

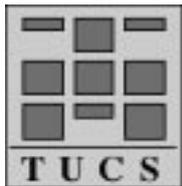
Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP '97)

Editors:

Wolfgang Weck

Jan Bosch

Clemens Szyperski



Turku Centre for Computer Science

TUCS General Publication No 5

September 1997

ISBN 952-12-0039-1

ISSN 1239-1905

Contents

Jan Bosch, Clemens Szyperski, Wolfgang Weck: Summary of the Second International Workshop on Component- Oriented Programming (WCOP'97)	1
Vito Baggiolini, Jürgen Harms: Toward Automatic, Run-time Fault Management for Component- Based Applications	5
Jan Bosch: Adapting Object-Oriented Components	13
Martin Büchi, Emil Sekerinski: Formal Methods for Component Software: The Refinement Cal- culus Perspective	23
Michael Goedicke, Torsten Meyer: Design and Evaluation of Distributed Component-Oriented Soft- ware Systems	33
Koen De Hondt, Carine Lucas, Patrick Steyaert: Reuse Contracts as Component Interface Descriptions	43
Philippe Lalanda: A Control Model for the Dynamic Selection and Configuration of Software Components	51
Leonid Mikhajlov, Emil Sekerinski: The Fragile Base Class Problem and Its Impact on Component Systems	59
Tobias Murer: The Challenge of the Global Software Process	69
Stefan Schreyjak: Coupling of Workflow and Component-Oriented Systems	77
Jørgen Steensgaard-Madsen: A generator for composition interpreters	87
Jose M. Troya, Antonio Vallecillo: On the Addition of Properties to Components	95
Wolfgang Weck: Inheritance Using Contracts & Object Composition	105

Preface

This volume constitutes the proceedings of the Second International Workshop of Component-Oriented Programming (WCOP'97), held June, 9 in Jyväskylä, Finland, as a ECOOP Workshop. Included in the book are a summary of the presentations and discussions at the workshop and the twelve accepted and presented papers.

We would like to thank all those who helped to make WCOP'97 a success. Most importantly, we thank all participants for coming and contributing to the lively and interesting discussions. Many thanks also to all the authors for submitting a paper. We are also grateful to the ECOOP workshop chair, Antero Taivalsaari, who did an excellent job in preparing the event. Thanks also to the University of Jyväskylä and the staff who supported us during the day. Finally, this volume could have not been produced without the financial support of the Turku Centre for Computer Science (TUCS), Turku, Finland, and without Mats Aspnäs, who helped with the text processing.

For the WCOP'97 Organizers
Wolfgang Weck, September 1997

Summary of the Second International Workshop on Component-Oriented Programming (WCOP'97)

Jan Bosch

University of Karlskrona/Ronneby, Dept. of Computer Science
Ronneby, Sweden
Jan.Bosch@ide.hk-r.se

Clemens Szyperski

Queensland University of Technology, School of Computing Science
Brisbane, Australia
c.szyperski@qut.edu.au

Wolfgang Weck

Turku Centre for Computer Science and Åbo Akademi University
Turku, Finland
Wolfgang.Weck@abo.fi

WCOP'97, held together with ECOOP'97 in Jyväskylä, was a follow-up workshop to the successful WCOP'96, which had taken place in conjunction with ECOOP'96. Where WCOP'96 had focused on the principal idea of software components and their goals, WCOP'97 was more directed towards composition and other topics, such as architectures, glueing, component substitutability, evolution of interfaces, and non-functional requirements.

WCOP'97 had been announced as follows:

COP has been described as the natural extension of object-oriented programming to the realm of independently extensible systems. The most prominent examples of such systems are constructed around compound document models such as OLE, OpenDoc, JavaBeans, or Netscape ONE and rest on object models such as SOM/CORBA, COM or Java's virtual machine. WCOP'97 intends to address their methodological and theoretical underpinnings.

COP aims at producing software components for a component market and for late composition. Composers are third parties, possibly the end user, 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. These needs raise open research questions like what kind of standards are needed and how they should be defined. Or what information specifications need to give, how this information should be provided, and how correct implementation and usage of specifications could be verified or enforced.

16 position papers were submitted to the workshop and formally reviewed. 12 papers were accepted for presentation at the workshop and publication with the proceedings. Unfortunately, the paper by M.Goedicke and T.Meyer could not be presented, because force majeure hindered the authors to attend the workshop. Still, 25 participants from 13 countries were counted at the workshop.

During the morning session, participants presented their work, which covered a wide range of topics. A major theme was how to select components for composition in a specific situation. Such a selection must rest on two pillars. Firstly, the selected components must be compatible with each other. Secondly, characteristics that are not part of the standardized component interface may decide which component to pick from otherwise equal ones. Examples are time or resource requirements, fault tolerance, degree of distribution, etc.

To address the compatibility of components, various approaches and philosophies were presented. An important property of component-oriented programming is that a single specification may be supported by multiple implementations. However, problems may arise if individual implementations depend on the implementation of other components. These dependencies may cause conflicts, which can often only be detected when the composed system is analysed as a whole.

One solution is that dependencies on other components as well as known conflicts with other components become part of a component's specification. Reuse Contracts [De Hondt et al.] have been proposed as a tool for this. They also allow the composer to decide quickly whether a given set of components may conflict.

[Mikhajlov & Sekerinski] suggest to define rules that, if being followed, exclude conflicts in principle. These rules affect the design of specifications, the implementation of components, and the implementation of a component's clients. For inheritance between classes of objects, such rules can be derived formally.

A third approach is to accept that components will have some dependencies that are not part of a specification and hence cannot be checked by the composer. The component creators, however, are aware of these dependencies. Thus, this knowledge, available during component creation time, has to be maintained and made accessible to system composers. [Murer] suggests that this requires tool support.

Finally, a component may not be applicable in a specific situation as it is. In these cases, it needs to be adapted, which can be done either by modifying the program's source code or by wrapping it. Both approaches have their disadvantages. Alternatives on a middle ground are needed. [Bosch] proposes the use of component adaptation types that can be superimposed on components.

One aspect of specifications is that they embody a contract between programmers of service providing components and service clients. Because it is impossible to test a provider component against all clients and vice-versa, it must be decided without testing both whether a specification is implemented correctly and whether a client uses it correctly. For this, formal methods are helpful, but need to be made applicable in practice. [Büchi & Sekerinski] address the problem of poor scalability by specification statements, which are used in refinement calculus.

The second mayor theme of the presented work were properties of components that are not part of the (functional) standard interface. One may want to add such properties to existing components when putting them together to a complete system. This allows the system's composer to pick those properties that are actually needed in the specific situation. [Troja & Vallecillo] discuss some technical precautions for this, such as a specific communication mechanism.

An example of such add-on properties are mechanisms for run-time fault man-

agement in distributed systems. [Baggiolini & Harms] propose to use wrappers for providing monitoring, diagnosis, or failure correction.

Components that are otherwise interchangeable will distinguish themselves by some important (unchangeable) properties, such as resource requirements. It is an important task to select the right components, meeting possible constraints imposed by the deploying system or the problem to be solved. [Lalanda] suggests that this selection may be best made at run-time, and proposes a special architecture.

Some of the work addressed other topics than these two main themes. Workflow systems seem to lend themselves to component-oriented software, because of their configurability and building-block-like structure. [Schreyjak] proposes a special component framework to support component-based workflow systems.

One way of composing systems is by expressing relations and cooperation between components in a special language. [Steensgaard-Madsen] proposes an interpreted language, in which the commands are components. Such language interpreters are specialized for an application domain and need to be generated automatically.

[Weck] discusses the problems of code inheritance across component boundaries, such as the danger for unwanted dependencies. Instead, inheriting classes need to refer to specifications of base classes. With this, inheritance can be replaced by object composition without sacrificing the possibility of static analysis, yet being more flexible.

Because of the many participants, during the afternoon session the workshop was split up into discussion groups. The participants expressed interest in four areas: Components, Architectures, Non-Functional Requirements, and Glue. The following are short summaries, based on presentations and notes provided by different participants of the discussion groups.

Components As a start, it was recognized that what makes something a component is neither a specific application nor a specific implementation technology. In this sense, "anything" may be cast into a component. To provide access to something about which so little is known, an interface needs to be provided. Interfaces are mainly seen as a collection of "Service Access Points", each of them including a semantics specification. The main purpose of components is reuse of both implementations and interfaces. For effective implementation reuse, the aforementioned independence from implementation technology is particularly important. Two kinds of life cycles are to be distinguished: that of the interface and that of the component itself. The latter is shorter than the former, because the interface exists as long as any implementation is around. For interfaces, formalization of semantics is necessary. Even more important, the interoperation between components must be described. On the technical level, one needs a binary interoperation standard and a mechanism to map semantics specifications to implementations using this binary standard.

Architecture Architecture describes compositions of components, and therefore relationships between them. This requires consideration of the component's interfaces. Architecture is to be stated in terms of interfaces rather than component implementations. In contrast, if architecture would be seen just as design patterns for composition, a concrete architecture may not be realizable because the components at hand may not fit together (architectural mismatch). On the other hand, in a given architecture, components are replaceable by others implementing the same interface. Thus, architecture represents the longer lasting and slower changing design as opposed to component implementations. More precisely, an architecture consists of a collection of interfaces

that represent slots to be filled (or roles to be played) by components. Some supporting white-box implementation, for instance, a kernel, may be bundled with a given architecture.

Non-Functional Requirements Examples of systems currently under construction were collected together with their specific non-functional requirements. For instance, an avionics system that plans trajectories of a plane and must react to route problems (such as a storm or being low on fuel) must be fast (2-3 second response time) and must adapt itself to many different situations that might arise. Secondly, a system for numerical computing on parallel processors must run fast on a given parallel machine. It also must be quickly portable to run on a new machine. Thirdly, software for controlling a kidney dialysis machine must be responsive (quickly read various sensors and updates actuators), flexible (to adapt easy and reliably to changes in hardware, such as a new pump model, or medical practice, such as a new protocol for dialysis), and demonstratable (to be shown to a regulatory agency to convince them of its safety and benefit).

There are different ways of meeting non-functional requirements, depending on the type of requirement. Some are automatically satisfied if each component of the system is properly designed. Others arise out of the interaction of components, and can only be addressed at that level, not at the level of individual components. Four ways of providing non-functional properties could be found. One can parameterize components so that specific properties can be requested of them; or one can reorganize the components to deal with the property; or one can design an overall architecture that is responsible for the property and that can provide it if the components adhere to the architecture; or, finally, a meta-level mechanism can provide access to the component interaction to deal with the property. The latter is similar to aspect-oriented programming.

Glue By glue, the participants understood middleware that is used to connect existing components. Examples are Tcl/Tk, scripting mechanisms, even make files. Some support for typing would be nice to have but hard to achieve due to the vast variety of types components may introduce. In general, the glue is more flexible than the components glued together, and thus should use a dynamic language. In connection with the discussion on architecture, it turns out that components are sandwiched between architecture and glue. To be accessible from within a given scripting environment, the components must meet some architectural requirements, like accepting messages sent by the script interpreter. Thus, the script (glue) builds on components that in turn are built for the scripting architecture.

Toward Automatic, Run-time Fault Management for Component-Based Applications

Vito Baggiolini and Jürgen Harms

Centre Universitaire d'Informatique Université de Genève

Rue Général-Dufour 24 1211 Genève 4

vito@cui.unige.ch

<http://cuiwww.unige.ch/~vito>

While components are well tested and intrinsically more reliable than custom-made software, applications built of components generally lack global verification (especially if they are independently extensible) and are subject to distributed failure scenarios. We discuss a simple framework for building fault-resilient applications based on a data flow architecture. We illustrate the characteristics that make this architecture particularly suitable for automatic fault management and explain the mechanisms we use for detecting, diagnosing and correcting faults at run-time.¹

1 Introduction

The background of this work lies in fault management of open distributed applications, a field we have been working in for the past few years. A year ago, we have terminated a project in which we built a management infrastructure for the operational E-mail service of our university [1, 2]. Now we are exploring the field from a more conceptual viewpoint, aiming at elaborating a sound basis for automated fault management [3].

The goal of fault management is to keep a system up and running and to cope with failures in the environment or in parts of the system itself. A *manager* monitors the behavior of the system, and, when it detects failures, it acts on the system to diagnose and correct them. Management can be done manually (by a human operator), or it can be carried out partly or fully automatically by a software application. Let us introduce a few related vocabulary items: a *failure* is the inability of a system to fulfill a task it was intended for, and a *fault* is the actual cause of the failure. Fault management aims at handling faults before they cause failures. Fault management has three phases: *monitoring* to detect fault symptoms, *diagnosis* to identify the type and location of the fault and *correction* to eliminate the fault.

Current approaches to fault management of distributed applications typically use network management paradigms and technology [4, 5]. Based on our experience gained in the E-mail management project, we believe that this approach does not have the potential to provide a basis for automatic management. There are two main problems we have identified:

¹This work was funded by a research grant of the Swiss National Science Foundation, Project 2129-04567.95

- Management is added as an afterthought to existing applications. This means that applications are used for a purpose they were not intended for. The architecture may for instance be completely unsuitable for management or there may be no appropriate means of acting on the application at run-time. If there is any functionality for management, it generally is aimed at humans (e.g. system managers), and unsuitable for automatic use by a manager application. This leads to complex and inefficient management solutions.
- The software is large and monolithic. A distributed application is composed of processes running on different machines and these processes correspond to large, monolithic programs (e.g. E-mail relays). A finer granularity is needed for management. Information about internal structure and dynamic behavior of the managee processes must be accessible and it must be possible to *selectively* act on aspects of their behavior and on their structure.

We believe that manageability must be considered in the software engineering process right from the start. In other words, we want to build *manageable applications*. We advocate a framework containing (1) "application components", which are the building blocks for the primary application functionality and (2) "management components" that are capable of handling typical failure scenarios. A developer constructs an application out of the application components and makes it failure-resilient by "mixing in" management components that take care of potential failures.

2 Problem Description

Components are mature and reliable pieces of software. They are extensively tested and have many less bugs than custom-made code. However, this is not the case for combinations of components [6, 7]. It is tedious to test all possible combinations between components of a framework and impossible to do so in the case of components produced by different manufacturers. Interoperability problems are inevitable. Global analysis of component-based applications is complex. It is unfeasible for applications that can be independently extended by the end-user, and of course for open distributed applications.

Fault management is confronted with various problems, of which the following two are of special interest for this paper:

- Propagation of faults. A fault in a component may cause wrong behavior in other components and provoke fault symptoms that often are distant from the cause and at first do not seem to have any relation to it. The symptoms must be traced back to the actual fault, which requires, amongst other things, accurate and detailed knowledge on dependencies between parts of the system. Such dependency information, however, is generally difficult to establish.
- Distributed failures. Distributed failures have their origin in faulty interactions of several components; examples of such failures are interoperability problems, deadlocks or data forwarding loops (the latter happen in message-passing systems [8]). Often, there is no single component that can be diagnosed as being faulty, and it may be very difficult to decide how and where to intervene to correct the problem. This becomes even more difficult if all components are actually behaving correctly on their own but their dynamic or structural combination occasionally fails to work properly.

3 A Manageable Data Flow Architecture

In this section, we present an application architecture that has many characteristics that enable fault management and that addresses many aspects of the problems listed above. We first describe our design and then explain its suitability for fault management.

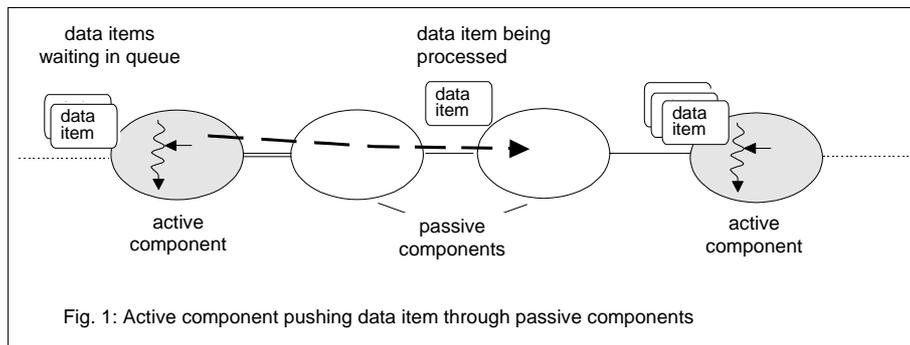
3.1 Description

The architecture we use is known as "data flow" or "pipe-and-filter" (see e.g. [9]). Well known examples are Unix shells which allow commands to be connected by pipes, image processing frameworks where images flow through several filters, routers that forward data through a network, E-mail relays forwarding messages, and so on.

This architecture is based on interconnected filter components that accept data items on one side, carry out transformations on them and push them out on the other side. This push-paradigm results in a flow of data items through the application.

The specific framework we are developing for our research contains active and passive components. Active components are the "motors" of the application, while passive components carry out the data processing. Active components contain a thread and an incoming queue in which upstream components can deposit data items.

An active component is typically followed by a number of passive components. It fetches one data item after the other from its queue and pushes them through the passive components up to the next active component (Fig. 1).



Passive components are stateless: except for transient information on the data item being processed, they do not contain dynamic data processing state. Once a data item has been processed, the component discards all related state. The only permanent state is related to configuration (e.g. a component's downstream peers, or its configuration parameters).

Components are connected together using a registration scheme. Connections are normally set up when the application is started, but they can be freely modified at runtime. Components can in principle be interconnected to arbitrary graphs. But typically, they are recursively grouped to composite components which have essentially the same behavior as elementary components and are again connected to graphs. This results in a hierarchical design with intuitive abstraction layers.

3.2 Features enabling fault management

We have identified a number of generic "manageability" requirements, i.e., characteristics that make an application suitable for fault management. Among these are:

modularity, homogeneity, accessibility and statelessness. This section explains them in more detail and illustrates how our framework complies with them.

Modularity. Our framework consists of weakly coupled modules which interact along well-established dependencies by forwarding encapsulated data items. This design has the following advantages for management: (1) it favors fault containment: a faulty component cannot corrupt others, and fault symptoms are less likely to propagate to other parts of the system. (2) It makes diagnosis easier: if a fault symptom propagates, this can only happen along well-known dependency paths. Thus, when a fault symptom is detected, it can be traced along the dependencies back to the actual fault. (3) It makes corrective interventions easier: components can be manipulated independently from one another and they can even be substituted at run-time.

Homogeneity. Our framework is based on one single processing paradigm (receive, process and forward data) and one single type of component interaction (one-to-one forward connections through which uniform data items are forwarded). This pattern is homogeneously applied throughout the whole application: all components (elementary or composite) follow the same processing paradigm, and the interconnections work the same through all abstraction levels. As a consequence, management is easier to accomplish. As only one paradigm must be managed, there are less possible failure scenarios to be dealt with and consequently a smaller set of management algorithms is sufficient.

Accessibility. The internal characteristics of applications built with our framework are accessible to management. The components can be manipulated selectively. The application structure (the connections between components) can be explored and re-configured, and components can be substituted. The interactions between components can be analyzed for correctness. All these interventions can be carried out at run-time.

Statelessness. In our design, all data processing state is encapsulated in the data items. The passive components contain data processing state only while they process a data item, and discard it thereafter. The active components do not contain data processing state at all, just references to the data items waiting in the input queue. This has several positive consequences: (1) Components can be manipulated independently, because there is no distributed state to be maintained. (2) Checkpointing is limited to data items, the components do not have state that needs to be checkpointed. (3) The application is more robust: since they are stateless, components cannot be crashed by a corrupt data item; at worst, that data item fails to be processed correctly.

3.3 Generic fault management strategies and algorithms

One of our goals is to find generic management strategies and algorithms that can be applied to different kinds of components and data items regardless of their specific function or contents. The idea is to take typical failure scenarios that can happen to applications based on the framework and to develop the algorithms for handling these faults. These algorithms are implemented as re-usable management components. The application components must be "management-enabled", i.e., they must contain functionality to detect failure symptoms, and they must provide a management interface through which they can be controlled by the management components. In the following, we discuss the monitoring mechanisms used in our framework and a number of algorithms and strategies for diagnosis and fault correction.

Monitoring. In order to detect suspicious situations as fast and close to the cause as possible, monitoring should ideally be carried out almost continuously and everywhere in the application. However, only a very limited overhead can be tolerated, and

the mechanisms and their use must be carefully chosen. We use the following lightweight monitoring mechanisms: (1) *Data consistency checks* integrated into the data objects are executed every time the data enters a component. They are based on simple assertions on invariant characteristics of the data and constitute a simplified form of pre- and post-conditions as used in software contracts²[10]. (2) A *hop counter* contained in the data items is incremented in every component the data passes through. It makes it possible to detect loop failures (i.e., data items being continuously forwarded in a circular manner between a set of components). (3) The *queue lengths* of the active components are monitored to detect bottleneck conditions. (4) *Time-out mechanisms* protect the forwarding of data items from one component to the next and help to detect deadlock conditions.

In addition to the above, components contain *self-tests routines* that are executed from time to time (possibly when the application is idle). They assert amongst others that the component have enough resources and that their configuration and internal data structures are consistent.

Diagnosis. Diagnosis starts when a failure condition has been discovered by monitoring. Diagnosis has the goal to establish a hypothesis about the fault type and the component(s) responsible for the failure. This hypothesis serves as a basis for carrying out corrective actions. Diagnostic activities include "isolation testing" of data items and components, inspection of the data stream between components, tracking of dependency paths and analysis of the application structure.

