Abstract. This report covers the seventh Workshop on Component-Oriented Programming (WCOP). WCOP has been affiliated with ECOOP since its inception in 1997. 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 2002, held in conjunction with ECOOP 2002 in Malaga, Spain, was the seventh 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; and Budapest, Hungary.

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 had been announced as follows:

WCOP 2002 seeks position papers on the important field of component-oriented programming (COP). WCOP 2002 is the seventh event in a series of highly successful workshops, which took place in conjunction with every ECOOP since 1996.
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 component technology standards, such as CORBA/CCM, COM/COM+, JavaBeans/EJB, and most recently .NET, but also the increasing appreciation of software architecture for component-based systems, and the consequent effects on organizational processes and structures as well as the software development business as a whole.

After WCOP'96 focused on the fundamental terminology of COP, the subsequent workshops expanded into the many related facets of component software. WCOP 2002 has an explicit focus on dynamic reconfiguration of component systems, that is, the overlap between COP and dynamic architectures. Also, submissions reporting on experience with component-oriented software systems in practice are strongly encouraged, where the emphasis is on interesting lessons learned, whether the actual project was a success or a failure.

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, 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. Although we had one session on foundations of components, the discussion was primarily concerned with the runtime characteristics of component-based systems. Thirteen papers were accepted for presentation at the workshop and publication in the workshop proceedings. About 40 participants from around the world participated in the workshop. The workshop was organized into four morning sessions with presentations, one afternoon breakout session with five 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 thirteen presenters, as grouped into four sessions, i.e. Foundations, Components and Generators, Monitoring, EJB and COTS, and, finally, Dynamic Reconfiguration.

2.1 Foundations

The first session consisted of three papers. The first paper by Andreas Gal addressed the performance of component-based systems. Due to the independent deployment, traditional compiler optimization techniques cannot cross component borders. The proposed solution is dynamic compilation that, at run-time, performs inter-component optimization. The solution has been implemented in Lava (combining Lagoona and Java), exploits profile-guided optimization and dynamic compilation.

The second paper was presented by Yahya Mirza and discussed a compositional collections component framework. The basic claim of the paper is primitive composition patterns need to be identified and supported kernel or virtual machine.

The third paper in the foundations session was presented by Peng Liang and addressed rate-monotonic analysis for scheduling field device components. Field devices have strict timing requirements and a prerequisite for using software components in this domain is the ability to accurately predict timing properties of the composed system.

2.2 Components and Generators

The second session also contained three papers. The first paper, presented by Pedro Jose Clemente Martin, discusses the use of UML for combining component-based and aspect-oriented software development. The approach separates component dependencies into intrinsic and non-intrinsic. In the case of intrinsic dependencies the component only depends on the framework or context. A non-intrinsic dependency is really an aspectual (cross-cutting) dependency. The approaches uses tag values in UML to express aspectual dependencies.

The second paper, concerning (web) services, was presented by Oliver Nano. His paper discusses the integration and composition of web services in component platforms such as EJB and CCM. The proposed approach is to define an algebraic meta model and to define operations on meta-objects that achieve service integration. Services are specified operationally over abstract operations.

Massimo Tivoli discusses the role of software architecture in the assembly of components. His paper starts from the premise that current COTS component composition technologies cannot solve the predictable assembly problem. For many systems important system properties include safety and liveness and the software architecture, providing a connection skeleton with connections captured by connectors, allows for predictability. The presented approach, given set of components C and desired properties P, automatically composes C such that P is met.
2.3 Monitoring, EJB and COTS

The third session was concerned with monitoring and dynamic adaptability of component-based systems and consisted of three papers. The first paper, presented by Adrian Mos, presented the COMPAS framework. COMPAS provides tool support for monitoring component-based systems, modeling and prediction of performance. The approach starts from the observation that usually poor performance is caused by poor design and not by poor code. The COMPAS framework provides non-intrusive monitoring, stochastic model generation, UML-MDA model representation, workload based performance prediction and tight integration between monitoring and modeling.

The second paper addressed container services for high-confidence software and was presented by William Thomas. High confidence software is defined as being able to withstand attacks and hazards without causing accidents and unacceptable losses. High confidence software is difficult to develop and validate and, in addition, the validation is often brittle. Although both design and run-time techniques exist, there is a clear shift towards run-time techniques. The author presented an approach in which containers, potentially multiple, are used to capture and verify properties. Mediators containers can be used to enforce properties during invocations, whereas monitor containers can enforce global properties.

