads of servers).

*When is a running system reconfigured?* Possible time points for reconfiguration can be derived from the system specification or by monitoring the running system and determining a "save" time point based on its execution state.

During the breakout discussion, the participants noted issues which were not directly related to any of the questions, but had to be addressed in the context of dynamic reconfiguration. Most important is the fact that any reconfiguration of a system needs to leave the system in a consistent (configuration) state. One way of achieving this goal is to view reconfigurations as transactions that can be rolled-back if the system does not end up in a consistent state. In the succeeding discussion in the plenum, it was noted that there is no real need to have transaction support for dynamic reconfiguration as this is simply one approach to ensure that any reconfiguration will leave the system in a consistent state. However, it is very probable that some form of reification support is needed for consistency checks. Finally, it was suggested that it would be possible to use concepts from aspect-oriented programming to incorporate support for unexpected configuration change.

### 3.4 Separation of Concerns

The fourth break-out group addressed various issues around separation (and composition!) of concerns. The groups scribe was Vania Marangozova, who contributed the material for this subsection.

An alternative session title: “How six researchers working in different domains define the key issues of the problem.” The groups participants were: Kris Gybels (Vrije Universiteit, Brussel, Belgium), working on languages; William Thomas (The MITRE Corporation, VA, USA), working on interception techniques in legacy systems for policies; Marek Prohazka (INRIA Rhne-Alpes, France), working on adding and configuring transactions; Olivier Nano (Universit de Nice-Sophia-Antipolis, France), working on adding system services to an existing system; Mircea Trofin (Dublin City University, Ireland), working on container configuration; and Vania Marangozova (INRIA Rhne-Alpes, France), working on adding and configuring replication.

The discussions in this group were organized around three subjects. In the first place, the participants tried to identify the key issues related to separation of concerns. In the second place, they tried to propose a classification of the approaches that handle these issues. In the third and last place, participants have tried to conclude on the feasibility and the applicability of separation of concerns. These three areas are detailed further in the following.

**Key Questions** To provide separation of concerns, the following three questions need to be addressed:

1. The What Question: What is a concern? Without a clear definition of what a concern is, it is very difficult to provide tools for separation of concerns!
2. The When Question: When are concerns separated? In particular, when does one think in terms of separated concerns? When are concerns composed? When is information about composed concerns available?
3. The How Question: How is separation of concerns achieved? What are the techniques?

*What is a concern?* There are two main approaches to defining the nature of concerns: the first one is adopted by the system-oriented community while the second approach is typical of language-oriented works. The system-oriented approach is interested in the modular management of system services such as transactions, security, fault tolerance, etc. A concern corresponds to a system service and the goal is to define the management of a system service separately from the business (non system) code, as well as separately from the management of other services. The language-oriented approach does not make any distinction between system and non-system concerns. It considers that every treatment may be considered as a concern. It follows the very generic definition given by Dijkstra. Given that, from a different point of view, the implementation of a system service can be considered as a business treatment, this general definition seems to be a better one.

*When are concerns separated?* To be able to manage separate concerns, it seems necessary to think in terms of such separate concerns already at design time. Separate concerns may be composed during different phases of the applications lifecycle. They can be composed statically, during design. They can also be composed during a configuration phase before the actual execution of

an application. Finally, there may be a need to change the composition of concerns during runtime. The moment of composition of concerns is related to the management of the (meta) information about these separate concerns. In fact, a system where composition is done once and is not subject to change does not need to keep the information of the concerns to be composed. On the contrary, if a system needs to undergo changes during execution, it will certainly need the information characterizing the separation of composed concerns.

*How are concerns separated?* As there are two definitions of what concerns are, there are also two approaches to manage separation of concerns. In the case of language-oriented works, concerns are expressed in terms of programming instructions and the separation techniques are related to code manipulation and transformation (AOP). As these techniques are working on a very detailed level, they are very powerful. However, the power of expression induces complexity. Another problematic point with this kind of techniques is that they suppose to be able to access and modify the source code. This is an issue since components are supposed to follow the black-box principle. The system-oriented approach has the opposite characteristics. It works on structural (architectural) entities and separation of concerns is more commonly obtained through interception at the boundaries of these entities. Compared to the language approach, there are fewer entities to consider, so it is simpler to manage. However, such structural models induce limitations on the set of concerns and separations that can be managed.