- *Isolation testing* is not based on the integrated test functions of data items or components, but essentially black-box testing. To test a component, some specifically crafted data items are sent through it, as a kind of "test suite" of data items. For this purpose, the component is temporarily taken out of the data stream (hence the name "isolation testing"), so that the tests can be carried out in the "live" execution environment without affecting the rest of the application. A similar procedure is used for testing data items: they are sent through a diagnosis component that is specially tailored for their type and capable of finding inconsistencies in their contents.
- *Data stream inspection* is used for instance for diagnosing cases of incompatibility between two components. A diagnosis component is inserted between the two components to determine which one of them is behaving faultily. In a similar way, a component's behavior can be assessed by observing the interactions with all its neighbors.
- *Dependency paths* are explored to trace back a fault symptom to the actual fault. For example, if a corrupt data item has been detected, it is likely that it has been corrupted by a previous processing step. Thus, the dependency path must be followed upstream. Conversely, if there is a bottleneck scenario, one must track the downstream dependencies to find the component that slows down processing.
- *Structural analysis* is used to diagnose deadlocks and loops, which are both caused by circular connections between components. Deadlocks happen if too many data items are present in a cycle of components; data items keep looping in

²Note that in a chain of components, the post-condition of a component coincides with the next component's pre-condition.

a cycle due to erroneous forwarding (routing) information. In this case, all components that are part of the loop must be identified, as well as those components where data items flow into and out of the loop.

Fault correction. Once the faulty component has been identified, corrective actions must be taken to eliminate the fault. Note that failures *inside* a component, e.g. design flaws or programming errors, cannot be corrected at run-time. Failure correction is thus limited to the following options: (1) re-configuration, re-initialization or substitution of a component and (2) reconfiguration of the application structure.

Examples for *component reconfiguration* are: augmenting the number of slots in the in-queue of an active component; switching to a resource-conservative mode in case of a resource shortage; insertion of additional active components in a composite component to enhance throughput. If a component is found to be buggy, it can be substituted by a compatible replacement component.

Management actions aimed at the *reconfiguration of the application structure* include for instance: de-coupling of incompatible components by insertion of a management component in between; insertion of additional components in parallel to an overloaded component to alleviate bottlenecks; insertion of buffers into a loop to limit the potential of deadlocks happening.

In this way, most typical day-to-day problems can be corrected automatically. But of course, it is impossible to handle all possible failures automatically, and the intervention of a human manager (e.g. a system administrator) may be necessary from time to time. Even in this case, automatic fault management is fundamental to keeping a service up and running. Firstly, problems are detected at an early stage, before they cause a deterioration of the service, so that there is enough time left to alert a human, if necessary. And secondly, once a human administrator is available, s/he does not have to troubleshoot the problem from scratch, but s/he can build on the diagnosis that has already been made by the fault management system.

Last but not least, run-time fault management contributes to the long-term amelioration of software products: a complete "history of events" related to a failure provides invaluable debugging information to the developers of both the application and the underlying component framework.

4 Prototype Implementation

We have implemented a prototype of an E-mail relay based on our framework. It is composed of around thirty elementary components grouped into nine composite components. Each of these implements a piece of high-level functionality (like the SMTP protocol engine, aliasing of addresses, formatting of the message body, etc.). In its current implementation, it contains a central management component with a user interface that gives a human manager access to the composite components and to all the elementary components they contain.

In the current prototype only monitoring is done automatically. The detected fault symptoms are dispatched to the management component, and the human manager can diagnose them at run-time using the components' management interface. This approach has proven to be very useful for the design and testing of management algorithms.

We use Java for our developments because it has many characteristics suitable for management. The language is object oriented, robust and includes support for multi-threading and exceptions, and the Java development kit (JDK 1.1) provides reflection, distribution, and persistence. Last but not least, our architecture (that we started to develop before JavaBeans were available) can easily be adapted to comply with the JavaBeans specifications, which would allow us to profit from the associated visual development environment.

5 Conclusions

There is an interesting symbiosis between component-oriented programming and automatic fault management: Component-based applications can benefit from run-time fault management to handle the unforeseen failure situations resulting from the lack of global testability. Fault management, in turn, profits from many characteristics of components as described in the paper.

The idea of management-enhanced component frameworks seems particularly promising to us. A framework containing management-aware application components as building blocks and management components that are able to handle framework specific failures makes it easier to build fault resilient applications. The advent of a standard component market will make failure handling by substitution of faulty components feasible.

The data flow architecture we have chosen is very suitable for management. While various kinds of applications can be built with this architecture, we do of course not expect that everybody - just for the sake of manageability - implement their applications with it. We intend to further investigate what are the core characteristics that make an application manageable, and how our approach can be applied also other application architectures. This, however, is a long-term goal of our research. For the time being, we are working on the implementation and automation of our algorithms and on their application to distributed store-and-forward applications.

Bibliography

- [1] V. Baggiolini, E. Solana, M. Ramluckun, S. Spahni, C.F. Tschudin and J. Harms, "Managing the E-Mail Service with the Distributed Management Tree", ICDP'96, Dresden, February 1996.
- [2] E. Solana, M. Ramluckun, V. Baggiolini, C. Tschudin, S. Spahni, J. Harms, "Vers une gestion automatique des applications distribuées: L'exemple du courrier électronique", 1er Colloque Francophone sur la Gestion de Réseau et de Service (GRES'95), Paris, October 1995.
- [3] J. Harms, C.F. Tschudin, "Generic fault management of heterogeneous distributed applications", Request for Funding, NFRS Project 2129-04567.95, Geneva, 1995. <http://cuiwww.unige.ch/Telecom-group/members/vito/GFM.html>.
- [4] G. Genilloud, M. Polizzi, "Managing ANSA Objects with OSI Network Management Tools", in Proceedings IEEE Second International Workshop on Services in Distributed and Networked Environments, Whistler, British Columbia, 1995.

- [5] J.W. Wong et al. "Distributed Applications Management Using the OSI Management Framework", Technical Report 448, University of Western Ontario, Canada, January 1995.
- [6] C. Szyperski, "Independently Extensible Systems - Software Engineering Potential and Challenges", Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 1996.
- [7] W. Weck, "Independently Extensible Component Frameworks", in M. Muehlhaeuser (ed.): Special Issues in Object- Oriented Programming, dpunkt Verlag, Heidelberg, 1997.
- [8] E. Solana, V. Baggiolini, M. Ramluckun and J. Harms, "Automatic and Reliable Elimination of E-mail Loops Based on Statistical Analysis", in Proceedings of the 10th Usenix Systems Administration Conference (LISA 96), Chicago, September 1996.
- [9] D. Garlan, M. Shaw, "An Introduction to Software Architecture", Carnegie Mellon University Technical Report CMU-CS-94-166, January 1994.
- [10] B. Meyer, "Applying Design by Contract", IEEE Computer, Vol. 25, No. 10, pp. 40-51, October 1992.

Adapting Object-Oriented Components

Jan Bosch

University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
Ronneby, Sweden
Jan.Bosch@ide.hk-r.se
<http://www.ide.hk-r.se/~bosch>

Several authors have identified that the only feasible way to increase productivity in software construction is to reuse existing software. To achieve this, component-based software development is one of the more promising approaches. However, traditional research in component-oriented programming often assumes that components are reused "as-is". Practitioners have found that "as-is" reuse seldomly occurs and that reusable components generally need to be adapted to match the system requirements. Component adaptation techniques should be transparent, black-box, composable, configurable, reusable and efficient to use. Existing component object models, i.e. white-box techniques, such as copy-paste and inheritance, and black-box approaches, such as aggregation and wrapping, these requirements. To address this, this paper proposes *superimposition*, a novel black-box adaptation technique that allows one to impose predefined, but configurable types of adaptation functionality on a reusable component. In addition, three categories of typical adaptation types are discussed, related to the component interface, component composition and monitoring.

1 Introduction

Component-oriented programming is receiving increasing amounts of interest in the software engineering community. The goal is to create a collection of reusable components that can be used for component-based application development. Application development then becomes the selection, adaptation and composition of components rather than implementing the application from scratch.

The abstract, naive view of component reuse is that the component can just be plugged into an application and reused as is. However, many researchers have identified that "as-is" reuse is very unlikely to occur and that in the majority of the cases, a reused component has to be adapted in some way to match the application's requirements. Adapting a component can be achieved using *white-box*, e.g. *inheritance* or *copy-paste*, and *black-box*, e.g. *aggregation* or *wrapping*, adaptation techniques.

In this paper, we argue that the aforementioned techniques are insufficient to deal with all required types of adaptation without experiencing, potentially considerable, problems. To address these problems, we introduce the notion of *superimposition*, a technique to impose predefined, but configurable, types of functionality on a component's functionality. The notion of superimposition has been implemented in the layered object model (**LayOM**), an extensible component object model. **LayOM** consists of

nested objects, methods, states, acquaintances and *layers*. The layers encapsulate the basic object and all messages sent to or from the object are intercepted by the layers. Through the use of these layers, **LayOM** provides several types of superimposing behaviour that can be used to adapt components. The advantage of layers over traditional wrappers is that layers are transparent and provide reuse and customisability of adaptation behaviour.

The remainder of this paper is organised as follows. In the next section, conventional component adaptation techniques are evaluated. In section 3, the notion of superimposing adaptation behaviour on components is discussed and a number of typical types of adaptation are described. Section 4 presents the layered object model and briefly discusses some examples of superimposing adaptation behaviour. In section 5, our results are compared to related work and the paper is concluded in section 6.

2 Component Adaptation Techniques

Component-based software engineering intends to construct applications by putting together reusable components. The naive view assumes that one selects a set of components that deliver parts of the application requirements and then put these components together by connecting inputs to outputs. However, research in software reuse has shown that components generally need to be adapted to match the application architecture or the other components.

2.1 Requirements for Component Adaptation Techniques

Before conventional component adaptation techniques are discussed, the requirements that a component adaptation technique has to fulfil in general are specified. These requirements provide a framework that can be used to evaluate conventional component adaptation techniques, but also to provide an insight in the required functionality of novel adaptation types.

- **Transparent:** The adaptation of the component should be as *transparent* as possible. Transparent, in this context, indicates that both the user of the adapted component and the component itself are unaware of the adaptation in between them. In addition, aspects of the component that do not need to be adapted should be accessible without explicit effort of the adaptation. Wrapping a component, for instance, requires the wrapper to forward all requests to the component, including those that need not be adapted.
- **Black-box:** As we identified in the previous section, the software engineer always has to develop some mental model of the functionality of a component before the component can be reused. This model should, however, be kept as small and simple as possible. One suitable approach is to make sure that the adaptation technique requires no knowledge of the internal structure of the component, but is limited to the interface of the component.
- **Composable:** The adaptation technique should be easily composable with the component for which it is applied, i.e. no redefinition of the component should be required. Secondly, the adapted component should be as composable with other components as it was without the adaptation. Finally, the adaptation should be composable with other adaptations.

- **Configurable:** As mentioned earlier, adaptation often consists of a generic and a specific part. For example, the adaptation type, changing operation names, has a generic part, i.e. replacing the selector in the message with another name and a specific part, i.e. which selectors should be replaced with what names. For the adaptation technique to be useful and reusable, the technique has to provide sufficient configurability of the specific part.
- **Reusable:** A problem of traditional adaptation techniques is that both the generic and the specific part are not reusable. A new technique should address this and provide reusability of the adaptation type and particular instances of the adaptation type, i.e. both the generic and the specific part.
- **Efficient:** All forms of component reuse, be it as-is or otherwise, require the reuser to construct a mental model of the functionality provided by the component. The larger and complex the mental model that the software engineer has to construct, the less efficient the actual reuse of the component will be.

2.2 White-box Adaptation Techniques

Copy-Paste When an existing component provides some similarity with a component needed by the software engineer, the most effective approach may be to just copy the code of that part of the component that is suitable to be reused in the component under development. After copying the code, the software engineer will often make changes to it to make it fit the context of the new component and additional functionality will be defined or copied from other sources.

Although the copy-paste technique provides some reuse, it obviously has many disadvantages, among others the fact that multiple copies of the reused code are existing and that the software engineer has to intimately understand the reused code. However, from our discussions with professional software engineers and students, we were surprised to see how often this technique is applied, especially when time pressure or other factors may force for a "quick-and-dirty" approach.

Inheritance A second technique for white-box adaptation and reuse is provided by inheritance. Inheritance as provided by, e.g. Smalltalk-80 and C++, makes the state and behaviour of the reused component available to the reusing component. Depending on the language model, all internal aspects or only part of the aspects become available to the reusing component. Inheritance provides the important advantage that the code remains in one location. However, one of the main disadvantages of inheritance is that the software engineer generally must have detailed understanding of the internal functionality of a superclass when overriding superclass methods and when defining new behaviour using behaviour defined in the superclass.

2.3 Black-box Adaptation Techniques

Different from white-box adaptation techniques, black-box adaptation techniques reuse a component but do not require the software engineer to understand the internals of the component. The component is accessed through its externally visible interface and its internal structure is fully encapsulated. Black-box adaptation techniques are only concerned with adapting the interface of the component, rather than the internal structure of the component.

Aggregation When defining a large component with much behaviour, one can define existing components that provide part of the required behaviour as part of the component, i.e. aggregation. Although aggregation is more oriented towards reuse than towards adaptation, the newly developed code in the aggregating component enables the reuse of components that otherwise might not have matched the application. However, aggregation is not a pure adaptation technique, but *also* suitable for component adaptation.

Wrapping Wrapping also declares one or more components as part of an encapsulating component but this component only has functionality for forwarding, with minor changes, requests from clients to the wrapped components. There is no clear boundary between wrapping and aggregation, but wrapping is used to adapt the behaviour of the enclosed component whereas aggregation is used to compose new functionality out of existing components providing relevant functionality. An important disadvantage of wrapping is that it may result in considerable implementation overhead since the complete interface of the wrapped component needs to be handled by the wrapper, including those interface elements that need not be adapted. Also, others, e.g. [7], have identified that wrapping may lead to excessive amounts of adaptation code and serious performance reductions.

2.4 Evaluating Conventional Techniques

In Table 1, an overview of the conventional adaptation techniques is presented that indicates how well each technique fulfils the specified requirements. From the table, one can identify that some requirements are dealt with well by the black-box techniques but not so well by the white-box techniques and visa versa.

The copy-paste technique, as well as inheritance, is transparent since the reused and adaptation behaviour are merged in a single entity. However, on the other requirements, the white-box adaptation techniques do not score so well. The black-box adaptation techniques are not transparent, since they encapsulate the adapted component. Obviously, these techniques are black-box by definition and wrapping is even composable since a wrapped component can again be wrapped by another wrapper adapting different aspects of the original component. Configurability and reusability are not well supported by these techniques since no distinction between generic behaviour and component-specific behaviour is made. Due to this, it is not possible to separate the generic aspects and apply them for a different component.

Requirement	Copy-Paste	Inheritance	Aggregation	Wrapping
transparent	+	+	-	-
black-box	-	-	+	+
composable	-	-	-	+
configurable	-	-	-	-
reusable	-	-	+/-	+/-
efficient	-	+	+	-

Table 1: Conventional adaptation techniques versus the identified problems and requirements

Concluding, none of the conventional component adaptation techniques fulfils the requirements that are required for effective component-based software engineering. Therefore, one may deduce alternative approaches are required. In this paper we propose superimposition as a novel technique to component adaptation.

3 Component Adaptation through Superimposition

We propose *superimposition* as a novel technique to adapt components in a component-based system. The notion underlying superimposition is that a component and the functionality adapting the component are two separate entities on the one hand and need to be very tightly integrated on the other hand. We believe that, in addition to a set of reusable components, a set of reusable component adaptation types is required. These adaptation types should be configurable and composable with each other to allow for complex component adaptations. In the next section, several types of component adaptation are identified and presented.

3.1 Component Adaptation Types

During our work on component adaptation, we have identified three typical categories of component adaptation, i.e. component interface changes, component composition and component monitoring. In the sections below, each of these categories is discussed in more detail.

Changes to Component Interface A typical situation in component-based system construction is when a component in principle could be reused in the system at hand, but its interface does not match the interface expected by the system. In such situations, the interface of the component needs to be adapted to match the expected interface. Below, some typical examples of component interface adaptation are presented.

- **Changing operation names:** Perhaps the most typical problem when reusing a component is that the names of some of the operations provided by the component do not match the expected interface. This problem has been identified by many software engineers. The *Adapter* design pattern [4] has been defined to address this but its implementation suffers from several problems as we identified in [1].
- **Restricting parts of the interface:** A second change to the component interface that a component may require is the exclusion of a part of the interface. In the reusing context, a part of the interface may not be relevant or even counter-productive (e.g. for typing reasons) it would be accessible by clients of the component. Adaptation of the component should then restrict access to the excluded operations.
- **Client- and state-based restriction:** In systems where a component is used by clients of various types, the component may need to act in several roles, see e.g. [11]. This requires the component to present a tailored interface to each client type, i.e. each client has only access to that part of the interface that it requires. This we refer to as client-based interface restriction. In addition, parts of the interface of the component may be accessible or restricted based on the *state* of the component.

Component Composition Components are intended for composition to form larger structures. Sometimes, the components have to be composed such that the resulting structure seems a single component from the system's perspective. Below, three types of component adaptation relevant for component composition are discussed.

- **Delegation of requests:** The easiest way for a component to providing required services not available within the component itself is to delegate a request for such a service to another component that is able to provide the requested service. To achieve this, the component needs to be extended with behaviour that delegates certain requests to other components.
- **Component composition:** In cases where two components need to be more structurally integrated, the two components can be aggregated in a encapsulating component. However, the encapsulating component needs to delegate requests to the contained components such that the requirements of the system are fulfilled.
- **Acquaintance selection and binding:** No component is an island, i.e. virtually all components require other components, acquaintances, to provide them with services in order to be able to deliver the functionality needed by the system. However, since the designers of reusable components are unable to make all but minimal assumptions about the context in which the component will operate, the binding of the acquaintances required by the component is often performed in an ad-hoc manner, e.g. when the component is instantiated. As we identified in [2], the traditional acquaintance binding omits several important aspects. Component adaptation should allow for flexible, expressive specification of the way acquaintances are selected and bound.

Component Monitoring Component monitoring implies that other components are in some form notified or invoked when certain conditions at the monitored component occur. Three examples of monitoring are described below. The latter types are specialisations of the earlier ones.

- **Implicit invocation:** This type actually is the general adaptation type for component monitoring. The concept of implicit invocation is concerned with notifying relevant components, either directly by message sending or indirectly through event generation, whenever certain conditions or actions take place at the monitored component.
- **Observer notification:** The *Observer* pattern [4] provides this type of behaviour but it presumes that the software engineer knows, when defining an object, that it will be observed by other objects. In component-based system construction, the reused components sometimes need to be observed, but generally the component is not prepared for this. Therefore, the observer pattern functionality needs to be superimposed on the component so it can be used as an observed component.
- **State monitoring:** In some cases, dependent components do not want to be notified for every state change in the observed component, but only when the component state exceeds certain boundaries. The conventional observer pattern behaviour is not prepared for this, but using superimposition it is well feasible to implement this behaviour. This allows the software engineer reusing the component to specify in what state regions the dependent components need to be specified.

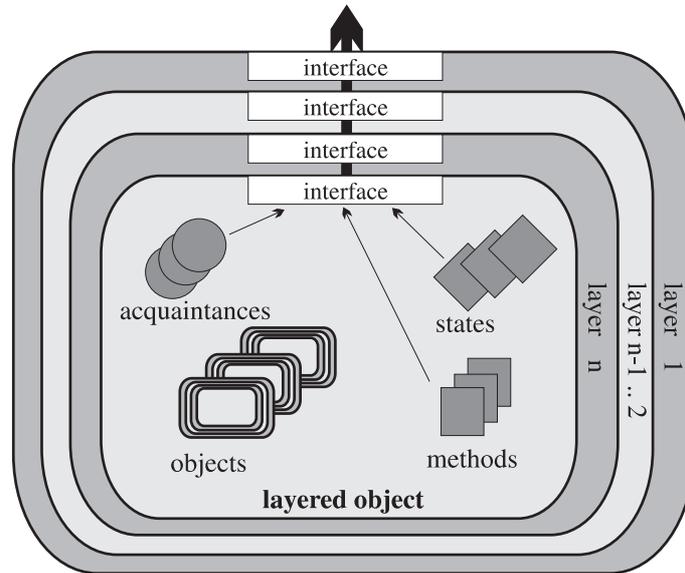


Figure 1: The layered object model

4 Superimposition using the Layered Object Model

Since traditional object and component models are unable to implement component adaptation through superimposition, more advanced models are required. One such model is the *layered object model* (**LayOM**) that we have been working on for several years. **LayOM** provides direct language support for superimposition and we have implemented most of the types of component adaptation in the previous section as part of the language using layer types.

The layered object model is an extended component object model, i.e. it defines in addition to the traditional object model elements, additional parts such as layers, states and categories. In Figure 1, an example **LayOM** object is presented. The layers encapsulate the component, so that messages sent to or by the component object have to pass the layers. Each layer, when it intercepts a message, converts the message into a passive message object and evaluates the contents to determine the appropriate course of action. Layers can be used for various types of functionality, either proactively or in reply to a received message. Layer classes have, among others, been defined for the representation of relations between classes and objects, design patterns [1], acquaintance handling [2] and superimposing behaviour.

A *state* in **LayOM** is an abstraction of the internal state of the object. In **LayOM**, the internal state of a component is referred to as the *concrete state*. Based on the component's concrete state, the software engineer can define an externally visible abstraction of the concrete state, referred to as the *abstract state* of a component. An *acquaintance* is an expression that defines a set of components that are treated similarly by the component. This set of components treated as equivalent by the component we denote as *acquaintances*.