Finally, the third paper by Zahi Jarir discussed the dynamic adaptability of services in the Enterprise Java Beans component model. The main motivation is that especially in the context of web services dynamic adaptation of behaviour is required, although the current EJB model does not support this. The proposed approach is to allow EJB applications to be aware of and adapt to variations in execution context through the use of an infrastructure for adaptable middleware and rule-based adaptation policies. This approach has been implemented in JonAS, an adaptable EJB infrastructure implementation.

2.4 Dynamic Reconfiguration

The last session consisted of four papers. The first paper was presented by Chris Salzmann and discussed the notion of architectural invariants. The approach is to separate the logical and technical architecture. The logical architecture focuses on component types and functional dependencies whereas the technical architecture is concerned with the actual components that implement the logical architecture. This is supported in the Service Architecture Definition Language (SADL), an XML-based definition language including support for services, components, bindings and sandboxes.

The second paper discussed the DiPS/CuPS component framework that allows for ‘hot-swappable’ system software, in particular for flexible protocol stacks. The DiPS part of the approach contains a model of plug-compatible components, responsible for exactly one protocol function, organized using the pipe-and-filter architectural style. Plug compatibility is achieved by two design decisions. First, component functionality is kept fully separate and second, the
framework is used to establish connections. The CuPS part supports, as a coordination layer, non-anticipated adaptations of a running protocol stack.

Eric Bruneton presented an approach to recursive and dynamic software composition with sharing. The aims of the author are to have composite components where individual components can still be shared, while allowing for dynamic reconfiguration and the satisfaction of non-functional properties. The Fractal component model is presented as a solution. It provides a flexible API to introspect and reconfigure Fractal components as well as one or more controller classes for each interface of the API. Fractal components are created by composing controller classes.

The final paper presented X-Adapt, an architecture for dynamic systems, and was presented by Finbar McGurren and Damien Conroy. X-Adapt is a reflection-based architecture; it can place proxies between server objects to intercept messages. These proxies interact with a configuration manager, which uses configuration rules and maintains a configuration map. The configuration manager draws on a set of monitor to detect changes to initiate and control service-level adaptation.

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. mechanisms and decision making for reconfiguration and run-time optimization. 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 briefly discussed.

3.1 Mechanisms for reconfiguration

The material in this subsection is largely based on a contribution by Stefan van Baalen, who volunteered to gather the results from the break-out group that he participated in.

The breakout group first considered the reasons for reconfiguration. The primary argument is that Systems are constantly evolving in response to new requirements. To address this, there is a need for mechanisms that permit reconfiguration of systems against a minimal cost. These mechanisms should allow us to change the component configuration (e.g. the dependencies that a component requires). One can identify two types of system reconfiguration, i.e. dynamic (at run-time) and static (at compile-time).

We have used mechanisms for static reconfiguration for a long time, but why are we currently interested in dynamic reconfiguration at run-time? One can identify several reasons, e.g. to update software while keeping current state, to evolve systems that cannot be stopped and rebooted (24x7x365) and, finally, many systems are ‘in the field’ and cannot easily be physically accessed for upgrading.
One can identify several types of reconfiguration, e.g. instance replacement, service level adaptation, change of component configuration, interface adaptation and change of message protocol. Reconfiguration can be initiated by several sources, e.g. sender, receiver, third party in system (e.g. monitor component) or external trigger (e.g. human interaction). The next topic is what elements are aware of a reconfiguration, i.e. only the sender, only the receiver, both the sender and receiver or, finally, only the connector (sender and receiver are not aware of change).

Further, there are several aspects that influence the complexity of reconfiguration. These include the component communication mechanism (method calls, events/message passing or black-board communication/shared memory), stateless vs. ‘statefull’ components and ‘big bang’ reconfiguration or parallel existence of old and new versions. Especially the situation of ‘statefull’ and parallel existing components raises a difficult need of shared component state needed between the old and new component.

As a conclusion, the break-out group presented a number of possible mechanisms for reconfiguration. First, aspect oriented programming has the necessary tools to allow reconfiguration of the components that form the final system. The modeling of these systems and the representation of each component and their dependences permit to obtain a framework to represent each aspect of the system. Then component functionality can to be adapted to new requirements using AOP techniques. Second, reification to meta-level and reflection in sender and/or receiver. In this case, calls to component are reified to the meta-component. The meta-component can perform all necessary actions to adapt the message. Code for reconfiguration is represented as state of this meta-component (codified as data). In this way, non-procedural artifacts such as business rules, restrictions, etc. as well as procedural artifacts can be reconfigured. Subsequently, the adapted call can be invoked on target component using reflection. Third, proxy objects can be placed between the sender and the receiver. In this case, all calls to component are actually done on its proxy component, the proxy can perform the actions necessary to adapt the message. Finally, the proxy can invoke the adapted call on target component. Finally, one may employ a component system with message delegation based on component identifiers. In this case, a component sends a message to other components by passing an identifier to the component system. The component system uses directory service to map identifier to the receiver component. Subsequently, the component system delivers message to target component. As the component system performs the mapping dynamically, it can ‘rebind’ a component name to a new target component.