**Feasibility and Applicability (Open Questions)** In order to separate concerns, it is very important to identify hidden assumptions like assumptions on the execution system, the programming paradigm, etc. It seems very difficult to program a concern which is really generic and which does not have any dependencies on other concerns. Moreover, it seems natural for a programmer to specify constraints on other concerns! The group concluded that separation of concerns is not feasible in all cases. For example, in the cases of system services like transaction management and replication there is a need to modify the “other” code in a specific way in order to make the system work. Given this, to push the idea of separation of concerns further, there is a need to further consider the following two points:

- Identification of different types of concerns. We will need to define more precisely the concerns to be composed in a system in order to provide the adapted tools for their management.
- Identification of dependencies. For each type of concern, we will need to define the dependencies between this concern and other code. This will help understanding of concerns organization and the way they need to be composed with other code.

## 4. Final Words

As organizers, we can again look back on a highly successful workshop on component-oriented programming. We are especially pleased with the constantly evolving range of topics addressed in the workshops, the enthusiasm of the attendees, the quality of the contributions and the continuing large attendance of more than 30 and often as many as 40 persons.

We would like to thank all participants of and contributors to the eighth international workshop on component-oriented programming. In particular, we would like to thank the scribes of the break-out groups.

## 5. Accepted Papers

The full papers and additional information and material can be found on the workshop's Web site (<http://research.microsoft.com/~cszypers/events/WCOP2003/>). This site also has the details for the Microsoft Research technical report that gathers the papers and this report.

The following list of accepted papers is sorted by the name of the presenting author.

1. Bobeff, Gustavo; Noyé, Jacques (École des Mines de Nantes, France). "Molding components using program specialization techniques"
2. Dumitrascu, Nicolae; Murphy, Sean; Murphy, Liam (U College Dublin, Ireland). "A methodology for predicting the performance of component-based applications"
3. Falcarin, Paolo; Lago, Patricia; Morisio, Maurizio (Politecnico di Torino, Italy). "Dynamic architectural changes for distributed services"
4. Gybels, Kris (Vrije U Brussel, Belgium). "Enforcing feature set correctness for dynamic reconfiguration with symbiotic logic programming"
5. Ketfi, Abdelmadjid; Belkhatir, Noureddine (Domaine U Grenoble, France). "Dynamic interface adaptability in service oriented software"
6. Kojarski, Sergei; Lorenz, David H. (Northeastern U, USA). "Unplugging components using aspects"
7. Lee, Seung-Yeon; Kwon, Oh-Cheon; Kim, Min-Jung; Shin, Gyu-Sang (Electronics and Telecommunications Research Institute, Korea). "Research on an MDA-based COP approach"
8. Marangozova, Vania (INRIA Rhne-Alpes, France). "Component quality configuration: the case of replication"
9. Matevska-Meyer, Jasminka; Hasselbring, Wilhelm; Reussner, Ralf H. (U Oldenburg, Germany). "Exploiting protocol information for speeding up runtime reconfiguration of component-based systems"
10. Nano, Olivier; Blay-Fornarino, Mireille (Universit de Nice-Sophia Antipolis, France). "Using MDA to integrate services in component platforms"
11. Overhage, Sven (Augsburg U, Germany). "Towards a standardized specification framework for component discovery and configuration"

12. Prochazka, Marek (INRIA Rhne-Alpes, France). “A flexible framework for adding transactions to components”
13. Puntigam, Franz (TU Wien, Austria). “State information in statically checked interfaces”
14. Reussner, Ralf H. (U Oldenburg, Germany); Poernomo, Iman H.; Schmidt, Heinz W. (both Monash U, Australia). “Contracts and quality attributes of software components”
15. Thomas, Bill; Vecellio, Gary (The MITRE Corporation, USA). “Infrastructure-based mediation for enforcement of policies in composed and federated applications”
16. Trofin, Mircea; Murphy, John (Dublin City U, Ireland). “A self-optimizing container design for Enterprise JavaBeans applications”