Layers are the entities that provide superimposition functionality to components. Individual instances of components can be extended with layers. Since layers intercept

messages sent to and from the component, layers are able to superimpose certain functionality on the component. An example is the *Adapter* layer type, shown below. An instance of class *Adaptee* is declared and a layer of type *Adapter* is added to object. The layer will intercept the messages sent to the object and change certain message selectors so that the component can interpret them.

```
// object declaration
adaptedAdaptee : Adaptee with layers
    adapt : Adapter(accept mess1 as newMessA,
                   accept mess2, mess3 as newMessB);
end;
```

A second example is the *Observer* layer type. Below an instance of class *Point* is shown that is extended with an *Observer* layer that will notify interested objects when certain methods of the object are invoked.

```
aPoint : Point with layers
    st : Observer(notify after on setX on aspect "X-axis",
                 notify after on setY on aspect "Y-axis",
                 notify after on moveTo on aspect "Location");
end;
```

Due to reasons of space, the facilities for component adaptation provided by **LayOM** are only discussed very brief. We refer to [1, 2] and to the indicated WWW page for more information.

5 Related Work

The notion of adapting reusable components to match the requirements of the application at hand is not extensively studied in the component-based software engineering community. Some object models provide *before* and *after* facilities that allow the software engineer to add pre- and post-behaviour to the execution of an operation in a component. In general, however, adapting components using the existing component object models do not fulfil the requirements identified in section 2.1.

The notion of superimposition has earlier primarily been used in the context of distributed systems, e.g. [3] and [8]. There it is used to indicate the additional, superimposing control over some algorithm.

Since superimposition is a novel technique, no existing implementations of superimposition exist besides the layered object model. However, meta-object protocols [9] can be viewed as types of superimposing behaviour for object-oriented systems. In general, reflective languages such as CLOS are suitable to implement superimposition.

6 Conclusion

Component-based software engineering is becoming increasingly important as a means to efficiently create applications from reusable components. Most traditional approaches assume that components are reused "as-is" in these applications, but in practice "as-is" reuse is very unlikely to occur and most components need to be adapted to match the requirements of the application. Component adaptation techniques should be transparent, black-box, composable, configurable, reusable and efficient to use. Conventional

techniques for adapting components can be categorised into white-box and black-box. Examples of the former are copy-paste and inheritance, whereas aggregation and wrapping are examples of black-box adaptation techniques. These approaches do not fulfil most of the identified requirements.

To deal with the problems and to meet the requirements that were identified, a new component adaptation technique, *superimposition*, was introduced. An object superimposition S of B over O is defined as the additional overriding behaviour B over the behaviour of a component object O. Different from, e.g. inheritance, a single unit of superimposed behaviour can change several aspects of the basic component's behaviour. One of our conclusions is that, in addition to a set or reusable components, component-based software engineering requires a set of reusable component adaptation types. To exemplify this, several types of adaptation behaviour are presented, categorised into component interface adaptation, component composition and component monitoring.

Superimposition is implemented as a language construct in the layered object model (**LayOM**) through the notion of layers. **LayOM** is an extended component object model that, next to instance variables and methods, contains parts such as states, categories and layers. The extended expressiveness of **LayOM** provides the software engineer with powerful component adaptation types through superimposition.

Bibliography

- [1] Bosch, J.: Design Patterns as Language Constructs. Accepted for publication in the Journal of Object-Oriented Programming, November 1996.
- [2] Bosch, J.: Object Acquaintance Selection and Binding. submitted, August 1997.
- [3] L. Bougé, L. Francez, N.: A Compositional Approach to Superimposition. Proceedings POPL'88, pp. 240-249, 1988.
- [4] Gamma, E., Helm, R., Johnson, R., Vlissides, J.O.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
- [5] Goldberg, A., Robson, D.: Smalltalk-80 - The Language. Addison-Wesley, 1989.
- [6] Helm, R., Holland, I., Ganghopadhyay, D.: Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. OOPSLA '90, pp. 169-180, 1990.
- [7] U. Hölzle, U.: Integrating Independently-Developed Components in Object-Oriented Languages. Proceedings ECOOP'93, pp. 36-56, 1993.
- [8] Katz, S.: A Superimposition Control Construct for Distributed Systems. ACM Transactions of Programming Languages and Systems, Vol. 15, No. 2, April 1993.
- [9] Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. The MIT Press, 1991.
- [10] H. Lieberman, H.: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Proceedings OOPSLA '86, pp. 214-223, 1986.
- [11] Reenskaug, T., Wold, P., Lehne, O.A.: Working With Objects: The Ooram Software Engineering Method. Prentice Hall, 1995.

Formal Methods for Component Software: The Refinement Calculus Perspective

Martin Büchi

Turku Centre for Computer Science
Lemminkäisenkatu 14A, 20520 Turku, Finland
mbuechi@abo.fi

Emil Sekerinski

Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14A, 20520 Turku, Finland
esekerin@abo.fi

We exhibit the benefits of using formal methods for constructing and documenting component software. Formal specifications provide concise and complete descriptions of black-box components and, herewith, pave the way for full encapsulation. Specifications using abstract statements scale up better than preconditions and allow for ‘relative’ specifications because they may refer to other components. Nondeterminism in specifications permits enhancements and alternate implementations. A formally verifiable refinement relationship between specification and implementation of a component ensures compliance with the published specification. Unambiguous and complete contracts are the foundation of any component market.

1 Introduction

The separation of specifications/interfaces and implementations of components is a prerequisite for the establishment of component software. It alleviates the necessity to distribute source code, thereby protects the implementation know-how and avoids overspecification. Overspecification basically prohibits future enhancements and alternate implementations. Furthermore, separate specifications enable the component integrator to understand the functionality without having to examine the source code.

The lack of easily and quickly understandable, concise, and complete specifications is the chief reason, why the advantages of the separation between specifications and implementations are not commonly exploited. Most current interface-description languages (IDLs) are limited to expressing syntactical aspects such as number, names, and types of parameters only. For example, the C library routines *strcpy* and *strcat* have the same interface (signature) and are, therefore, up to their name — a vague hint at the functionality — indistinguishable. The IDLs completely ignore the behavior of components which is usually given as incomplete, ambiguous, partly overspecific, and

often outdated textual description¹. Incompleteness forces the component integrator to derive additional properties by trial and error which might be invalidated in future versions of the component. Ambiguity often remains undetected in an informal setting and causes mysterious errors; if noticed, forces the integrator to make unfounded assumptions. Overspecification unnecessarily restricts future enhancements. Incompleteness, ambiguity, and overspecification hinder alternate implementations — the ground stone of any component market. Without the possibility of automated consistency verification, informal descriptions are rarely kept up to date. Due to these deficiencies, programmers request components as source code. Overwhelmed with implementation details — in addition to the aforementioned problems of this approach —, they then often choose to reinvent the wheel rather than reuse the badly specified existing components.

Formal specifications can solve these problems. The creator of a component can test, whether based solely on the specification the component may be appropriately used. Ambiguities can be detected by consistency proofs. Overspecification can more easily be detected in a concise formal language. Formal verification, here in the form of refinement proofs, guarantees that the implementation actually behaves as specified. Furthermore, a specification which is created before the component is implemented, can facilitate a structured development and, thereby, create more general, robust and efficient components and often also helps to save costs. “When quality is pursued, productivity follows,” K. Fujino, as quoted in [18]. For a recent overview of formal methods success stories see [8, 16].

The adaptation of formal specifications has been slow because of difficult notations which differ too much from implementation languages and lack of tool support, but also due to ignorance and prejudice. We aim for a lightweight approach to formal methods, based on a rigorous and general formalism, but without an overwhelmingly rich expressiveness of the language, modeling, and analysis in favor of lower costs in terms of time, know-how, and money. We deviate from the classic dogmatic view on formal methods, without abandoning the foundations.

Section 2 makes a plea for formal specifications as contracts, Sect. 3 shows why nondeterminism is also relevant for practitioners. Refinement between specifications and implementations to ensure compliance and refinement between different versions of a specifications are the topics of Sect. 4. Section 5 points to related work and Sect. 6 draws the conclusions.

2 A Plea for Formal Contracts

The buyer of a microprocessor or another chip usually requests a detailed description in form of a data sheet and, quite commonly, also an executable specification in form of a VHDL program. These descriptions form a contract, if a sale takes place. They describe all relevant information for deployment, such as form factor, voltage, and signal delay in a standard way, that does not require interpretation or understanding of the physics required to build the chip. Contrast this with the typical description of an ActiveX software component: Incomplete plain textual descriptions augmented with a formal part that merely describes the number and types of parameters. The customer has no possibility to verify in advance whether the desired part meets the requirements. He often spends hours of trial and error to find out how the component must be used. He relies on testing of a few cases as the only way to gain confidence.

¹We denote by interface the syntactical aspects only and by contract both the interface and the behavioral specification.

	Obligations	Benefits
Client	Ensure precond.	Assume postcond.
Server	Ensure postcond.	Assume precond.

Figure 1: Obligations and benefits from pre- and postconditions

Nobody and nothing guarantee that he uses only functionality which will continue to exist in future versions. Hence, there is an urgent need for better contracts! A good contract is clear, complete, and concise. A bad contract is ambiguous, misses important points, lays down irrelevant details, and is unnecessarily long. That current contracts, respectively in our terminology interfaces with textual addition, are too weak has also been acknowledged at WCOP'96 [26]. The lack of standardized contracts for software components is due to the high degree of freedom compared to hardware, the immaturity of the field, the difficulties in automated verification, and the — partly unnecessary — complexity and ill-definedness of common programming languages, which further complicates verification.

Jézéquel and Meyer [11] recently argued that the crash of the Ariane 5 [2] was due to a reuse specification error. A poorly documented limitation in a component originally designed for the Ariane 4 with different physical requirements caused the error. Jézéquel and Meyer conclude, that reuse without a contract is sheer folly. Yet, contracts are the most important non-practice in component software. Clearly, white-box components do not solve these problems for large systems as they overwhelm the designer with details, rather than providing suitable abstractions.

A simple and popular form of contracts is that of assumptions, called preconditions, and promises, called postconditions. If a component is used in a correct manner, it has to satisfy its contract, i.e. establish the promised postcondition. If, however, its preconditions are not met, it has no obligations whatsoever and is free to behave arbitrarily (Fig. 1). If a system consisting of several components fails to perform the requested job, the failure can be attributed either to a component not fulfilling its contract or to the integration, i.e., even if all components behave as specified the whole system does not perform correctly. Without formal contracts, locating such errors is much more involved.

Pre- and postconditions that are only checked at runtime help to locate errors, but do not prevent them as static analysis does. A program can still fail at a customer's site with input values which have not been tested. Programmers annotate their programs with pre- and postconditions in order to make them more reliable. During test runs, they are checked at run-time, but for the production version, these checks are often disabled for efficiency reasons. This is like having lifeboats on a ship for a test cruise but getting rid of them for a transatlantic cruise with passengers because of the additional load. Static analysis, on the other hand, allows only the removal of lifeboats which will provably never be used. Because of the deficiencies of run-time only checking, programmers are not inclined to use specifications at all.

Pre- and postconditions being predicates, they cannot contain calls to other methods, except pure functions. This means that using pre- and postconditions one has to reinvent the wheel afresh for each method, rather than being able to build upon other specifications. Specifications in form of abstract statements are not affected by this scalability problem. Consider the partial specification of component Buffer using abstract statements:

```

component Buffer ...
  b : set of Item
  print(d: Device) = for all x in b do d.print(x) end
end Buffer

```

If we were to specify the same component using pre- and postconditions, we would have to expand the definition of the base type `Device`'s `print` method incurring a number of disadvantages. The specification of how `print` ultimately sets the pixels on a device would be rather lengthy and not of our interest here. We lose the information that a method of `d` is invoked. Reasoning about the program, we cannot use the knowledge that `d` is of (behavioral) subtype of `Device` with a more deterministic specification. Pre-postcondition specifications contradict encapsulation and specialization.

Specifications by abstract statements come close to contracts as proposed by Helm et al. [10]. Contracts of Helm et al. specify "behavioral dependencies" between objects in terms of method calls and other constructs. Contracts are expressed in a special purpose language and then have to be linked to the underlying programming language. By contrast, we like to see abstract statements as a moderate extension of the underlying programming language for expressing contracts.

Changes to the specification of the `print` method, e.g. improved version decreasing nondeterminism, are not automatically reflected in the specification of `Buffer`. Pre-postconditions do not support 'relative' specifications in the sense of relying on previous specifications. The loss of self-containedness of abstract statement specifications can easily be compensated by a specification browser supporting in place expansion or hypertext-like facilities. Abstract statements also lend themselves to grey-box specifications, which reveal parts of the internals, such as call-sequences [6].

The process of writing a formal specification often leads to more generally useful, easier to integrate, and longer-lived components. Rough edges, special cases, and anomalies resulting from implementation difficulties and lack of overview during implementation can often be detected and eliminated by a specification.

For example, the above specification of `Device` states that for all elements in `b`, the method `print` is called in an arbitrary order. No element is printed twice, since a set contains an element at most once. If this is desired, we should have used a bag (multiset) rather than a set. If we like that the elements are printed always in the same order, we should have used a sequence rather than a set and an iteration in `print`. The specification also states that printing an empty buffer is a no-op rather than an error.

By writing the specification, or at least parts of it, before starting the implementation, one can benefit from the commonly advocated advantages of a structured development process [22]. Cliff Jones [13] argues that formalism employed to justify early data structure and design decisions which helps avoid the most costly errors is the most important application of formal methods. Specifications written after the implementation tend to lay down irrelevant details of the specific implementation and lack the desired degree of abstraction. However, as with frameworks, an iterative approach is usually preferred over the waterfall model in order to get very general components. The number of iterations can be reduced by formal specifications which help to eliminate implementation quirks.

For some reasons forced to use a component that comes without a formal contract, it might even be worthwhile to write a specification of it as it is perceived and used. Such a 'contract assumption' can greatly simplify testing and reduce the time to evaluate the suitability of new and alternate versions of the component.

A component should not only formally specify its own contract, but also the (min-

imal) contracts of its required components. A calendar component might require a database component which satisfies a certain contract [3, 27]. The component integrator can choose such a component, or — in a more dynamic scenario — the calendar component can ‘shop’ for the desired component at runtime. Formal specifications of required and existing components simplify also the creation of wrappers/adaptors.

3 Nondeterminism: Avoiding Overspecification

Nondeterminism is an approach to deliberately leave a decision open, to abandon the exact predictability of future states. As such, nondeterminism appears to be neither commonly desirable nor is it used in implementation languages. On the other hand, nondeterminism is a fundamental tool for specifications to avoid laying down unnecessary details².

A nondeterministic specification leaves more choice for the implementation, which can be used for optimizations. Even if this degree of freedom is not used in the envisaged first implementation, it greatly increases the likelihood that future enhancements and alternate implementations can be made compliant with the specification. The earlier specification of the component Buffer is an example.

Nondeterminism often enhances the comprehensibility of specifications because the reader does not have to wonder why something has to be exactly in a certain way, when other choices would be as good. Many things are actually nondeterministic and should be acknowledged and specified as such.

Nondeterminism from an outside perspective often stems from information hiding, where the actual implementation is deterministic. A SQL database query without any sorting options returns an arbitrarily sorted list of records; a square root function returns an arbitrary value satisfying the specified precision. Both implementations are deterministic, but the outcomes are determined by hidden state components and implementation details.

Nondeterminism can also be present in the implementation. This occurs typically when the implementation is concurrent, for example by distributed objects. Given only the specification, it does not matter for a user of the component if and when the nondeterminism is resolved. This way, a large variety of implementations become valid refinement of the specification.

We can also interpret nondeterminism as ‘free will’ of a component which can in no way be influenced from the outside. Writing a combined specification consisting of existing components and a custom ‘glue component’ which we implement ourselves, we have to distinguish between two forms of nondeterminism. Nondeterminism within existing components which is beyond our control, called demonic nondeterminism, and nondeterminism in our custom component which we can control in our favor, called angelic nondeterminism. In this sense, we can consider program execution as a game, the rules of which are given by the specification [5]. Demonic choices are moves made by an opponent (the existing component), and angelic choices are our moves. The combined specification is correct, if we can make moves such that we can achieve the desired goal, no matter what the opponent does. Hence, such a combined specification can help to decide whether a given component is suitable to solve a certain task.

²Nondeterministic constructs would leave more choices for compiler optimizations also. This could be beneficial especially in the case of portable executable formats and just-in-time compilers, where static optimizations for a specific architecture are not applicable.

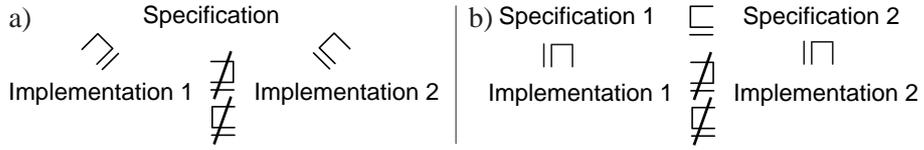


Figure 2: Refinement relationships

4 Refinement: Ensuring Compliance with Specification

Employing formal specifications, we want to make sure that the implementation actually complies with its specification or, more precisely, is a refinement thereof [5]. A statement T refines a statement S , if considering the observable input-output behaviour, the output of T for a given input would be possible with S as well. Taking the possibility of nondeterminism into account, we formally define that $S \sqsubseteq T$ (S is refined by T) as

$$S \sqsubseteq T \stackrel{\text{def}}{=} \forall q. wp(S, q) \subseteq wp(T, q)$$

where q is a set of states and $wp(X, q)$ denotes the weakest precondition of q with respect to statement X , i.e., the set of states from which X is guaranteed to terminate in a state of q . Refinement is reflexive, transitive, and antisymmetric [5].

A component D is a refinement of component C , if the observations we can make when using D would be possible with C as well. Let $S[C]$ be a statement using component C . Formally, we define $C \sqsubseteq D$ (C is refined by D) by:

$$C \sqsubseteq D \stackrel{\text{def}}{=} \forall S. S[C] \sqsubseteq S[D]$$

If an implementation of a component constitutes a refinement of its specification, any client designed according to the specification will work with the implementation. This is exactly what is implied by the above definition. On the other hand, two implementations of a component which are both refinements of the same specification might not be in a refinement relationship to each other (Fig. 2 a). Hence, it is important that clients only rely on properties guaranteed by the specification. Testing cannot uncover reliance on unspecified implementation features, only formal analysis can.

Nierstrasz and Tsichritzis [23] remark that encapsulation is violated if clients of a software component must be aware of implementation details not specified in the contract in order to make correct use of a component. In particular, if changes in the implementation that respect the original contract may affect clients adversely, then encapsulation is violated. Clients which rely only on properties specified in the contract never fall into this trap. However, a contract may be too weak so that the corresponding components can not be used intelligently.

Unfortunately, fully automatic proving of refinements similar to type checking performed by a compiler does not seem to be feasible in the foreseeable future. Because of this, contracts are approximated by interfaces, guaranteeing only syntactic compatibility. The B-Toolkit [4] and Atelier-B [25], two environments for the B method [1], show, however, that semiautomated refinement proofs are feasible for industrial applications. Automatic proofs of refinement are an absolute necessity for components which at runtime shop for other components satisfying a required contract.

Instead of a full refinement proof, automatic test data generation from the specification, approximate reasoning with upper and lower bounds, finite state analysis with binary decision diagrams based on equivalence classes, and relative debugging can be employed to gain confidence in the correctness and to isolate parts meriting formal proofs [7, 12, 9]. In relative debugging, an alternative to a full refinement proof stemming from the field of scientific computing [21], the specification and the implementation are executed in lockstep and their states are compared after each statement according to a mapping relation.

If we want to better a component, we have to incorporate the improvements into the specification. Otherwise, they cannot be used by any clients which are only allowed to rely on features guaranteed by the specification. An advanced component should also be usable by an old client, which knows nothing about the improvements. If the new specification is a refinement of the old specification and the new implementation refines the new specification, then by transitivity of refinement it also refines the old specification. Hence, refinement is also an important relationship between different versions of a component's contract. However, an implementation of the new specification does not necessarily refine an implementation of the old specification (Fig. 2 b). In a world where progress is acceptable only if it is compatible with the current state, refinement is a crucial relationship.

We have not addressed performance, but the separation of specifications and implementations, which are refinements thereof, permits the replacement of inefficient components by efficient components which adhere to the same specification without any modification to the clients. It also allows for several components providing the same functionality which are tuned for different usages, e.g., set implementations as arrays and as lists. We could even specify an advanced set component as a refinement of the basic set component which provides a meta level method for controlling performance-relevant mapping decisions [14]. An implementation of the advanced set component might package the two set implementations into a single component and select the actual data structure according to the preferences set through the meta method.

5 Related Work

The Interface Specification Language (ISL), a pure extension of the CORBA IDL, developed in the Component-Based Software Engineering project at CSTaR Software Engineering Lab supports the descriptions of pre-/postconditions of operations, invariants and protocols of interfaces [15]. ISL opts for featurism, including multiple inheritance across components, rather than simplicity as we promote it. ISL is a sublanguage of the Architecture Specification Language, which also includes the Glue Specification Language and the Configuration Specification Language.

Participants in the Composable Software Systems project at Carnegie Mellon try to specify not only functional behavior of components, but also reliability, performance, and security aspects [24]. They call concepts similar to refinement behavioral subtyping for classes [17] and matching for components [27].

Bertrand Meyer propagates design by contract for component software, albeit of a less formal nature [19, 20]. The Object Systems Group at the University of Geneva under Nierstrasz and Tsichritzis have researched a number of applications of semi-formal methods to component software [23].

6 Conclusions

We have argued that only formal contracts paired with refinement can guarantee full encapsulation of software components, which is the base for improved and alternate implementations. Formal contracts lead to a more structured development, more orthogonal and, hence, longer-lived and more generally useful components, often at a lower cost.