3.2 Model Extraction and Model Driven Architecture

The second break-out group focused on the relationship between models and architectures. Starting from the observation that there is still a lack of consensus on what an architecture is, the group listed a few understood characteristics:

– Business rules are towards the top of the hierarchy;
- Components and connectors permeate through the hierarchy;
- Down to but excluding platform/technology level.

The group described its own interests in this space:

- **Engineering**: Develop graphical representation of architecture, covering both static and dynamic views. Using such representations as input to formal language systems, with a goal to generate systems. The current implementations are function-based systems; component-based systems (with aspects) are work in progress.
- **Reverse Engineering**: Automatically capture specifications of blackbox COTS at the level of component interfaces and component connectors. To date a graphical approach is used but input of specifications is required. A possible probabilistic approach has been identified.

In the area of reverse engineering, the group focused on compile time versus run time opportunities. At runtime, a performance model can be determined by extracting properties at intercepting component proxies. These models can be represented at PIM and PSM levels. At compile time, an activity graph (encoded in XMI) can be generated. Sequence diagrams (also encoded in XMI) can be generated by probes at run-time.

The group sketched the relationships between UML, XMI, source code, and executable:

1. UML and XMI should be equivalent;
2. UML is the basis for source and XMI is derived from source;
3. Executables are derived from source and XMI is reified from executables.

Finally, the group identified a number of issues. In particular, given component invocations and timestamps (and other technology-related information), how can transaction flow in the system (and thus the ‘start’ of connectors) be determined? How can the result be represented? Is MDA suitable here? Can UML be extended to cope with components and aspects?

### 3.3 Decision Making for Dynamic Reconfiguration

The third break-out group also started by defining dynamic reconfiguration. This group defined it as a system, existing of components & connectors, where the set of components may change during runtime change. Also, a number of stimuli initiating reconfiguration were identified, e.g. a failing component, system improvement (functionality or quality attributes), specialization and response to changing resources.

The decision making process is basically concerned selecting the optimal configuration among a set of possible configurations. To address this, the group first defined four cases, organized along two dimensions, i.e. adaptable versus self-adaptive and open versus closed. Open reconfiguration allows for new component types to be added to the system, whereas closed does not. Further, self-adaptive
systems initiate and reconfigure without external involvement whereas adaptable systems are reconfigured by a party external to the system.

The main conclusion of the break-out group is that the optimal configuration for a system is depending on the ability of the system to represent relevant information about the itself, e.g. the quality attributes, to evaluate this information and to predict the effect of reconfiguration on the relevant properties of the system. To achieve this, abstractions are needed.

3.4 Run-Time Optimization

The fourth break-out group addressed the issue of run-time optimization in component-based systems. The relevance of this topic is clear when one realizes that before deployment optimization can only take place within a component. Inter-component optimization can only take place once the system exists. Especially in the face of dynamic reconfiguration, run-time optimization is unavoidable.

Optimization is typically about minimizing the use of resources, such as CPU cycles, memory, power consumption and inter-machine communication bandwidth. One concern with run-time optimization is that in order to be effective, one has to measure the property to optimize, preferable both before and after the optimization. Nevertheless, several techniques are available to perform the actual optimization. At the code level, dynamic recompilation and code optimization and optimized method invocation can be used. At the component level, one may replace an entire component version, migrate a component between address spaces or machines and optimize the component configuration.

However, run-time optimization provides optimized resource usage against a cost because of the need to monitor, making decisions and the optimization operation itself. This cost may express itself as increased footprint and development effort and the increased difficulty to predict the behaviour of the system.

Although the break-out group realized the benefit and importance of run-time optimization, the group also identified a number of problems that need to be addressed. These include security and resource concerns associated with inline optimization and optimization of embedded/RT systems, especially with small footprints.

4. Final Words

As organizers, we look back on yet another 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 seventh international workshop on component-oriented programming. In particular, we would like to thank the presenters of the break-out group results.
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/WCOP2002/). This site also has the details for the Microsoft Research technical report that gathers the papers and this report.

3. E. Bruneton, T. Coupaye, and J. B. Stefani. “Recursive and Dynamic Software Composition with Sharing.”
10. Ch. Salzmann “Invariants of component reconfiguration.”
11. P. Liang, G. Arévalo, S. Ducasse, M. Lanza, N. Schaarli, R. Wuyts, and O. Nierstrasz. “Applying RMA or scheduling field device components.”