Nondeterminism is a necessity for providing freedom of implementation. Refinement guarantees that implementations adhere to their specifications and that new versions are plug-compatible. Abstract statements do not have the scalability problems of pre-postcondition specifications because they allow for external calls.

Formal methods are needed to compensate the loss of the closed-world assumption and the impossibility to test a component in all possible environments. They are, however, no universal panacea nor is their application very simple, but we regard them as a necessity for the establishment of component software.

The paper presented at the workshop (<http://www.abo.fi/~mbuechi/>) also contains sections on the specification of invariants and temporal properties and on the design of specification languages, which are left out of this version.

Acknowledgments We would like to thank the referees for helpful comments, Wolfgang Weck and Ralph Back for a number of fruitful discussions and inspiring ideas, and Lars Nielsen and Marcel Mettler for their help with this paper.

Bibliography

- [1] J. R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] European Space Agency. Ariane 5: Flight 501 failure, July 1996. <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>.
- [3] Ásgeir Ólafsson and Doug Bryan. On the need for “required interfaces” of components. In M. Mühlhaeuser, editor, *Special Issues in Object-Oriented Programming*, pages 159–165. dpunkt Verlag Heidelberg, 1997. ISBN 3-920993-67-5.
- [4] B-Core. *B-Toolkit*. England, 1995.
- [5] R. J. R. Back and Joackim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer Verlag, to appear 1997.
- [6] Martin Büchi and Wolfgang Weck. A plea for grey-box components. In *Foundations of Component-Based Systems '97*, 1997. <http://www.abo.fi/~mbuechi/>.
- [7] Marsha Chechik and John Gannon. Automatic analysis of consistency between requirements and designs. In *Tech. Rep. CS-TR-3394.1, University of Maryland, College Park*, 1995. <http://www.cs.utoronto.ca/~chechik/>.
- [8] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(10), December 1996.

- [9] Craig A. Damon, Daniel Jackson, and Somesh Jha. Faster checking of software specifications by eliminating isomorphs. In *Proceedings ACM Conference on Principles of Programming Languages*, 1996. <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/nitpick/www/home.html>.
- [10] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP '90 Conference on Object-Oriented Programming Systems, Languages and Application*, pages 169–180, October 1990.
- [11] Jean-Marc Jézéquel and Bertrand Meyer. Put it in the contract: The lessons of ariane. *IEEE Computer*, pages 129–130, January 1997.
- [12] Craig A. Damon, Somesh Jha, and Daniel Jackson. Checking relational specifications with binary decision diagrams. In *Proc. Foundations of Software Engineering '96*, 1996. <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/nitpick/www/home.html>.
- [13] Cliff B. Jones. A rigorous approach to formal methods. *IEEE Computer*, pages 20–21, April 1996.
- [14] Gregor Kiczales. Why are black boxes so hard to reuse. In *Proceeding of OOPSLA'94*, 1994. <http://www.parc.xerox.com/spl/projects/oi/towards-talk/transcript.html>.
- [15] W. Kozaczynski and J. O. Ning. Concern-driven design for a specification language. In *Proceedings of the 8th International Workshop on Software Specification and Design*, Berlin, Germany, March 1996.
- [16] WWW Virtual Library. Formal methods. <http://www.comlab.ox.ac.uk/archive/formal-methods.html>.
- [17] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), November 1994.
- [18] Carlo Ghezzi, Dino Mandrioli and Mehdi Jazayeri. *Software Engineering*. Prentice Hall, 1991.
- [19] Bertrand Meyer. Applying ‘design by contract’. *IEEE Computer*, 25(10):40–51, October 1992. See also <http://www.eiffel.com/doc/manuals/technology/contract/index.html>.
- [20] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
- [21] David Abramson, Ian Foster, John Michalakes, and Rok Susic. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, November 1996.
- [22] Caroll C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.
- [23] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice Hall, 1995.

- [24] David Garlan, Daniel Jackson, Mary Shaw, and Jeannette Wing. Composable software systems, 1996. <http://www.cs.cmu.edu/~Compose/> and <ftp://ftp.cs.cmu.edu/project/compose/brief.ps>.
- [25] Stéria Méditerranée. *Atelier-B*. France, 1996.
- [26] Clemens A. Szyperski and Cuno Pfister. Component-oriented programming: WCOP'96 workshop report. In M. Mühlhaeuser, editor, *Special Issues in Object-Oriented Programming*, pages 127–130. dpunkt Verlag Heidelberg, 1997. ISBN 3-920993-67-5.
- [27] Amy M. Zaremsky and Jeannette M. Wing. Specification matching of software components. In *SIGSOFT Foundations of Software Engineering*, October 1995. Also CMU-CS-95-127.

Design and Evaluation of Distributed Component-Oriented Software Systems

Michael Goedicke and Torsten Meyer

Specification of Softwaresystems, Dept. of Mathematics and Computer Science
University of Essen, Germany
{goedicke,tmeyer}@informatik.uni-essen.de

With the emergence of the Object Management Group's CORBA (Common Object Request Broker Architecture) and comparable platforms heterogeneous and distributed computing is facilitated. Providing location-, language-, and platform-transparency, CORBA promotes the independent development of software components and standardizes the interaction between components. However, the design of entire software architectures for distributed component-oriented software systems with complex client-server relationships is still a major problem. In this contribution we consider an approach to describe the architecture of distributed software systems. This approach is based on a component model of software which contains additional information about distribution. Rather than describing the distribution properties within a component most of these properties are stated with the use relation between components which may be local or remote. We sketch how this design description can be transformed into a distributed object-oriented implementation according to OMG's CORBA standard. We discuss how a performance model can be derived systematically from an architecture description. Thus the design of complex, hierarchically structured distributed software systems can be assessed wrt. response time of remote operation invocations, for example ¹.

1 Introduction and related work

In constructing software systems various development stages are passed (cf. figure 1). While general experience shows (cf. B. Boehms spiral model) that these stages are visited more than once it is important to assess designs i.e. software architectures quite early on (cf. [8]). The role of the software architecture for a long-lived product should not be underestimated (cf. [7], [14]).

In this paper, we present our architectural framework for developing component-oriented distributed systems which fits well into the entire development cycle shown in figure 1. We sketch how distributed architectures can easily be designed using our architecture description language Π . The step to implement a distributed architecture is realized by transforming software components in Π to distributed objects according to the Object Management Group's standard CORBA (Common Object Request Broker Architecture). Since the resulting implementation structure corresponds to the design structure, knowledge about the execution of the distributed system may feedback giving new insights at the design and requirements stage.

¹This work is partly funded by the DFG project QUAFOS, contract MU1158/2-1.

Related research w.r.t. self-contained and independent software components is done in many places. A prominent example is ROI (Regis Orb Implementation), an integration of the architecture description language REGIS/DARWIN with IONA Technologies CORBA implementation ORBIX (cf. [3]). In contrast to Regis/Darwin, Π has a richer language for describing semantic properties in interfaces.

Architectural support on top of CORBA is also provided in [12] using design patterns and application frameworks (cf. [6], [1]). As patterns and frameworks are often specialized w.r.t. sets of horizontal functionalities or vertical areas of application domains, our approach is more general. Our approach is based on formally specifying software architecture while patterns and frameworks are less rigorously founded.

In chapter 2 we show how the design of distributed component-oriented software systems can be described using the Architecture Description Language Π . It is also briefly shown how Π and CORBA are integrated. We introduce our concept for simulation-based evaluation of the design architecture in chapter 3.

2 The Architecture Description Language Π

In this chapter we will present Π , our architecture description language (ADL), which supports the design of distributed component-oriented software systems. It provides concepts for separating the development of distributed independent software components from the interconnection and configuration of such components: although component dependency requirements can be stated with a single independent component, the explicit connection structure can be defined at a different point in the design process.

According to [9], [10] a software *component* is a unit which provides its clients with services specified by its interface and encapsulates local structures that implement these services. Furthermore, it may use services of other components to realize the exported ones. Each component encapsulates one or more Abstract Data Types (ADTs), hence an object-based structuring of the whole architecture is enforced. Collections of components connected via use relations are called *configurations*. Also, configurations have in principle the same interfaces to their environment as single components and thus may be used as components hierarchically.

In Π , each component is described by four sections (cf. figure 2). The *export section* gives an abstract image of the component's realization; the abstract data types stated here are public and may be used by other components. The *body section* describes the realization of a component; here, the construction of the exported abstract data types is encapsulated. According to the concept of formal import, only requirements to

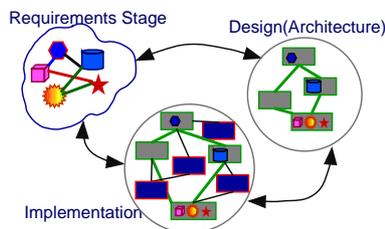


Figure 1: Stages of development.

Common Parameters	Export
	Body
	Import

Figure 2: Sections of a component in Π .

imported abstract data types are specified in the *import section*. While configurations of components are built, the import section has to be actualized with export sections of potential server components via use relations. Finally, in the *common parameters section* abstract data types are stated which are imported and exported unchanged.

Π is a multi-formalism language and single views can be seen as overlapping partial specifications of a component. Each section can be specified by four views: the *type view* describes the component's invariant properties (according to execution of operations) by means of algebraic specification techniques, the *imperative view* defines imperative operation signatures and algorithms, the *concurrency view* specifies possible orderings of operation executions, the *interaction view* encapsulates information according to distribution of components.

Due to the fact that in Π each component specification is parameterized by its formal import it can be used with different parameter actualizations. A component developed and viewed in isolation is some kind of component template in contrast to the same component used within the specific context of the other components. Different instantiations of a component which can be connected via use relations are called *component incarnations* and the isolated component template is called *Concurrently Executable Module* (CEM). Thus, our approach takes an open world perspective: according to the concept of formal import a clear distinction can be made between the independent development of self-contained CEMs and the connection of component incarnations. During the development of a single CEM only requirements to imported services are described, the actual mapping from a component incarnation's requirements to services offered by potential server incarnations is made within the component connections. However, within a configuration of component incarnations not all open imports have to be actualized, but can be connected to the import requirements of the entire configuration in order to allow other components to be linked to the configuration at a different time. For generating an implementation or a performance model from the design architecture a closed world assumption is still needed, but future work will address dynamically evolving systems also at the implementation stage.

Now we present a very simple introductory example from the area of Computer Supported Cooperative Work (CSCW): let us consider a software component representing a group of people engaged in a common task. Thus we have to specify a CEM GROUP for representing groups of people. In its export section an abstract data type Group is stated, including the constant `new_group` for constructing objects of type Group as well as operations on Group (e.g., inserting members). While GROUP's export section describes the service the CEM offers to its environment, GROUP's body section encapsulates the realization of the abstract data type Group and declares it as

the module `secret`. Assuming the ADT `Group` is constructed as a list of group members, the CEM `GROUP` needs an ADT for lists as well as an ADT for group members in order to realize the exported ADT `Group`. In `GROUP`'s import section, requirements to an ADT `List` for lists of members are described (e.g., the constants `new_list` for creating empty lists and operations on `List`). In `GROUP`'s common parameters section the ADT `Member` for representing group members is stated (e.g., the constant `new_member` for creating new group members and operations on `Member` for changing member properties). The ADT `Member` has to be specified in `GROUP`'s common parameters section, because potential users of `GROUP` must also have access to `Member` (e.g., for inserting members into a group). Note that the CEM `LIST` in isolation represents generic lists; it is parameterized by the ADT `Element` for list elements in `LIST`'s common parameters section. Only within the use relation between `LIST` and `Member`, the ADT `Element` is actualized with the ADT `Member` and the CEM `LIST` finally exports lists of members. Figure 3 shows three exemplary CEM incarnations for groups, members and lists.

A distributed software system can now be described as a configuration of distributed components which communicate via local or remote use relations (cf. [11]). For each remote use relation between two distributed components, a communication protocol and functional as well as non-functional attributes for this protocol can be specified. Further, non-functional requirements regarding remote use relations and the performance of potential server components can be stated with a client component. The remote use relation is only valid for that component, if its performance attributes satisfy the component's performance requirements.

In our CSCW example presented in figure 3, let us assume that the components `GROUP` and `LIST` are located on the same host (because `LIST` represents the internal realization of `GROUP`) while `GROUP` and `MEMBER` are distributed. Thus `GROUP` and `LIST` are connected via a local use relation while `GROUP` and `MEMBER` as well as `LIST` and `MEMBER` are connected via remote use relations.

So far we have shown how an architecture of distributed components can be described with the Π language. Now we will sketch how this design can be transformed into a distributed implementation compliant to OMG's CORBA standard (cf. [12], [13]). We have chosen CORBA for the implementation of the design, because component models in Π and CORBA resemble closely and can easily be integrated and CORBA represents a standard for the interaction of distributed objects. The standard allows to deal with openness and evolving systems.

Within CORBA the systems which actually perform service requests are called *object implementations*. Client objects and object implementations are isolated components, they communicate via interface descriptions. Interfaces are defined using the *Interface Definition Language* IDL. IDL is a definition language, it is independent from the actual programming language, its platform and the location of the objects as well.

Now we discuss the CORBA representation of a Π CEM at a very abstract level. The public abstract data types specified in the export section of the CEM are described in an *export IDL-module*. The *object implementations* for this export IDL-module represent the body section of the CEM. Further, an *import IDL-module* exists for stating the formal import of the CEM as CORBA provides no means for structured descriptions of imported services. If the CEM is described in its context, i.e. it is an incarnation connected with other incarnations via actual use relations, the mapping from formal to actual import is done within the implementation objects for the import IDL-module (this is the CORBA counterpart of a use relation). Our concept for Π /CORBA integration provides no explicit representation for the common parameters section in CORBA;

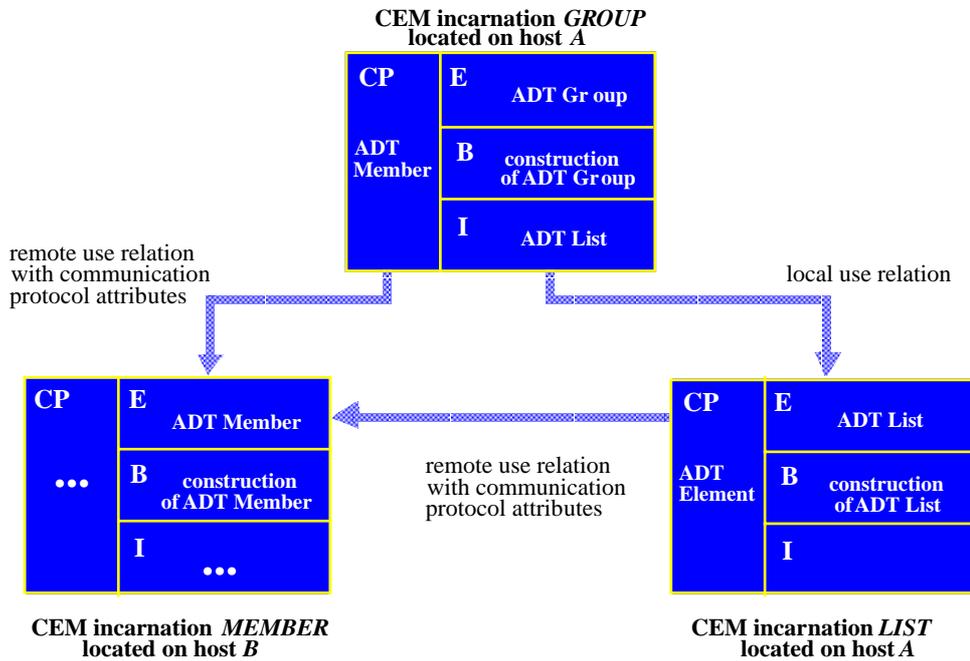


Figure 3: Distributed software architecture in II.

common parameter ADTs are described implicitly by specifying them both in the export and the import IDL-module. Details regarding our II/CORBA integration concept can be found in [11].

Although the architecture design of distributed components can be transformed to an implementation of distributed objects, no performance-related information of a component's interaction view is exploited. Therefore, in the next chapter we will present our concept for evaluating software architectures with emphasis on performance-related system properties.

3 Performance Evaluation of the Distributed Architecture Design

In addition to functional requirements, also non-functional requirements (e.g., response time, throughput, etc.) have essential impact on the design of distributed systems. This is true a priori, i.e. the analysis and assessment of a components' performance should be possible while the entire design architecture is still unfinished, as well as a posteriori, i.e. measuring the efficiency of the components' implementations.

Using the II 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 the Queuing Specification and Description Language QSDL ([2], [4]). 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 that has been developed at

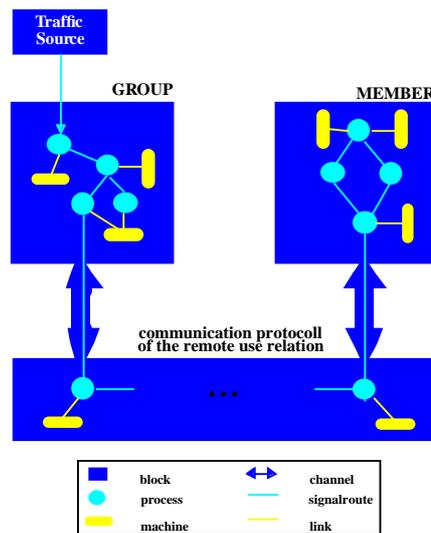


Figure 4: QSDL-systems for the CSCW example.

the University of Essen. By executing the simulator, stochastic performance measures can be gained.

We have identified interfaces between the component model in Π and the system specification in QSDL using ViewPoints a method engineering and integration framework. Thus performance requirements of a software system identified in its component model can be evaluated in its corresponding QSDL-system. Finally, the simulation results can be transferred back to the Π world also by means of the ViewPoint framework. Details according to the ViewPoint framework can be found in [5], while our Π /QSDL integration concept is described in [4]. In this paper, we only want to sketch the QSDL concepts at a very abstract level and show the QSDL representation of our CSCW example.

The starting point is the formal description of a system's behaviour in SDL, the Specification and Description Language of the ITU ([2]). Such a specification is usually called an SDL-system. An SDL-system consists of subsystems - called *blocks* - which contain communicating *processes*. These processes are described using extended finite state machines. An SDL-system may serve as basis for functional validation and simulation. The aspect of performance evaluation, however, is the objective of QSDL (Queuing SDL, cf. [4]). The QSDL approach is based on the adjunction of time consuming machines that model the congestion of processes for limited resources. *Machines* are building blocks providing a waiting room, a number of servers, a scheduling strategy, and in particular a set of services. Each service provided by a machine has a service specific speed value. The processes may request the services provided by the machines. Further, by adding *workload models* and a mapping of workload to machines a performance model can be generated.

Returning to our CSCW example, the component incarnations GROUP and MEMBER are distributed while the component incarnation LIST is local to GROUP. Now we will investigate the QSDL representation of GROUP, MEMBER, and the remote use relation between them (cf. figure 4).

Globally, a Π component incarnation can be represented by a QSDL block. The component's implementation in the body section can be described by the internal processes of the QSDL block. The component's interfaces can be represented by signalroutes entering and leaving the block. Communication protocols used by remote component connections have to be modelled as separate QSDL blocks. Finally, Π data types are represented by QSDL data types which are used as signal parameters and Π operations are described by QSDL procedures. Remote operation calls in Π are realized by signal transfers of the corresponding QSDL-procedures where the input parameters of the operation call are mapped on the signal parameters of QSDL. The output parameter of an operation call has to be modelled by a returning signal from the remote procedure to the calling unit; this returning signal has as its signal parameter the output parameter of the operation call.

Within a QSDL specification, measurement of performance-related system properties is done with the help of the *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 or the state distribution of a process). QSDL provides a standard sensor library for the most usual performance attributes and also allows individual user-defined sensors. Finally, the evaluation results can be visualized using the QUEST tool.

In Π , the concept for a remote use relation's performance attributes is adapted to QSDL's sensor concept and the interaction view's performance requirements are sensors extended by a compare operator and a concrete value (e.g., response time ≤ 10 ms). Thus bidirectional relations of performance-related system properties between Π and QSDL 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 in the development cycle's requirements stage.

4 Conclusions

In this paper we sketched how distributed component-oriented software architectures can be designed and evaluated based on a concept of independently created and interconnected software components. We covered some important aspects of the design stage of the development cycle with the architecture specification language Π and used the OMG standard CORBA for the implementation stage. We closed the gap between design and implementation by providing an integration concept for Π and CORBA.

We also discussed how performance-related requirements regarding distributed communication can be integrated into the Π design model. Such a design model may be analysed quantitatively in order to gain information about the distributed system's functional and non-functional behaviour. This information may either justify design decisions or may lead to changes in the design architecture.

However, no direct relations between the CORBA implementation and the QSDL performance model exist. Always the Π design architecture lies at the center of the design information flow. Thus the CORBA implementation of a design may help to identify sensible measures for performance-related system properties which then can be described in a software component's interaction view and finally evaluated within the QSDL performance model. Also, the results of a performance evaluation in QSDL may not only lead to changes in the design model but also in the CORBA implementation (e.g., a more efficient ORB realization).

5 Further Work

While tool support is realized for implementing local Π specifications in the target language C, we have almost completed the automated transformation from distributed Π specifications to OMG/IDL and the target language C++ for object implementations including runtime support. We use Sun SparcStations with Solaris and SunSoft's CORBA realization NEO. While tool support exist for both the Π -language and QSDL separately, we are also researching on implementing our Π /QSDL integration concept.

For developing large dynamic evolving systems it is also important to overcome the closed world assumption not only at design stage, but also within the distributed CORBA implementation and the QSDL performance model. Currently we are researching how to use dynamic invocation within the architecture's CORBA implementation in order to access newly added objects at run-time. While the evaluation of a QSDL-system always requires an environment (at least an abstract description of a load generator), QSDL-processes can be created dynamically.

6 Acknowledgments

The Π -language was developed while the first author was with H. Weber, now head of the Fraunhofer Institute ISST Berlin/Dortmund. His inspiration is gratefully acknowledged.

Bibliography

- [1] Bosch,J., Molin,P., Mattsson,M., and Bengtsson,P.O. (1997): Object-Oriented Frameworks - Problems & Experiences, submitted.
- [2] CCITT (1994): CCITT Z.100 Specification and Description Language (SDL), ITU-T.
- [3] Crane,J.S., and Dulay,N. (1997): A Configurable Protocol Architecture for CORBA environments, Proc. of ISADS 97, Berlin, Germany.
- [4] Diefenbruch,M., and Meyer,T. (1996): On Formal Modelling and Verifying Performance Requirements of Distributed Systems, Proc. IDPT 96, ISSN 1090-9389, Austin, USA.
- [5] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M. (1992): Viewpoints: A Framework for Integrating Multiple Perspectives in System Development, International Journal of Software Engineering and Knowledge Engineering, vol. 2, pp. 31-57.
- [6] Gamma,E., Helm,R., Johnson,R., and Vlissides,J. (1995): Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading,Massachusetts.
- [7] Ghezzi,C., Jazajeri,M., and Mandrioli,D. (1991): Fundamentals of Software Engineering; Englewood Cliffs, New Jersey: Prentice Hall.
- [8] Goedicke,M., and Nuseibeh,B. (1996): The Process Road between Requirements and Design, Proc. IDPT 96, ISSN 1090-9389, Austin, USA.

- [9] Goedicke, M., Cramer, J., Fey, W., and Große-Rhode, M. (1991): Towards a Formally Based Component Description Language - a Foundation for Reuse, Structured Programming Vol. 12 No. 2, Berlin: Springer.
- [10] Goedicke, M., and Schumann, H. (1994): Component-Oriented Software Development with Π , ISST report 21/94, Fraunhofer Institute for Software-Engineering and Systems Engineering.
- [11] Goedicke, M. and Meyer, T. (1997): A concept for the interaction of components, report Π -2 DFG project QUAPOS, University of Essen.
- [12] Mowbray, T.J., and Malveau, C. (1997): CORBA Design Patterns, Wiley Computer Publishing, New York: John Wiley & Sons, Inc.
- [13] Object Management Group, Inc. (1995): The Common Object Request Broker Architecture and Specification, revision 2.0.
- [14] Shaw, M., and Garlan, D. (1996): Software Architecture - Perspectives on an Emerging Discipline, Prentice Hall.

Reuse Contracts as Component Interface Descriptions

Koen De Hondt, Carine Lucas and Patrick Steyaert

Programming Technology Lab, Computer Science Department
Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium
{kdehondt,clucas,prsteyae}@vub.ac.be
www: <http://progwww.vub.ac.be>

Current interface descriptions are poor in describing components, because they only provide an external view on a component and they do not lay down how components interact with each other. Suggestions to improve component interface descriptions at last year's workshop are reconsidered and reuse contracts are put forward as a solution that goes one step further.

1 Introduction

One of the major issues at last year's Workshop on Component-Oriented Programming was the need for more information about how a component relies on its context, than traditionally provided by the current state of the art interface description languages.

Murer, Scherer and Würtz [4] stated that useful interoperability information is commonly published in additional documentation, since current interface description languages are not equipped with this capacity. Because such (informal) documentation is insufficient to manage interoperability and version control on a reasonable level, they introduce 3 levels of interoperability information. The Interface level provides an IDL-like interface of the component and addresses how components fit together structurally. The Originator level enhances the interface level information with information about which types and versions of components can work together. The Semantic level provides a complete description of a component's functionality. Murer, Scherer and Würtz argue that the second layer is necessary because it can be used to unveil interoperability issues which are investigated at development time, but rarely retained afterwards. These issues cannot be dealt with by the third level because there does not exist a technique for the complete description of semantic interoperability of components.

Ólafsson and Bryan [5] argued that, apart from the provided interface, a component interface description should also state the "required interfaces". A required interface is the interface of an acquaintance component that is required to enable a component to interact with that acquaintance component.

Although Ólafsson and Bryan argue that required interfaces are essential to understand the architecture of a component-based system, we claim that they in fact contain too little information to get a good understanding of the architecture, since an interface does not say what actually happens when one of its methods is invoked. For instance, an interface does not state the call-backs to the originating component. In our opinion, what is crucial in order to get a good understanding, is a description of the interaction structure, or the software contracts in which components participate. For this reason required interfaces are also insufficient to support component composition correctly, for

they allow the composition of components that have compatible provided and required interfaces, but not the correct interaction behavior.

While Murer, Scherer and Würtz store the information on interaction structure in a separate layer, we believe that it should be part of the interface of a component, so that it can be used to make the architecture clear, to help developers in adapting components to particular needs, and to verify component composition based on their interface instead of auxiliary (and perhaps informal) documentation.

At last year's workshop on Composability Issues in Object Orientation (CIOO '96), Lucas et al. argued that clear composition interfaces are needed to be able to compose components [3]. These composition interfaces should provide all the necessary information for the composition, while hiding the unimportant implementation details. Lucas et al. introduced reuse contracts (for inheritance) [6] as such composition interfaces.

In this paper, reuse contracts are applied to the domain of components. It will be shown that reuse contracts are not interface descriptions to which components have to comply exactly. Instead they can be adapted by means of reuse operators. These reuse operators state how a reuse contract is adapted. By comparing reuse operators applied to a reuse contract, conflict detection can be performed and composability of components can be validated. This capacity makes reuse contracts more than just enhanced interface descriptions.

2 Reuse Contracts

Essentially, a reuse contract is an interface description for a set of collaborating participant components. It states the participants that play a role in the reuse contract, their interfaces, their acquaintance relations, and the interaction structure between acquaintances. Reuse contracts employ an extended form of Lamping's specialisation clauses [1] to document the interaction structure. While Lamping's specialisation clauses only document the self sends of an operation, specialisation clauses in reuse contracts document all inter-operation dependencies. In their most basic form, specialisation clauses in reuse contracts just list the operation signatures, without type information or semantic information, such as the order in which operations are invoked.

Formally, a reuse contract is defined as a set of participant descriptions, where each participant description consists of a unique name, an acquaintance clause (the set of the participant's acquaintances), and an interface description. An interface description is a set of operation signatures consisting of a unique name (within the interface description) and a specialisation clause. A specialisation clause is a set of unique acquaintance names each with a set of operation names attached to them.

Since such formal specifications are hard to read, a visual representation of reuse contracts was developed. A participant is depicted by a rectangle containing the participant's name and interface. An acquaintance relationship is depicted by a line connecting two participants. Invoked operations, together with the operations that invoke them, are notated along this line. For clarity, the line can also be annotated with the name of the acquaintance relationship. As a shortcut, self-invocations are notated in the interface of a component, instead of along an acquaintance relation with itself.

Figure 1 shows a reuse contract for navigation in a web browser. The `handleClick` operation on the `WebBrowser` component invokes the `mouseClick` operation on the `WebDocument` component. The `WebDocument` component invokes its `resolveLink` operation when the mouse was clicked on a link (the details of the detection of the link is of no importance here). The

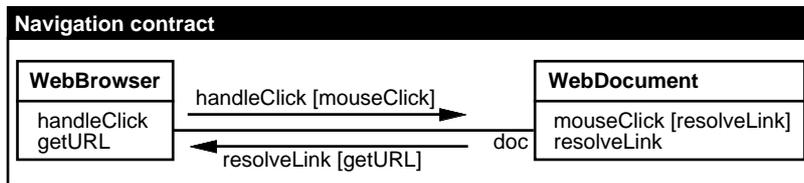


Figure 1: Example Reuse Contract

`resolveLink` operation invokes the `getURL` operation on the `WebBrowser` component in order to get the contents of the web page pointed to by the link. For simplicity, no arguments of operations are shown here.

A reuse contract documents the assumptions each participant makes about its acquaintances. For instance, in figure 1 the `WebBrowser` can safely assume that the `WebDocument` may invoke `getURL` when it invokes `mouseClick`. When a component developer builds a component, he can rely on these assumptions to implement the component according to the participant descriptions. However, requesting that a component is fully compliant with the interface and interaction structure descriptions, would make reuse contracts too constraining, and consequently too impractical to use. Instead, components may deviate from the reuse contract, but the component developer has to document how they deviate exactly, so that this information can be used later on to perform conflict checking.

Therefore, reuse contracts are subject to so-called reuse operators, actions that adapt participants and the interaction structure between these participants. In practice, a developer performs several adaptations at once in order to reuse a component. A few basic reuse operators were identified into which such adaptations can be decomposed [2]. More general adaptations are aggregations of the basic reuse operators. Each reuse operator has an associated applicability rule, that is, a reuse operator can only be applied when certain conditions apply. Applying a reuse operator on a reuse contract results in a new reuse contract, called the derived reuse contract.

Typical reuse operators on reuse contracts are extension and refinement, and their inverse operations, cancellation and coarsening. These operators come in two flavors: one flavor handles the operations on a participant, while the other flavor handles the operation on the context of a reuse contract, being the set of participants and their acquaintance relationships. A participant extension adds new operation descriptions to one or more participants in a reuse contract. The newly added operations do not refer to operations already present in the participants. A context extension adds new participant descriptions to a reuse contract. The acquaintance clauses of the new participants only contain names of participants that are added by the same extension. A participant refinement adds extra operation invocations to the specialisation clauses of already existing operations. A context refinement adds extra acquaintance relationships to a reuse contract.

The top of figure 2 shows how a web browser component with a history to store the already viewed URLs changes the original reuse contract given in figure 1. This new reuse contract is the result of applying the following reuse operators to the original reuse contract: a participant extension to add the operation `addURLtoHistory` to the interface of the browser component and a participant refinement to add the operation `addURLtoHistory` to the specialisation clause of `getURL`. Note that the browser

component's name has changed to `HistoryWebBrowser`. This is achieved through a renaming operation.

The bottom of figure 2 shows another adaptation of the original reuse contract. The rationale behind this adaptation is that the a new document component, called `PDFViewerPluginDocument`, only contains links that point to places within the PDF document and the targets of these links can thus be retrieved by the component itself. This retrieval is achieved with a new operation `gotoPage` instead of delegating this responsibility to the browser component through the operation `getURL`. Therefore the original navigation reuse contract is adapted as follows: a participant coarsening removes the invocation of `getURL` from the specialisation clause of `resolveLink`, a participant extension adds the new operation `gotoPage` to the interface of `PDFViewerPluginDocument`, and a participant refinement adds the invocation of `gotoPage` to the specialisation clause of `resolveLink`. A renaming operation is also required to change the name of the document component.

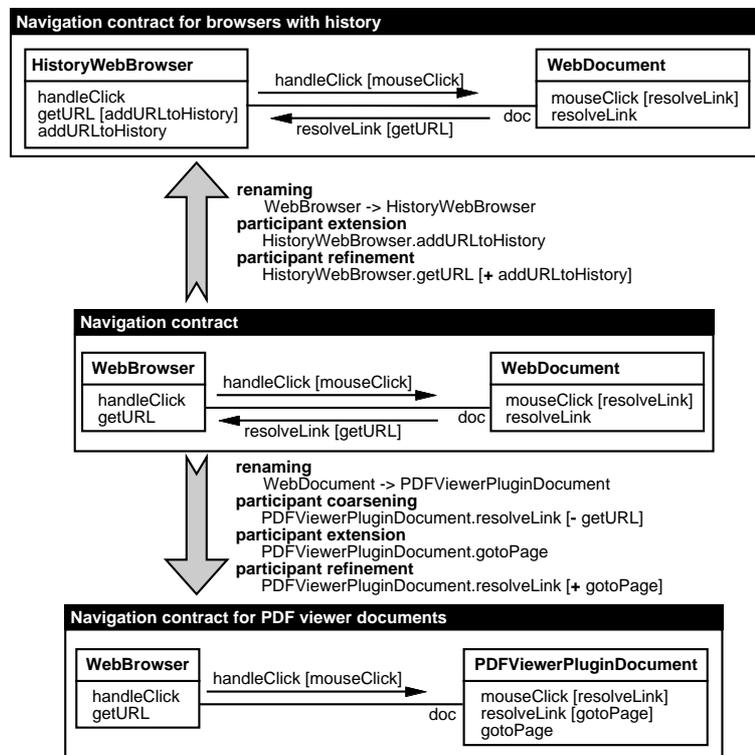


Figure 2: Two adaptations of the original reuse contract

3 Component Composition

When a reuser now wants to combine the `HistoryWebBrowser` with the `PDFViewerPluginDocument`, he runs into trouble, because his application will not behave correctly. Since link resolving is done by the `PDFViewerPluginDocument` instead of by the `HistoryWebBrowser`, the

HistoryWebBrowser's history will not be updated when the user clicks on a link in a PDFViewerPluginDocument.

With standard interface definitions, this problem would not have been detected until the application was running, because HistoryWebBrowser and PDFViewerPluginDocument have compatible provided and required interfaces.

With reuse contracts however, this problem is detected when the two components are composed. By comparing the reuse operators that were used to derive the two reuse contracts in figure 2, one can easily determine what inhibits composition of HistoryWebBrowser and PDFViewerPluginDocument. The top reuse contract is derived by applying a combination of an extension and a refinement on the original reuse contract. The extension adds the operation `addURLtoHistory` to the interface of the browser component, while the refinement adds an invocation of this operation to the specialisation clause of the operation `getURL`. The bottom reuse contract is a coarsening of the original reuse contract: the invocation of `getURL` was removed from the specialisation clause of the document component. Based on this comparison we can conclude that `getURL` and `addURLtoHistory` have become *inconsistent operations*[2][6]: HistoryWebBrowser assumes that `getURL` will be invoked, so that the history can be updated (through `addURLtoHistory`), while this assumption is broken by PDFViewerPluginDocument.

This example illustrates but one of many problems that may inhibit component composition. As the example shows, composition problems are in fact evolution problems. Reuse contracts are derived from an original reuse contract through an evolution step which is recorded in a reuse operator.

Consider the diagram in figure 3. We call the reuse contract corresponding to the original ensemble of components the *base (reuse) contract*, the reuse contract corresponding to the first modification the *derived (reuse) contract* and the reuse contract corresponding to the second modification the *exchanged base contract*.

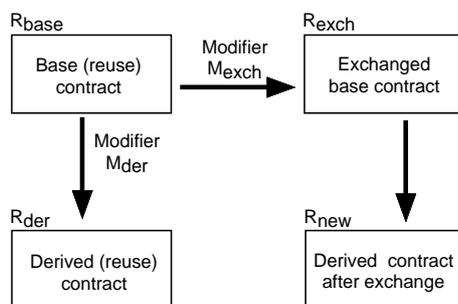


Figure 3: Base Reuse Contract Exchange

By examining the effect of applying modifier M_{der} to the exchanged base contract, a table of possible conflicts can be derived. For some conflicts, checking the applicability rules of the reuse operators is enough to detect conflicts, while for other conflicts extra information about M_{der} and M_{exch} is required. A complete list of conflicts can be found elsewhere [2]. The problem of inconsistent operations in the example is but one of many.

4 Ongoing Work

Although we believe that reuse contracts can greatly enhance the current component interface description languages, we have not yet integrated reuse contracts in an environment supporting such specifications. Currently, we employ reuse contracts mainly in a Smalltalk development environment. The reuse contracts are stored in a repository separate from the Smalltalk class repository. We have conducted an experiment to incorporate reuse contracts in Java interfaces. For now, this experiment restricts itself to reuse contracts for inheritance [6], however. The Java compiler was extended to support the specification and the checking of reuse contract related information, such as specialisation clauses and reuse operators.

5 Conclusion

In this paper we have presented reuse contracts as enhanced component interface descriptions. Since we believe that the interaction structure between a component and its acquaintances is crucial to get a good understanding of the component architecture, and to ensure correct composition, reuse contracts not only provide the interface of a component, but they also document what interface a component requires from its acquaintances and what interaction structure is required for correct inter-component behavior.

Component evolution is an integral part of the reuse contract approach. Reuse operators define relations between reuse contracts and their derivations. When reuse contracts are evolved in parallel, the applied reuse operators can be compared to perform conflict detection. When conflicts occur, this indicates that some components cannot be composed.

Bibliography

- [1] John Lamping. Typing the specialization interface. In *Proceedings of OOPSLA'93 (Sep. 26 - Oct. 1, Washington, DC, USA)*, volume 28(10) of *ACM Sigplan Notices*, pages 201–214. ACM Press, October 1993.
- [2] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Vrije Universiteit Brussel, 1997.
- [3] Carine Lucas, Patrick Steyaert, and Kim Mens. Research topics in composability. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP'96, Linz*, pages 81–86. dpunkt Verlag, 1997.
- [4] Tobias Murer, Daniel Scherer, and Andy Würtz. Improving component interoperability. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP'96, Linz*, pages 150–158. dpunkt Verlag, 1997.
- [5] Asgeir Ólafsson and Bryan Doug. On the need for "required interfaces" of components. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP'96, Linz*, pages 159–165. dpunkt Verlag, 1997.

- [6] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA'96 (Oct. 6-10, San Jose, California)*, volume 31(10) of *ACM Sigplan Notices*, pages 268–285. ACM Press, 1996. Also available at <http://progwww.vub.ac.be/prog/pools/rcs/>.

A Control Model for the Dynamic Selection and Configuration of Software Components

Philippe Lalanda

Thomson-CSF Corporate Research Laboratory
Domaine de Corbeville, F-91404 Orsay, France
lalanda@thomson-lcr.fr

Component-oriented programming, which amortizes development cost on several systems, is becoming increasingly important in the software-intensive industry that is facing today both economical and technical challenges. However, there is currently little guidance for software developers on how to compose software components in order to produce running applications. We believe that the development of domain-specific software architectures (DSSA) provides a way to integrate properly software components developed by different organizations. A DSSA takes into account the domain of applications under consideration and provides the computational framework necessary to solve typical problems of the domain. The purpose of this paper is to present an architectural approach that permits the development and exploitation of DSSAs. This approach builds on a model of dynamic control that permits to select and configure software components both statically and dynamically.

1 Introduction

The industry of software-intensive systems is facing today both economical and technical challenges. On one hand, shrinking budgets and sharp competition require to reduce significantly development and maintenance costs, shorten time-to-market, and improve predictability in terms of cost and development time. On the other hand, the size and complexity of systems have dramatically increased in the past few years. Software control is replacing electronic control as the most flexible means to meet market needs, and is enhancing noticeably systems capabilities. Studies show that software size has thus been multiplied in average by 10 in the last five years in many companies specialized in software-intensive systems [1]. This has brought considerable problems in terms of suitability, efficiency, scalability and portability. Component-oriented pro-

gramming, which amortizes development cost on several systems, is therefore becoming increasingly important. The key paradigm of this approach is megaprogramming [2], that is the ability to define a system by putting together software components. This approach enables industrial companies to evolve their new and existing systems into software product families. These are groups of similar products addressing a same business domain and sharing common assets, including requirements, design, source code and test cases. This approach is tremendously reuse promoting and permits effective capitalization on a given domain. However, component-oriented programming raises

several important issues. In particular, unguided composition of software components is unlikely to produce applications able to meet pre-determined requirements. A software application is not a simple collection of specialized components, but a coherent organization built to tackle a given domain. We therefore believe that the definition of domain-specific software architectures (DSSA) permits more effective composition. A DSSA takes into account the domain under consideration and provides the computational framework necessary to solve typical problems of the domain. It describes the type of components that can be integrated in the system, their possible connections, and the rationale under their collaboration. The purpose of this paper is to present an

architectural approach that permits the development and exploitation of DSSAs. This approach builds on a model of dynamic control that permits to select and configure software components both statically and dynamically. We will show that it permits the integration of software components developed in separate projects, while ensuring overall architectural coherence. This approach is developed and experimented in various domains including real-time mission planning and updating in the avionics domain.

2 Domain-specific software architecture

2.1 Software architecture

Software architecture brings a design level which is not concerned with code and data structure but with the global structure of a system, its main constituents and the way they communicate, synchronize and share functionalities [3]. Its purpose is to organize the large-grained objects (or components) of a system, to explain their relationships and evolutions, and to bring solutions for their implementation. It is today widely acknowledged that many of the life-cycle concerns of software applications are actually tackled at the architecture level. An architecture provides an understandable support for discussions between the various stakeholders of the system under construction. It provides many other advantages[4]:

- it reduces risk, cost, and time-to-market,
- it increases predictability, reliability, quality,
- it provides early identification of potentially very large reuse opportunities, and
- it brings new possibilities for early analysis and validation that reduce risk and cost.

The software architecture field has received wide attention recently. Research is today conducted in different areas in order to permit easier building and validation of software architectures. This includes works on design patterns [5] [6], object-oriented frameworks, the definition of expressive notations for representing architectural designs, the use of formal techniques for early architectural validation, and the development of design methods.

2.2 Architectural models for DSSA

Domain-specific software architecture [7] is a subdomain of software architecture where a reference architecture, partly abstract, is developed in a well understood domain. A DSSA comprises:

- A reference architecture which describes a general computational framework. It is a starting point for building a family of applications that solve a particular problem domain.
- A software component library which contains reusable chunks of domain expertise.
- An application configuration method for selecting and configuring components within the architecture to meet particular application requirements.

The motivations for developing a DSSA are the same as those for using software architecture plus the desire to develop and reuse a pool of components (assets) pluggable in the architecture, and to generate new applications by selection/composition of components based on application requirements. A DSSA therefore provides a framework for top-down design, because all the components and interconnections are predetermined. It provides the foundations for components interoperability. Generic archi-

tectural models have been proposed recently to support the development of DSSAs. GenVoca [8] is a domain-independent model for defining scalable families of hierarchical systems as compositions of reusable components. A reference architecture in GenVoca is made of *realms*, that is sets of reusable components that export and import standardized components organized into semantics layers, and design rules to identify (and then preclude) illegal components combination. An application is obtained by combining subsystems, that is combinations of components of the same realms. This approach, which makes the assumption that a system can be expressed as a combination of primitive components (it is actually an equation), is limited to specific domains. Rapide [9] constitutes a more general approach. Rapide is a computer language for

defining and executing models of system architecture. An architecture is defined by a set of modules and their interconnections, with no restriction on their organization. All communication between modules is explicitly defined by connections between module interfaces. Interfaces specify both the operations a module provides and, in addition, the operations it requires from other modules. In this approach, components communicate directly. Selected components can introduce new requirements for capabilities other component implementations will need to satisfy. The integration of components is therefore not straightforward and depends on the current system's configuration. In

the next section, we present an alternative approach for the design and development of DSSAs. This approach is based on a dynamic control model where components are kept independent and where communication is made through a shared data structure.

3 Blackboard-based control

We have been developing and experimenting with a domain-independent control model that permits to:

- define DSSAs made of independent software components cooperating through shared knowledge bases,
- select and configure software components both statically and dynamically.

Our approach builds on a model of dynamic control [10] where a system has (a) a repertoire of independent domain and control components that are described in terms of their resource requirements and result properties; (b) a control plan expressing its desirable behavior; (c) a meta-controller that chooses at each point in time the currently enabled component, domain or control, that best matches the current control plan.

3.1 Components

Domain and control components are kept in a library. Domain components are concerned with the solving of a particular problem. They retrieve data from the knowledge base and write their contribution to the problem solving in it. Control components deal with the management of the system control plan. They can replace the current plan, postpone it, refine it, *etc.* Each component has a set of triggering conditions that can be satisfied by particular kinds of *events*, that is global changes in the system resulting from external inputs or previously executed components. Global changes occur in a shared knowledge base which is accessible by all the components. When an event satisfies a component's triggering conditions, the component is enabled and its parameters bound to variable values from the triggering situation. A given component will be enabled, and therefore executable, whenever events satisfying its triggering conditions occur, regardless of its relative utility in achieving the current goals. Conversely, at each point in time, many competing components will be enabled and the system must choose among them to control its own goal-directed behavior. To support this control decisions, each component has an interface that describes the kinds of events that enable it, the variables to be bound in its enabling context, the task it performs, the type of method it applies, its required resources (e.g, computation, perceptual data), its execution properties (e.g, speed, complexity, completeness), and its results properties.

3.2 Control plan

A control plan describes the system's intended behavior as a temporal graph of plan steps, each of which comprises a start condition, a stop condition, and an intended activity in the form of a tuple (task, parameters, constraints). Control plans do not refer explicitly to any particular component in the system's repertoire. They only describe intended behaviors in terms of the desired task, parameter values, and constraints. Thus, at each control cycle, the system has a plan of intended action, which intentionally describes an equivalence class of desirable behaviors and in which currently enabled specific components may have graded degrees of memberships.

3.3 Meta-controller

In our model, the meta-controller makes no difference between domain components and control components. Both are managed the same way. The meta-controller attempts to follow the current control plan by executing the most appropriate enabled components. Specifically, the meta-controller configures and executes the enabled component that: (a) is capable of performing the currently planned task with the specified parameterization; and (b) has a description that matches the specified constraints better than any other enabled component that also satisfy (a). If the selected enabled component is a control component, the control plan is updated. Otherwise, a domain component is executed in order to contribute to the problem solving process. This gen-

eric architectural model supports the development of a wide variety of DSSAs. The dynamic control model provides a framework in which appropriate sets of components can be selected and configured at both design time and run time. The integration of components is actually very simple since components are considered independently and are only characterized by their own properties. At run-time, if useful new application-relevant component should become available, the new components can be substituted for old ones or added to the knowledge base alongside the old ones, without interrupting system operations. The architecture's event-based enabling of components, its plan-based meta-control choices among competing components, and its effort to retrieve necessary knowledge from the shared knowledge base are not preprogrammed to require any particular components. They operate on whatever components are available in the component library at run-time. In the other hand, the approach does not guarantee the delivering of a solution to a given problem, since appropriate components might be missing.

4 Domain of Experiments

We have been developing DSSAs in several domains. In particular, our approach has been applied to autonomous office robots in two applications: office surveillance and office delivery. Detailed results about these experiments can be found in [11]. We are now conducting experiments on the planning and real-time updating of aircraft missions. We give in this section a brief description of the purposes and main features of such a system, and explain why it has been chosen as a viable domain for the building of a DSSA. The primary goal of a mission planning system is to provide the aircraft's

automatic pilot with a multi-dimension trajectory. A *nominal* trajectory is computed off-line before the mission starts. As the mission progresses, the goal of the mission planning system is to check the consistency between the expected plane position as given by the nominal trajectory and its actual position. If some deviation is detected, the system has to analyze the discrepancy and generate a new trajectory fulfilling the mission goals as well as possible. Mission planning systems involve two important sets

of software components:

- Computing modules calculating various dimensions of a trajectory,
- Plans expressing various ways to react to unexpected events.

Numerous components are available. This is due to the multiple ways to compute a trajectory. A trajectory can be formed in order to meet different requirements like speed, fuel preservation, or discretion. It can also be expressed along different dimensions. Also, calculating algorithms vary in function of the target plane (or helicopter). Similarly, many different reaction plans are available depending on the pilot's profile, mission objectives, planes characteristics, *etc...*

This domain is ideally suited for the development of a DSSA for several reasons :

- The domain is well understood and the basic avionics technology and operating environment are relatively stable,
- There are several existing systems and a need for many new applications,

- Common abstractions and features can be identified across different existing systems,
- Domain-specific components are available and variability can be expressed at the architectural level with a variation in components.

Strategies necessary to react to unexpected components have been encapsulated in control components which transform them into control plans. Algorithms specialized in trajectory calculations have been wrapped up in domain-specific components. According to the principles previously presented, the architecture supports both design time and run-time configuration. At design time, one can select the strategies and calculating algorithms required to tackle an application, characterized by a specific plane and specific mission objectives. At run-time, new components can be integrated in order to provide new reactions to a given situation or new ways to compute a trajectory, or to replace components.

5 Conclusion

Reusable software components have been a persistent, but elusive goal of the software engineering community. We believe that this is partly due to a lack of architectural perspectives. Reuse has to be prepared at the very beginning of software development when setting the architectural foundations of a system. The general DSSA approach

aims to factor large classes of applications into reusable reference architectures and components. It provides a coherent computational framework where software components can be plugged with confidence. Our generic architectural model permits the design and implementation of DSSAs in many domains. It makes use of the basic blackboard organization in order to enable the integration and interoperation of diverse components. The additional features of the dynamic control model provide the necessary additional support for flexible run-time configuration and meta-control. This approach has been successfully applied to diverse domains like autonomous robots and monitoring systems and is now used for the planning and real-time updating of aircraft missions. It has allowed us to integrate smoothly legacy code with newly developed components and to select and configure them for specific applications.

Bibliography

- [1] ROADS. Esprit Project. Information Technologies RTD Programme, Domain 1 : Software Technologies, 1996.
- [2] B.W. Boehm and W.L. Scherlis, *Megaprogramming*, Proceedings of the DARPA Software Technology Conference, April 1992.
- [3] D.E. Perry and A.L. Wolf, '*Foundations for the study of software architecture*', ACM SIGSOFT Software Engineering Notes, vol. 17, no 4, 1992.
- [4] D. Garlan and M. Shaw, *An introduction to software architecture*, Advances in Software Engineering and Knowledge Engineering. New York: World Scientific, Vol. I, 1993.

- [5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, *Pattern-oriented Software Architecture: A System of Pattern*, Wiley & Sons.
- [7] E. Mettala, *Presentation at ISTO Software Technology Community Meeting*, June, 1990.
- [8] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, *The Gen-Voca Model of Software-System Generators*, IEEE Software, September 1994.
- [9] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann, *Specification and Analysis of System Architecture Using Rapide*, IEEE Transactions on Software Engineering, vol. 21, number 4, April 1995.
- [10] B. Hayes-Roth, *A blackboard architecture for control*, Artificial Intelligence, num. 26, 1985.
- [11] B. Hayes-Roth, P. Lalanda, P. Morignot, M. Balanovic and K. Pflieger, *A domain-specific software architecture for adaptative intelligent system*, IEEE Transactions on Software Engineering, 1995.

The Fragile Base Class Problem and Its Impact on Component Systems

Leonid Mikhajlov

Turku Centre for Computer Science
Lemminkäisenkatu 14A, Turku 20520, Finland
lmikhajl@aton.abo.fi

Emil Sekerinski

Dept. of Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland
esekerin@aton.abo.fi

In this paper we study applicability of the code inheritance mechanism to the domain of open component systems in light of so-called *fragile base class problem*. We present five requirements restricting code inheritance that are sufficient to circumvent the problem. We propose a system architecture based on *disciplined inheritance* and present three check lists for component framework designers, component framework developers, and its users.

1 Introduction

One of the most important characteristic features of open component systems is the late integration phase. End users obtain components from the software market and integrate them into their system. In general, parties developing the components are unaware of each other as well as of the end users that are going to integrate these components.

The component oriented paradigm stemmed from the main principles of object orientation. Such concepts as encapsulation and subtyping are intrinsic to component development. One of the key ideas in object-oriented development is the construction of new objects by incremental modification of existing ones. Apparently, the possibility of constructing new components by reusing previously designed ones is highly desirable. The primary reuse mechanism employed in object-oriented languages is (code) inheritance. Whether the inheritance mechanism can be used in component system development and whether it can extend over component boundaries is unclear and requires close consideration.

In this paper we consider the so-called *fragile base class problem* and its influence on application of inheritance over component boundaries. At first glance the problem might appear to be caused by inadequate system specification or user assumptions of undocumented features. We consider an example which demonstrates that this is not the case and that the problem can be very concealed.

We abstract the essence of the problem into a flexibility property and explain why unrestricted code inheritance violates this property. We present orthogonal examples suggesting five restrictions for disciplining inheritance. In [8], we have proved that these requirements are sufficient.

We propose a disciplined approach to inheritance which allows safe implementation reuse and provides high degree of flexibility. We present three check lists, for component framework designers, component framework developers and its extenders. By verifying that every requirement in the corresponding list holds, all parties can make sure that they successfully avoid the fragile base class problem.

2 The Fragile Base Class Problem

The fragile base class problem becomes apparent during maintenance of open object-oriented systems. Imagine that a customer has obtained a certain component framework consisting of a collection of classes. In this framework inheritance is employed as the primary implementation reuse mechanism. The customer willing to make slight modifications to the functionality provided by a framework class inherits from it and overrides several methods. So far everything works fine and objects generated from the resulting class are perfectly substitutable for original ones generated from the framework class. When framework developers release a new version of their system, naturally they claim that the new version of the system is fully compatible with the previous one. Unfortunately, soon after obtaining the new version of the framework, the customer discovers that some custom extensions are invalidated. The following example, adopted from [15], illustrates the presented scenario. In this example a class *Bag* belongs to a framework, *CountingBag* is its custom extension, and *Bag'* is a revision of the *Bag*.

```

Bag = class
  b : bag of char

  init ≐ b := []
  add(val x : char) ≐
    b := b ∪ [x]
  addAll(val bs : bag of char) ≐
    begin var y | y ∈ bs.
      while bs ≠ [] do
        self.add(y);
        bs := bs - [y]
      od
    end
  cardinality(res r : int) ≐
    r := |b|
end

CountingBag = class
  inherits Bag
  n : int
  init ≐ n := 0; super.init
  add(val x : char) ≐
    n := n + 1; super.add(x)
  cardinality(res r : int) ≐
    r := n
end

Bag' = class
  b : bag of char
  init ≐ b := []
  add(val x : char) ≐ b := b ∪ [x]
  addAll(val bs : bag of char) ≐ b := b ∪ bs
  cardinality(res r : int) ≐ r := |b|
end

```

It is easy to notice that if *Bag'* is used as the base class for *CountingBag*, the resulting class returns the incorrect number of elements in the bag.

Apparently, inheritance is responsible for the problem. Different kinds of problems connected to inheritance have been widely discussed in the literature [13, 17, 5]. The

source of these problems can be traced back to the fact that inheritance violates encapsulation [13]. In a closed system (at least in theory) encapsulation can be compromised in order to achieve the desired degree of flexibility. Correctness of the resulting system can be verified on the integration phase. Since in open systems it is impossible to conduct a global integrity check, it becomes impossible to guarantee the overall correctness of the system. Therefore, the fragile base class problem is of particular importance for open component systems. If this problem were not considered at the design stage, it is too late to try to amend it during exploitation.

The problem of safe modification of base classes in presence of independent extensions deserves a separate name. We have encountered the name fragile base class problem in the technical literature describing component standards [19, 4]. Although we noticed slight deviations from our understanding of the problem, we think that this name expresses the essence of the problem rather well.

3 Failure of the Ad-Hoc Inheritance Architecture

Let us analyze the reasons for failure in modifying a system relying on ad-hoc code inheritance. Assume that we have a base class C and an extension class D inheriting from it. We say that D is equivalent to $(M \mathbf{mod} C)$ ¹, where M corresponds to the extending part of the definition of D and the operator \mathbf{mod} combines M with the inherited part C . We refer to such M as a *modifier* [18]. The model of single inheritance employing the notion of modifiers was proved by Cook and Palsberg in [2] to correspond to the form of inheritance used in object-oriented systems.² For example, in our previous example M has the form:

```

M = modifier
  n : int
  init  $\hat{=}$  n := 0; super.init
  add(x : char)  $\hat{=}$  n := n + 1; super.add(x)
end

```

Therefore, we have that C belongs to the component framework, while $(M \mathbf{mod} C)$ represents a custom extension of this system. When system developers state that the new version of their system is fully compatible with the previous one, they essentially say that C' is a *refinement* of C . We say that some class C is *refined by* another class C' if the externally observable behavior of objects generated by C' is the externally observable behavior of objects generated by C or its improvement. In other words, objects generated by C' must be substitutable for objects generated by C in any possible context.³ Ensuring substitutability of the custom extension $(M \mathbf{mod} C)$ for the framework class C amounts to verifying that C is *refined by* $(M \mathbf{mod} C)$.

Under these two conditions all participating parties, i.e. framework developers and its extenders rely on the following property:

if C is refined by C' and C is refined by $(M \mathbf{mod} C)$
then C is refined by $(M \mathbf{mod} C')$

¹We read \mathbf{mod} as modifies.

²In their paper modifiers are referred to as wrappers. We prefer the term modifier, because the term wrapper is usually used in the context of object aggregation.

³Readers familiar with the notion of behavioral subtyping [7, 1] can think of class refinement as behavioral subtyping. We give a formal definition of class refinement in [8].

Further on we refer to it as the *flexibility property*. Unfortunately, this flexibility property does not hold in general, as demonstrated by our example. This consideration brings us to the question, how we can redesign the system architecture so that the flexibility requirement would hold.

4 Disciplined Inheritance

In this section we suggest a component framework architecture based on *disciplined inheritance* which relies on inheritance for implementation reuse. We represent every class in the framework by two, an interface definition class augmented with specifications of the intended functionality and its default implementation. We refer to the augmented interface class as a specification class. We assume that the default implementation remains completely hidden behind the specification class. Therefore, the user of the framework can rely only on the information provided by the specification class. When reusing the framework, the user derives an extension class from an appropriate framework class. However, what the user sees is just the specification class. This specification class is too abstract to be executed and at run time is substituted with its default implementation.

The formal study undertaken in [8] brought us to formulation of five sufficient requirements disciplining the inheritance mechanism which allow us to circumvent the fragile base class problem. We justify these requirements by means of orthogonal examples, each violating the flexibility property in a different manner. In these examples C is a base class, M is a modifier and C' is a revision of C . Due to space limitations we omit the proof of sufficiency of these requirements which is presented in [8].

Let us briefly introduce the used terminology. When an extension class invokes its base class method, we say that an *up-call* has occurred; when a base class invokes a method from a class derived from it, we refer to such an invocation as a *down-call*. A call of a method from another method in the same class is referred to as a *self-call*. We refer to an invocation of a base class method by an extension class as a *super-call*.

“No cycles” requirement: *An implementation class and a modifier should not jointly introduce new cyclic method dependencies.* Cyclic method dependencies can appear due to the presence of *self-recursion* in the implementation class and the modifier. A class (modifier) is said to be self-recursive if methods of this class (modifier) call other methods of the same class (modifier). Due to dynamic binding such method calls of the base class may be redirected to methods of the modifier. Since at run time the specification class is substituted with its default implementation, which might have a different structure of self-recursion, a cyclic method dependency may occur leading to non-termination. Consider the following example:

$C = \text{class}$	$M = \text{modifier}$	$C' = \text{class}$
$x := 0$		$x := 0$
$m \hat{=} x := x + 1$		$m \hat{=} \text{self}.n$
$n \hat{=} x := x + 1$	$n \hat{=} \text{self}.m$	$n \hat{=} x := x + 1$
end	end	end

“No implementation class self-calling assumptions” requirement: *Implementation class methods should not make any additional assumptions about the behavior of the*

other methods in itself. Only the behavior described in the specification class must be taken into consideration. If this requirement is violated, caller methods in the implementation class can assume properties inferred from implementations of the callee methods in the same class. Such assumptions may be broken if the modifier overrides these callee methods. This requirement can be justified by the following example:

```

C = class                                M = modifier
  m(val x : Real, res r : Real) ≐        m(val x : Real, res r : Real) ≐
    pre x ≥ 0                             r := -√x
    post r2 = x
  n(val x : Real, res r : Real) ≐
    pre x ≥ 0
    post r4 = x
end                                        end

```

```

C' = class
  m(val x : Real, res r : Real) ≐
    r := √x

  n(val x : Real, res r : Real) ≐
    self.m(x, r); self.m(r, r)
end

```

“No specification class down-calling assumptions” requirement: *Methods of a modifier should disregard the fact that specification class self-calls can get redirected to the modifier itself. In this case bodies of the corresponding methods in the specification class should be considered instead, as if there were no dynamic binding.* This requirement is similar to the previous one, except that now the modifier can make extra assumptions that can be violated by an implementation class. In the following example $\{p\}$ is an assertion statement, which skips if the predicate p is *true* and aborts otherwise.

```

C = class                                M = modifier  C' = class
  x := 0                                  x := 0
  l ≐ {x ≥ 0}; x := 5                      l ≐ x := 5
  m ≐ self.l                               m ≐ {x ≥ 0}; self.l
  n ≐ x := 5; self.m                       n ≐ x := 5; self.m
end                                        end          end

```

“Conditional access to specification class state” requirement: *If a specification class allows access to its instance variables to a modifier class, then its implementation class should not change the data representation of the specification class. Otherwise, the modifier may only access the state of its base class through calling base class methods.* If the instance variables of the specification class are directly accessible to its extensions, they are implicitly a part of the interface of the specification class [13]. In this case it becomes impossible for an implementation class to provide an alternative data representation. This requirement can be justified by the following example:

<pre> C = class x := 0 m $\hat{=}$ x := x + 1 n $\hat{=}$ x := x + 2 end </pre>	<pre> M = modifier n $\hat{=}$ x := x + 2 end </pre>	<pre> C' = class x := 0; y := 0 m $\hat{=}$ y := y + 1; x := y n $\hat{=}$ y := y + 2; x := y end </pre>
---	---	--

“No modifier invariant function” requirement: *A modifier should not bind values of its instance variables with values of the specification class instance variables to generate an invariant.* When creating a modifier, its developer usually intends it for a particular base class. A common practice is an introduction of new variables in the modifier and binding their values with the values of the base class instance variables. Such a binding can be achieved even without explicitly referring to the base class variables. We can say that such a modifier carries an *invariant function* which is applied during the modifier application and returns an invariant of the resulting class. The following example demonstrates that, in general, such an invariant function can lead to problems.

<pre> C = class x := 0 l(res r : int) $\hat{=}$ r := x m $\hat{=}$ x := x + 1; self.n begin var r. n $\hat{=}$ self.l(r) end end </pre>	<pre> M = modifier y := 0 m $\hat{=}$ y := y + 1; super.m begin var r. n $\hat{=}$ super.l(r); {r = y} end end </pre>
<pre> C' = class x := 0 l(res r : int) $\hat{=}$ r := x m $\hat{=}$ self.n; x := x + 1 begin var r. n $\hat{=}$ self.l(r) end end </pre>	

5 Application

From the above requirements we derive three check lists for component framework designers, component framework developers, and its extenders. By verifying that every item in a list holds, all parties can make sure that they successfully avoid the fragile base class problem. First we present the *check list for framework designers*:

1. The framework designers should decide whether they want to fix the data representation of the framework class. In some languages [16] this decision can be expressed by putting the declaration of the instance variables in a *private* or *protected* section of the class definition. Private attributes are only accessible by methods of the same class, while protected attributes can be accessed by the extension class as well. When instance variables of the specification class appear in the protected section, implementation class must have the same data representation. When instance variables of the specification class are declared in the private

section, implementation class can change the data representation. If future extensions are expected to require more freedom in modifying the class state than is allowed by the class client interface, the class can provide a number of low-level state modifying methods. Since having these methods as a part of the class client interface may be inappropriate, we suggest declaring them as protected [13].

2. Specification class method bodies must indicate all *self* method calls.

Now let us consider the *check list for framework implementors*:

1. The implementation class can change the data representation of the specification class only if it is declared as private.
2. A method of the implementation class may self-call only those methods that are self-called by its counterpart in the specification class.
3. When verifying that some method of the implementation is a refinement of a matching method in the specification, instead of considering the bodies of the self-called methods, one should consider the bodies of the corresponding methods in the specification.

And finally let us consider the following *check list for framework extenders*:

1. A method of an extension class may only self-call those methods that are self-called by its counterpart in the specification class, or it may make up-calls to these methods. Plus, an extension method may always make up-calls to its counterpart in the specification class.
2. When verifying that some method of the extension is a refinement of a matching method in the specification, one should disregard the fact that due to dynamic binding the base class can make down calls to the extension class methods. One should consider that the base class calls its own methods instead.
3. The extension class may not establish an invariant binding values of inherited instance variables with values of its own instance variables.

6 Related Work and Conclusions

We have discussed the fragile base class problem and its impact on component systems. The name fragile base class problem was introduced while discussing component standards [19, 4] since it has critical significance for component systems. Modification of the components by their developers should not affect component extensions of their users in any respect. Firstly recompilation of derived classes should be avoided if possible [4]. This issue constitutes a syntactic aspect of the problem. While being apparently important, that problem is only a technical issue. Even if recompilation is not necessary, component developers can make inconsistent modifications. Such inconsistent base class modifications constitute a semantic aspect of the problem, which is the focus of our study. This aspect of the problem was recognized by COM and Oberon/F developers [19, 11]. They see the root of the problem in inheritance violating data encapsulation and choose to abandon inheritance, by employing the forwarding architecture. Although solving the problem this approach comes at the cost of reduced flexibility.

We discuss how disciplined inheritance enables consistent modifications of base classes, although encapsulation is violated. To discipline inheritance, we formulate five requirements. In [8], we formalize the problem in an extension of refinement calculus [3, 9, 10] and prove that these requirements are sufficient. In fact only the “no direct access to base class state” requirement was known from the literature. The other requirements have emerged while attempting to formally prove the property ensuring safety of base class modification.

Consideration of these requirements allows us to formulate an architectural approach based on disciplined inheritance combining flexibility of an ordinary inheritance architecture with safety of a forwarding architecture. We have presented three check lists for framework designers, framework developers and framework users to be used as guidelines for constructing component systems employing inheritance and avoiding the fragile base class problem.

Other papers consider the question of ensuring the substitutability of the classes created by inheritance from some base classes for these base classes [1, 7, 14]. This problem constitutes only a part of the fragile base class problem and as such is one of the flexibility property premises.

The fragile base class problem in our formulation (although they do not refer to it by this name) is considered by Steyaert et al. in [15] and by Kiczales and Lamping in [6]. The first paper is most closely related to our work. The authors introduce *reuse contracts* “that record the protocol between managers and users of a reusable asset”. Acknowledging that “reuse contracts provide only syntactic information” they claim that “this is enough to firmly increase the likelihood of behaviorally correct exchange of parent classes”. In our opinion such syntactic reuse contracts are insufficient to guard against the fragile base class problem.

The objective of the paper by Kiczales and Lamping is to develop a methodology for informally specifying a framework, so that the specification would accurately describe its functionality and provide the framework user with appropriate leeway for extending and reusing it. However, their recommendations are based only on empirical expertise. We believe that such methodology should be grounded on a mathematical basis and developing such methodology constitutes the ultimate goal of our research.

Effects of disciplining inheritance the way we propose on component and object-oriented languages and systems require separate consideration and constitute the subject of our future research. The other research direction is in generalizing the results by weakening the restrictions we have imposed on the inheritance mechanism.

Bibliography

- [1] Pierre America. Designing an object-oriented programming language with behavioral subtyping. *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of LNCS, pp. 60-90, Springer-Verlag, New York, N.Y., 1991.
- [2] William Cook, Jens Palsberg. A Denotational Semantics of Inheritance and its Correctness. *OOPSLA '89 Proceedings*, pp. 433-443, 1989.
- [3] R.J.R. Back. Correctness Preserving Program Refinements: Proof Theory and Applications. *vol. 131 of Mathematical Center Tracts*. Amsterdam: Mathematical Center, 1980.

- [4] IBM Corporation. IBM's System Object Model (SOM): Making Reuse a Reality. *IBM Corporation, Object Technology Products Group*, Austin, Texas. <http://www.developer.ibm.com/library/ref/reference.html>
- [5] Walter L. Hürsch. Should Superclass be Abstract? *Proceedings ECOOP'94*, M. Tokoro, R. Pareschi (Ed.), LNCS 821, Springer-Verlag, Bologna, Italy 1994, pp.12-31.
- [6] Gregor Kiczales, John Lamping. Issues in the design and specification of class libraries. *OOPSLA '92 Proceedings*, pp. 435–451, 1992.
- [7] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*,16(6):1811-1841, November 1994.
- [8] Leonid Mikhajlov, Emil Sekerinski. The Fragile Base Class Problem and Its Solution. *TUCS Technical Report No 117*, Turku Centre for Computer Science, Turku, June 1997.
- [9] Carroll C.Morgan. Programming from Specifications. *Prentice-Hall*, 1990.
- [10] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. In *Science of Computer Programming*, 9, 287–306, 1987.
- [11] Cune Pfister, Clemens Szyperski. Oberon/F Framework. Tutorial and Reference. *Oberon microsystems, Inc.*, 1994.
- [12] Dick Pountain, Clemens Szyperski. Extensible Software Systems. *Byte Magazine*, 19(5): 57–62, May 1994. <http://www.byte.com/art/9405/sec6/art1.html>.
- [13] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *OOPSLA '86 Proceedings*, pp.38-45, 1986.
- [14] Raymie Stata, John V. Guttag. Modular reasoning in the presence of subtyping. *OOPSLA '95 Proceedings*, pp. 200–214, 1995.
- [15] Patric Steyaert, Carine Lucas, Kim Mens, Theo D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. *OOPSLA '96 Proceedings*, pp. 268-285, 1996.
- [16] Bjarne Stroustrup. The C++ Programming Language. *Addison-Wesley*, 1986.
- [17] David Taenzer, Murthy Gandi, Sunil Podar. Problems in Object-Oriented Software Reuse. In *Proceedings ECOOP'89*, S. Cook (Ed.), Cambridge University Press Nottingham, July 10-14, 1989, pp.25-38.
- [18] Peter Wegner, Stanley B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. *Proceedings ECOOP'88*, S. Gjessing and K. Nygaard (Ed., LNCS 322, Springer-Verlag) Oslo, August 1988, pp.55-77.
- [19] S. Williams and C. Kinde. The Component Object Model : Technical Overview. *Dr. Dobbs Journal*, December 1994.

The Challenge of the Global Software Process

Tobias Murer

Computer Engineering and Networks Laboratory
ETH Zuerich, Switzerland
murer@tik.ee.ethz.ch

The emerging technologies such as software components and the Internet challenge the way software is produced and marketed. The social, technical and organizational aspects of the software business change significantly compared to the traditional understanding. Discussions about software components are often mainly limited to the technical aspects of interoperability. The purpose of this position paper is to motivate for a broader interdisciplinary discussion about components including technical aspects, but also organizational, social and even marketing aspects. We investigate these various aspects to develop the concept of a software engineering environment capable to face the outlined challenge.

1 Introduction

Software components [1] are a promising paradigm shift and could answer the software crisis by attacking the conceptual essence of the difficulties inherent in the nature of the software [2]. Compared to other engineering disciplines where components are used successfully and decide the business success, software component technology is not yet mature. Discussions about components are often mainly limited to the technical aspects [6] of interoperability. Contrarily, industry reports to be more challenged by organizational or social aspects when using components in large software projects. Cox [3] argues that the difficulties we have long experienced in the software field including our efforts to reuse software components are the result of a "technocentric" view of software development. I agree and provocatively add that component research issues do not differ significantly from object research issues from a technical point of view. The specific nature and challenge of components become only visible if we accept software production and markets to happen in a complex socio-technical system. Thus, to take full advantage and to evolve the paradigm shift we need to go beyond objects. The investigations should encompass technical but also social, organizational and market issues. This also allows to distinguish components from objects in an attempt to gain a more specific focus on what components are all about.

The emerging technologies such as software components [4] and the Internet will have an increasing impact on the way software is produced and marketed. Traditional approaches of the software business are substituted or complemented by various new approaches closely related to the underlying emerging technology. Although the changes are triggered by technical issues the software component business is not restricted to a technical challenge. Technical aspects like interoperability issues are certainly critical for a successful deployment of the component business. But the significantly changing organizational and social aspects facing the problem of the decentralized management of software processes and products shared world-wide among

heterogeneous organizations are also a difficult task. Even marketing aspects tightly correlated with the emerging technologies should be investigated carefully to establish the various new business opportunities.

We are interested in components developed and composed across organization boundaries. We believe that developing high quality interoperating software components in a global context involving different autonomous organizations is as much an organizational and social challenge as it is a technical one. This paper intends to motivate for a broader interdisciplinary discussion about components and outlines technical aspects, but also organizational, social and even marketing aspects of software components. We investigate these various aspects in the GIPSY (Generating Integrated Process support SYstems) project to develop the concept of a software engineering environment capable to support the development and maintenance of high quality software components. Thus, our concerns are issues that are relevant for software engineering environments.

2 Component Technology

This section discusses software components and the Internet as the emerging technologies that will change the way software is produced and marketed. The Internet should be mentioned shortly as the complementary technology to software components playing two important roles. It is the enabling technology for global co-operations and for new marketing opportunities in the component business.

The repeated attempts to define a 'component' indicates that the software component technology is still far away to be mature. Everybody agrees that components are for composition. However, controversies arise on any attempt to go much further. Last year's workshop proposes a component definition [6] outlining its technical and organizational challenge: "A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A component can be deployed independently and is subject to composition by third parties." Components are products that are delivered with third-party composition in mind. Thus, components should be constructed as context independent as possible to be suitable for independent extension or composition. A dilemma is threatening the reuse and composition of components. The more a component is independent of a specific interoperability context the more it can be reused in other contexts. But, a component that is totally independent of any context can not be reused because it needs to work in a context to be composable. This contradiction challenges the design of reusable components.

Improved composition techniques are a critical issue for the successful deployment of components. Defining, providing and finding useful interoperability information become a severe problem. If we think about interoperability in a global context there is an important issue: How can developers, as well as users manage interoperability issues of a huge amount of components developed worldwide by independent providers? The current general way to describe interoperability information is to provide the component's interface definition and some additional informal documentation. This level of information is obviously not detailed enough to manage interoperability issues on a reasonable level. Unfortunately, component descriptions enhanced with semantic information are restricted to specific types of components.

Therefore we should think about little steps towards an improved interoperability management by introducing intermediate levels between the two extremes, the syntactic interface and full semantics. These approaches could be based on recent prom-

ising trends to distinguish functional and non-functional aspects of a component and to cope with every aspect separately [7]. A rather pragmatic non-technical approach is to focus on information inherent with the development process. During the development process of a component, the developer needs to investigate how and with which other types of components his component interoperates. Therefore, the information about which versions of other components a component requires or interoperates with is inherent in the corresponding development process. We propose to retain this information and publish it in an appropriate form. The idea is to encourage developers to make as much information available as possible about the contextual assumptions they based their development on [5].

The promising idea of a hybrid approach is to use whatever notation is most appropriate and least general-purpose for the description of every single component within a system. Appropriate means that the notation should allow to describe the structure and behaviour of a certain component as complete and as abstract as possible. A user can learn as much as possible about the structure and behaviour of a component by reading such a description. This implies that the least general notation that is barely sufficient for the task should ideally be employed. E.g., if the model of a system part is based on a finite state machine, we should take a finite state machine notation to describe this component rather than using the more 'powerful' programming language C. The composition of two components representing a finite state machine can be done on a higher level than the composition of two C programs. Such a hybrid approach enables improved composition at least in certain areas of a component-based system. It also allows to use more domain-specific languages that even can be used by domain engineers to develop families of applications that are easily specified.

3 Organization

To understand the software component business we should carefully observe other engineering disciplines where components play a key role. Car manufacturers for example are managing a complex network of enterprises delivering the various components of a car just in time to be put together to the final product. The various nodes of this network have to co-operate tightly on a high level to assure a high quality work including the areas of development, production, maintenance and marketing. Therefore, enterprises weaken their boundaries and appear as one unitary organization from the viewpoint of an external observer and build a so-called 'virtual organization' [8]. This flexible and dynamic organization concept has many advantages, single nodes can for example be exchanged by alternative enterprises. The organizational and social aspects of such a heterogeneous virtual organization are an immense challenge since the organizations differ typically in many aspects such as location, technology, methods, culture, policy, strategy, skill, quality and more.

The analogy to the software component business is obvious: Different component producers need to co-operate tightly on a high level to assure proper component interoperability by sharing and linking their development processes resulting in tightly co-operating virtual organizations. But compared to the car manufacturer there are two major differences that make virtual organization within the software component business much more challenging to manage. The car manufacturer keeps its organizational structure quite stable once developed and the applied composition techniques are more advanced. Within the software component business the product has a more dynamic nature and virtual organizations are formed more dynamically. Component producers

form a virtual organization with the goal of building a configuration of properly inter-operating components. In fact, for every client that likes to use a set of inter-operating components, there is a virtual organization of component producers co-operating to meet this specific requirement. As a vision, the development efforts of many individual component producers together may be regarded as one large global development process consisting of many linked processes. Managing such a software process on a high level across enterprise boundaries within a virtual organization is an enormous challenge since the various process parts are managed decentralized and linked into the global context of the virtual organization.

Besides components Java also popularized concepts like 'write once, run everywhere' and the distribution of software components across the Internet. Components are collected from different nodes of the global Internet and configured on the client machine to support a certain task. Everyone knows the problem of hyperlinks in the World Wide Web pointing to nowhere. The obvious reason for that is the decentralized management of link configurations by every web participant not controlled by any organization or policy. Whereas web surfers are just upset about the missing links, the consequences in the component business are more dramatic. If a client intends to collect a configuration of components from different locations of the web, every component should be available at the right version and the configuration should be confirmed from the virtual organization of the involved component producers as a stable package of properly inter-operating components. Missing versions and incorrectly configured sets of components can not be accepted within the software component business. Versions and configurations need to be available, persistence and stable. Assigning expire dates to components and valid configurations should be considered. In addition, versions of compilers and runtime systems should also be kept, especially if we think about software only compiled and distributed on demand.

The consequences seem to be significant, if we focus on the responsibility of providing component interoperability. If a producer for example releases a new version of an operating system, everyone hopes ('plug and pray' mentality) that actual software versions running on top of the new release of the operating system still work properly. In fact, every software running on top of an operating system is invalidated after a new release of the operating system until the component producer establishes a new link from its component to the new version of the operating system confirming proper interoperability. The complex network of component versions, configurations and dependencies updated decentralized is difficult to control. This world-wide global configuration management system for products is a severe organizational challenge. Introducing a simple version concept to Java is definitely a first step to face the challenge [9]. There are also new business opportunities like third party configuration dealers (e.g. 'virtual software house') managing and offering stable configurations.

4 People

People are the most critical resource in building software systems efficiently and effectively and decide its success. Humphrey [10] puts an emphasis on two basic requirements that decides successful software development - a well defined, quality based process and the best technical people. He considers identifying, motivating and organizing innovative people as a critical task. As in no other engineering discipline, highly skilled and motivated people can be more productive by factors than people with inferior education and poor motivation. Highly skilled and motivated people, communication and

its consequent, organization are critical for success [2]. Whereas skill and motivation can be improved with an appropriate human resource management (e.g. with a development oriented human resource management [11]) enabling communication within a virtual organization is a severe challenge. Communication among the involved heterogeneous organizations that differ in various aspects such as culture, policy, strategy, skill, methods and others require simple communication mechanism. Therefore, different organizations cooperating tightly to produce high quality software within a virtual organization must share a common understanding of organization. Many projects fail because they lack organization and communication before they hit technological limitations.

5 Facing The Challenge

As outlined above, the decentralized management of products and processes and organization forms within a virtual organization represents a severe challenge with a significant impact on the concepts of supporting tools. The difficulty mainly comprises the heterogeneity of the virtual organization, since the participating autonomous organizations vary in many respects including various locations, cultures, policies, strategies, methods and more.

We are investigating the concept for a software engineering environment (SEE) framework capable to support the global software process within a virtual organization aimed to produce high quality software. We believe that the outlined significantly changing challenge force the requirements for a SEE to be revised compared to traditional concepts. A SEE should more support between the nodes of the organization than within the nodes where autonomy is particularly preserved. There is a strong need for a common understanding of organization to be shared among the participating organizations. Thus, a SEE should enable communication about organization rather than forcing the various nodes to use sophisticated models. We concentrate our investigations on two distinct layers with different requirements. On the engineering layer we focus on the specification and generation of highly integrated tool components supporting formal languages. Compiler-compiler technology is extended by the object-oriented paradigm to a tool specification technique providing powerful decomposition, extension and import mechanisms [5]. This allows for a fast transformation of formal methods to describe components and their composition into supporting tools. The use of formal methods makes a program independent of its developer context. This is vital for the development of high quality software across organization boundaries. The proposed flexible approach is also appropriate to support hybrid approaches as mentioned above since various tools are needed to support the increasing number of involved languages.

Whereas the engineering layer requires sophisticated techniques enabling detailed tool descriptions, the requirements for the organizational layer are rather different. On the organizational layer the SEE should provide a simple mechanism to enable organizational integrity. Organizational integration includes two aspects; one is the common understanding of the structure and evolution of product, process and organization form shared among all development process participants (users and tools); the other is the management of relationships among the different organizational structures (e.g. answering the question of which part of the product is the result of which process steps and who was the executing developer). Keeping track and understand the evolution of process and product is essential for process improvement within iterative and explorat-

ive development methods. We believe organizational integrity to be the most important design issue of a SEE aimed to face the outlined challenge.

A promising and rather obvious approach to handle organizational integration is to achieve structural unity among the three main organizational structures [11]. It means using similar structures to manage the organizational aspects of product, process and organization form. This also allows to represent structural aspects and to establish relations among the three structures in a simple way. We use a simple 3D model to represent structure (2 dimensions) and evolution (3rd dimension) of a software product and its process. Development processes of different components that interoperate in a certain way are linked together. The links are established on different layers with respect to the history dimension, thus indicating which versions of the components interoperate. Whereas this approach promises a rather easy way to obtain organizational integrity, its limitations are also clear. To gain structural unity each structure has to sacrifice some of its specific characteristics. The resulting shared structure should be very simple. We claim that gaining organizational integrity by sacrificing sophisticated but aspect-specific structures is the better choice than vice versa to provide powerful methods to face the challenge outlined in the section above.

6 Markets

All aspects of the traditional software business such as software engineering, maintenance, distribution and marketing are challenged by the emerging technologies. Because of the impact of the emerging technologies on component marketing aspects a tight cooperation between component development and marketing is required to establish the various new business opportunities triggered by the technology shift. The emerging technology gives highly skilled small competitors the chance to compete on the global market. Components allow small competitors to provide a solution for a specific need, there is no need for a single provider to offer an integrated all inclusive product. Such monolithic products can be substituted by competitive configurations of components provided by virtual organizations dynamically built among small providers. The Internet is the perfect platform for cooperation and marketing to reach the global market with only low investment. The component market may transform the producer-driven software market into a customer-driven one. Some potential new business opportunities that could be built on top of the simple model introduced above are outlined in the following.

An electronic product catalogue with interoperability, version and configuration information about components could be built on top of the model introduced above and could be an important part of a virtual software house of components on the World Wide Web (software offers, catalogue, on-line consulting, distribution, updating, pay-per-use and more). Such an interactive catalogue could be provided with the outlined method. Attractive 3-dimensional navigation through virtual software stores may be performed in order to find consistent combinations of components. Once a user has determined a configuration of components to be valid, it can be automatically downloaded and installed.

Managing components, in particular finding out whether a given set of components is consistent, is often a difficult task. Here, software agents may be envisaged that roam over the global network of linked processes to search for this information, as consistency checking can be performed automatically according to fixed rules. A user may check the consistency of a specific set of components, i.e. those currently on

his machine, or a set assembled by himself, possibly containing components of many different developers. This technique could be used by value-added resellers and system integrators to provide and guarantee available, stable and maintained configurations of properly interoperating components.

An interesting business opportunity is a virtual organization (task force) of highly skilled information technology experts that is only built dynamically to work on a certain project. The experts represent as subcontractors the nodes of the virtual organization whereas the network is organized by the management and main contractor of the virtual organization. There is no static organization, experts become only part of the virtual organization if their specific skill is required in an ongoing project. The management acquires new projects, it establishes or maintains contacts to potential subcontractors and manages their competence and availability. The management also leads the projects including the assignments of appropriate experts to the projects. These highly skilled task forces are an effective and efficient way to work on software projects based on the idea of bringing people with the appropriate skills to the concrete projects in a flexible way. The challenge is to organize such task forces and to lead these sophisticated organization forms. The model introduced could be an approach to supporting such virtual organizations.

7 Conclusions and Outlook

All aspects of the traditional software business are significantly challenged by emerging technologies such as software components, Java or the Internet. There are major changes of the social, technical and organizational aspects of the software component business compared with traditional approaches, especially, if the focus is put on the use and development of components within virtual organizations. The decentralized management of products, processes shared among heterogeneous virtual organizations is challenging but essential for the development and marketing of high quality software components in a global context. To face the challenge future SEEs need to provide more support between groups across organizational boundaries and less within groups where skill and motivation of group members decide the success and not a sophisticated tool. The software engineering environment community should carefully consider these changing aspects for the concept of SEEs.

A simple model needs to be defined enabling organizational integration within virtual organizations, which encompasses the common understanding of the structure and evolution of product, process and organization form shared among all process participants (users and tools). We propose a similar 3D model to represent the structure and evolution of a software product and its process. This leads to the structural unity of product, process and organization form, which we believe to provide organizational integrity. This simple model could be the underlying concept of future SEEs supporting software process and configuration management within virtual organizations. A closer coupling of product, process and organization form is an important topic that will be supported by future practical large-scale software engineering environments.

Bibliography

- [1] B.J. Cox. *Planning the Software Industrial Revolution*. IEEE Software, November 1990, pp. 25-33.

- [2] F.P. Brooks. *The Mythical Man-Month* Essays on Software Engineering, 20th Anniversary Edition, Addison Wesley, 1995
- [3] B.J. Cox. *No Silver Bullet Reconsidered* American Programmer Magazine, November 1995
- [4] J. Udell. *Componentware* BYTE, pp.46-56, May 1994
- [5] T. Murer, D. Scherer, A. Wuertz. *Improving Component Interoperability Information* Workshop Reader ECOOP 96, June 1996, pp. 150 -158
- [6] C. Szyperski, C. Pfister. *WCOP'96 Workshop Report* Workshop Reader ECOOP 96, June 1996, pp. 127 - 130
- [7] G. Kiczales. *Aspect-Oriented Programming* Proc. ECOOP 97, 1997
- [8] W. Davidow, M. Malone. *The Virtual Organization : structuring and revitalizing the corporation for the 21st century* Burlingame, 1992
- [9] M. Jordan, M. Van De Vanter. *Modular System Building with Java(TM) Packages* Proc. Software Engineering Environments Conference (SEE97), April 1997
- [10] W.S. Humphrey. *Introduction to the Personal Software Process* Addison Wesley, 1997
- [11] T. Murer, D. Scherer. *Structural Unity of Product, Process and Organization Form in the GIPSY Process Support Framework*, Proc. Software Engineering Environments Conference (SEE97), April 1997

Coupling of Workflow and Component-Oriented Systems

Stefan Schreyjak

University of Stuttgart, Germany
Stefan.Schreyjak@informatik.uni-stuttgart.de

The use of component-oriented systems in a workflow system solves several problems of today's workflow systems like e. g. lacking adaptability. For that purpose, however, both systems must be modified and coupled in a two level system. In this paper several modifications are proposed that will improve the systems' cooperation, like the creation of interfaces and the provision of a process context, as well as the introduction of interactively changeable views in the application.

1 Introduction

Workflow systems and component-oriented systems have similar assignments and use similar technical approaches to facilitate the development of applications. But they differ in the application domain and in the magnitude of the application. Workflow systems control enterprise-wide applications that consists of activities. Component-oriented systems control local applications that consists of software building blocks. This paper is going to show how both systems can benefit from each other.

First, basic concepts and terms of workflow management are introduced. Some definitions in the area of component-oriented programming follow. In section 3 several problems are described with which today's workflow systems are confronted. These problems motivate the use of component-oriented programming in workflow systems. A pure combination of both systems does not result directly in the desired success. The newly emerging inadequatenesses are explained in detail in section 6. In the following section a coupling of the systems instead of a combination is demanded. This is proved by proposing an approach for each itemized problem how the problem can be solved by a modified workflow system and a modified component-oriented system. In the last section all essential ideas are summed up.

2 Workflow Management Systems

This section introduces essential concepts and terms [WfM96] of workflow management, as well as terms of component-oriented programming.

A *workflow management system* (WFMS) is a software system for the coordination and cooperative execution of business processes in distributed heterogeneous computer environments. The objectives of a WFMS are in a first phase the modelling of the structure of an enterprise and of the sequence of all business procedures, and in a second phase the controlling, supervising and recording of the execution of all modelled processes.

A sequence of business procedures is modelled in a *business process*. All sequential or parallel relationships between process steps are specified in business process models. It is determined for each step which work objects (data and documents) and which human, technical and organizational resources are necessary for the execution. A business process is modelled as a graph with process steps as nodes and control and data flow relationships as edges. An executable process model is called *workflow*.

A process step, also called *activity*, is a piece of continuous work executed by one person. The *actor* of an activity can use an interactive application program to fulfil the objective of the activity. As an alternative, manual activities without computer assistance might also be possible as well as automated activities which do not need human interaction.

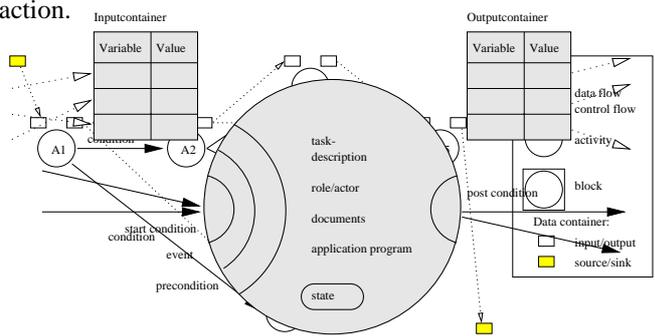


Figure 1: Model of a business process

Figure 1 shows a model of a business process which is used in IBM's workflow system "FlowMark" [LR94] and in Surro [SB96a]. All activities are connected by control flow connectors. Each activity has an input and an output container used for input and output data. The transport of data is controlled by data flow connectors.

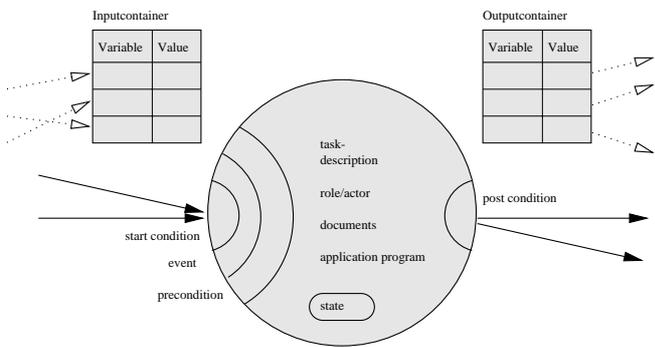


Figure 2: The inner structure of an activity

Figure 2 shows the inner structure of an activity. The input data container consists of a set of variables which are set in the beginning by data flow connectors. The application can read data from the input container and write data into the output container. Incoming control flow connectors are evaluated in the start condition. If this condition is true, the activity waits for the occurrence of an event. When the event has occurred or no event was specified, the precondition is tested. If it is true, the activity is assigned to an actor and an entry is put on his work list. The actor can then start the program

associated with the activity. The application processes the data in the input container and writes output data. The workflow system can test with the postcondition if the task of the activity has been accomplished successfully. If this is not the case, the activity is return to the same actor.

The central component of a workflow system is the workflow engine. It reads the workflow specification, instantiates the workflow, dispatches the scheduled activities to the actors and transports data and documents to them. Application data and documents, as well as workflow management data, are stored in a database or document management system. The engine uses an organization module, which knows the whole organizational structure of the enterprise, to cast a role, e. g. that of a clerk to a real person. This module can also be used as a generalized resource planner. The specification of the business process and the organizational structure are created with a visual workflow editor. A workflow information system shows several views on the actual state of the workflow system and allows to view even the history of the processing. An activity manager is used as an actor's user interface to the workflow system. The actor uses this program to start workflows and activities allocated to him.

Workflow systems are mainly used in enterprises with highly structured business processes which are executed very often. Users of a workflow system are actors incorporated in the process and work managers who model the business process. Work managers must have programming skills. They remain, however, specialists for the business process, not for the programming.

Programs which are assembled using components can be executed in activities. Because this domain is of great importance some terms of *component-oriented programming* (COP) are described in the following to clarify how the terms are used in this paper:

Components are reusable software building blocks which can be assembled or disassembled in an application without violating its integrity. There are atomic and composed components. Atomic components offer functionality of one specific domain, i. e. functionality must be distributed among components in a way that interfaces are of minimal size.

A *component model* specifies an architecture which enables components to interact and consists of a set of protocols. Components are instances of one component model.

A *component-oriented system* is an architecture for a software system which provides the infrastructure for components. An implementation of this architecture consists of an application development environment and a runtime environment for components. A *component-based system* uses components only in the development environment. In the runtime environment (respectively the compiled application) components do not exist anymore.

The term *component-oriented application* defines an application which is built using components as building blocks and is used to distinguish itself from applications conventionally built.

3 Problem Description

The use of current workflow systems reveals some problems. In this section all those problems are discussed which can be solved or reduced using component-oriented programming.

Adaptability Problem in Activities

Before the emergence of workflow systems, business processes were realized with big monolithic business applications. In order to preserve marketability, business processes must be adaptable to the actual market situation. E.g. customers requests can result in the need for adapting business processes or the management wants to increase productivity by improving the process. These requests for change must be implemented by programmers of business application vendors. Hence, adaption is costly, time-consuming and expensive.

These facts have resulted in the development of workflow systems. Hereby, business processes are divided into a function-oriented part — the work steps — and a process-oriented part — the relationship between the steps. One step in the work of an actor is used as a criterion for splitting the business application into activities. These relationships are described in the workflow specification modelling the business process. Because of the more abstract representation of the workflow, the user can adapt the workflow more easily than programmers can adapt their monolithic business application.

Workflow systems, however, only allow adaption in the process-oriented part. If the function oriented part has also to be changed, the old problem arises: adaption in applications has to be implemented expensively by the vendor.

Heterogenous System Environments

Workflow systems are used in a very heterogenous infrastructure of computer equipments, which is grown by and by. Hence, workflows have to be platform independent. The specification is already platform independent, but the applications are not. Therefore, an activity needs one platform specific application for each platform. Every application used in a workflow system needs to exist in several portations: one application for each platform. Alternatively, applications with similar functionality have to exist for each platform. If a lot of different programs exist several problems arise, like operating errors while interacting with the application, missing functions or incompatible data formats. Altogether, this approach is unsatisfying, costly and expensive.

Incomplete Reuse of Workflows

The reuse of workflows in a different enterprise requires the possibility of adapting the workflow by the user and the possibility to run the applications on all used platforms. As a consequence of the two subsections above reuse of workflows is not really possible.

Unspecific Applications

Workflow systems are especially useful when workflows are executed often. Therefore, the applications should be accommodated very well to their operation area in order to achieve high productivity. But customized applications must be bought expensively from software vendors. Instead, generic standard applications are often used. But these applications can only be limited incompletely to their task domain. The user is allowed to do much more than he should. The workflow system has only few possibilities to check what is done in activities. The incomplete adaption to the task domain makes the task more costly and error prone than necessary. For example, automation of mini control flows can usually not be done in this generic applications.

No Mutual Profit of Services

Standard applications have no knowledge about their use in a workflow system. So they cannot use services offered by the workflow system. Vice versa, the workflow system sees activities as black boxes. It is not able to control the applications in the

activities. As a consequence of this lacking arrangement, services may be implemented twice.

4 Combining Workflow and Component Systems

An approach is to use component-oriented programming for the development of applications in activities. Workflow systems are a large and rewarding application domain because the complexity of program logic in activities is typically low and often similar. It turns out quickly that today's systems are not prepared satisfactorily for that use if they are not modified for that purpose. So, the combination of these unmodified systems shows the problems mentioned above in a different perspective:

- A simple combination does not eliminate the problem of lacking mutual profit.
- Now, the user has the possibility to create customized applications with components providing standard functions. Unfortunately for every new adaption (like a different color) a new additional application is needed. In addition, applications cannot be customized dependent on the task domain (e. g. a certain activity in a process).
- The problem of heterogenous system environments is not solved if the components remain platform dependent.
- Reuse will be possible when the problems of heterogeneity and adaptation will be solved.
- The applications in the activities can be adapted by the user if component-oriented systems are used. This is not valid for a component-based system! Indeed the possibility to adapt applications is not supported very well from today's component systems. The user cannot adapt an application easily:
 - ◊ Functionality dealing with the cooperation of components can only be gathered with difficulty. Events and callbacks are used as a communication technique. The code of these callbacks is spread over all components. Therefore, the user must accumulate the knowledge of the structure of a component-application in a reengineering phase by himself. There is no support by the system. Moreover, often only one structuring element exists, the hierarchy of GUI-widgets (controls).
 - ◊ The replacement and addition of components is not supported in an adequate way by today's component systems. There are no integrity checks whether all required interfaces and preconditions are met when using a new component. The user has to test it by himself.
 - ◊ The glue code is necessary for the creation of callbacks and allows the cooperation of components. Glue code is usually written in the same language with which the component is implemented, and not in a language which is tailored to user skill. Such a language should be easy-to-learn, easy-to-understand and easy-to-use. Most of all the code must be interactively testable.

5 Coupling Workflow and Component–Oriented Systems

The combined use of unmodified systems solves the problems mentioned only partly. Therefore, it has to be examined if it would not be more successful not only to combine but also to couple both systems. In such a coupling both systems can be modified and enhanced with additional functionality to solve these problems. Objective of the coupling is to allow transfer and adjustment of similar technologies for a redundancy–free partitioning of responsibilities and to allow a highly effective cooperation. The coupling results in a two level system where each single system can be used on its own, too.

In the following subsections, a solution is proposed for each problem mentioned. Also, some technical aspects concerning implementation aspects are discussed.

To: No Mutual Profit of Services

Both systems must be modified in a way that they offer their services in public interfaces. Double implementations can be avoided if it is defined which system implements the service and which system uses the service. A replacement of one system is only possible when these interfaces are standardized.

An example for such an API are controlling functions for components. A controlled abort, a resume, as well as storing and loading of component states are important functions to support error handling of the workflow system [SB96b].

An example for the distribution of functionality is the transport of software. With the help of internet protocols a workflow system can transport and install applications at the actor's computer [SB96a]. This functionality can also be used for the transport of components, just like the loading of java applets in a html–page.

A more advanced example for distribution is the realization of persistent applications. Workflows are persistent because activities are considered atomic units. Component–oriented applications are generally not persistent, but in the application domain of workflow management persistent applications are desirable because an actor may want to change his working place within a running activity. For that purpose complete persistence is not necessary, all what is needed are some persistent states. Hence, a component–oriented application can be divided into activities such, that the workflow system is able to support the persistence. The component system does not need to implement it, too.

To: Unspecific Applications

Component–oriented applications need to be aware of the surrounding process to be able to customize to a specific task. This can be achieved with a process context.

A *process context* is a logical storage used by components to store and retrieve data. The workflow system provides and manages different contexts. Contexts can be defined through conditions. For example, a person can form a context, but also a workflow instance or an activity in a workflow template. The concept of a process context allows *context sensitive* applications.

A component can store the attributes' values, configuration scripts or other data in the context. For example, a context sensitive command history can be implemented. Default values in data entries are another example. Thus, an actor can use the old data he had typed in last time using this application.

A context database can be compared with the X–resource database, with the restriction that only one context exists which is identified via the application name or class.

The data container concept can also be seen as a primitive process context.

To: **Heterogenous System Environments**

The problem of heterogeneity can be overcome if the component system and the components are implemented in a platform independent programming language. The object-oriented language Java was designed for that purpose. But a number of other languages exist which also possess that property, like Tcl/Tk [Ous94] or Python [WvA96].

Proprietary component models like ActiveX and OpenDoc, as well as platform dependent components can be used in a component system through proxy components or bridges. A proxy delegates all accesses to the proper component which runs in a component system on a remote node. This should be transparent to any user of the system.

To: **Adaptability Problem in Activities**

Analogous to the modelling of workflows, application must be modelled to allow easy-to-use adaptation for the users.

The user can change an application built with components on three adaptation levels: The first level is limited to changes of the component attributes. On the second level glue code used for the component communication can be adapted to change the cooperative behaviour of the component-application. These two levels can be controlled by the process context. On a third level the user can replace and add whole components and change the structure of the component-application.

To perform this adaptation work the user needs easy-to-use editor tools. Editors for component attributes can be found in many existing component systems. Editors for the glue code consist of conventional programming editors. No editor exist for the communication structure. Here, graphical methods are necessary to describe the functionality of the whole application on a level of high abstraction. Tools for a structural change can be found in interface builders common to all component systems, but the structure of the application is identical to the hierarchy of the GUI-elements.

Adaptations made by the user have to be stored. The scope of adaptations may differ. Global adaptations are valid for all users, while local adaptations are only visible in a certain context.

The understanding of a component-application's functionality can be improved by the introduction of views. A *view* shows a certain aspect of the application. The user can use a view to change something interactively in the component-application. The following views has been identified up to now:

- The GUI-view shows layout and structure of the GUI-hierarchy.
- The structure view shows the hierarchy of components in form of a tree.
- The model view visualizes the functionality of (composed) components. A model of a component abstracts from implementation details and facilitates the understanding for the user. Now, the user can make adaptations at the design level instead of the implementation level by giving access to the glue code.

An open question remains as to which visualization models are suitable for the user group. The research domain visual programming and object-oriented design can give a stimulus to that. Enhanced state charts [Har88] would be one possibility currently being considered.

6 Related Work

Leymann proposes in [Ley95] a method to build business applications in an object-oriented manner through removing all flow dependencies from business objects. Flow control will be managed by a dedicated control object. Adler [Adl95] calls this coordination model process planner. He had implemented control objects with the help of tcl scripts. Leymann uses a workflow system as a control object and calls this method “heavyweight scripting” in contrast to “lightweight scripting” found in typical script languages like tcl, perl, rexx, etc. The workflow system Surro [SB96a] realizes this concept. The workflow engine uses an ORB to call business object methods in an activity. Therefore, Surro can be seen as a component-oriented system with business objects as components. These concept *integrates* a workflow system and a component system into a single level system. For each activity a corresponding component exists and the engine is used as a central control unit.

The flexibility of such business applications benefits from process modelling. But it results also in bad scalability. Using small components increases the number of components and consequently the expenditure of the centralized workflow engine. The engine has to handle each component as an activity. Therefore, only a small number of complex components are used as business objects which model typically real entities of the business domain, like application forms or business accounts.

Consequently, the integration concept seems not to be adequate for small components. System requirements of workflow and component systems are too different to be combined in a single level system.

7 Conclusion

In today’s workflow systems the adaptability of workflows ends at platform dependent activities. This results in bad reusability. Furthermore, control over applications is minimal and applications cannot use workflow services. Therefore, potential for automatism is not used.

A first approach to improvement is to use component-oriented systems in activities. But a simple combination of unmodified systems does not succeed satisfactorily. There is a need for higher coupling.

The creation of a two level system-coupling requires several modifications in both systems. The following modifications have been proposed in this paper. The introduction of interfaces will allow redundance-free partitioning of system tasks. The introduction of a process context will enable the use of highly specific applications (context sensitive applications). By using interactive editable views, modelling the functionality of an application built of components, the user can more easily adapt these applications.

Integration of both systems into a single level system results in bad scalability. In the next project steps the approaches mentioned will be enhanced and evaluated in a prototype.

Bibliography

- [Adl95] ADLER, Richard M.: Distributed Coordination Models for Client/Server Computing. **In:** *IEEE Computer* 28 (1995), April, Nr. 4, S. 14–22

- [Har88] HAREL, David: On Visual Formalisms. **In:** *Communication of the ACM* 31 (1988), May, Nr. 5, S. 514–529
- [Ley95] LEYMANN, F.: Workflows Make Objects Really Useful. **In:** *Proceedings of the 6th International Workshop on High Performance Transaction Systems (HPTS)*. Asilomar, September 1995
- [LR94] LEYMANN, Frank ; ROLLER, Dieter: Business Process Management With FlowMark. **In:** *Proc. 39th IEEE Computer Society Int. Conference (CompCon)*. San Francisco, Cal. : IEEE, Februar 1994, S. 230–234
- [Ous94] OUSTERHOUT, John K.: *Tcl and Tk Toolkit*. Massachusetts : Addison Wesley, 1994
- [SB96a] SCHREYJAK, Stefan ; BILDSTEIN, Hubert: Beschreibung des prototypisch implementierten Workflowsystems Surro. Universitt Stuttgart, Software–Labor. – Fakulttsbericht Nr. 1996/19, Software–Labor Bericht SL–5/96
- [SB96b] SCHREYJAK, Stefan ; BILDSTEIN, Hubert: Fehlerbehandlung in Workflow–Management–Systemen. Universitt Stuttgart, Software–Labor. – Fakulttsbericht Nr. 1996/17, Software–Labor Bericht SL–3/96
- [WfM96] WFMC. Workflow Management Coalition Specification — Terminology and Glossary, 1996.
<http://www.aiai.ed.ac.uk/WfMC/DOCS/glossary/glossary.html>
- [WvA96] WATTERS, Aaron ; VAN ROSSUM, Guido van ; AHLSTROM, James: *Internet Programming with Python*. MIS Press/Henry Holt publishers, October 1996

A generator for composition interpreters

Jørgen Steensgaard-Madsen

Information Technology, Building 344, DTU
DK-2800 Lyngby Denmark
jsm@it.dtu.dk

Composition of program components must be expressed in some language, and *late composition* can be achieved by an interpreter for the composition language. The point taken here is that a suitable notion of component is obtained by identifying it with the semantics of a generalised notion of a structured command. Consequently experiences from programming language design, specification and implementation apply.

Composition of statements in structured-programming languages depends on principles that can be generalised and applied to a rich family of components. A particular component can be considered as defining objects or a commands according to convenience.

A description language including type information provides sufficient means to describe component interaction according to the underlying abstract notion of components. Actual compositions can be type checked before execution.

1 Introduction

An application program may contain an *interpreter* that maps *commands* into computations which are characteristic for the program's application domain. The set of acceptable commands is called the *command language*. Typically an interpreter is programmed to operate in a cyclic fashion: read a command, perform the computation, repeat. The term *command language interface* is used to describe this situation.

Interpretation seems also to be relevant for late composition of independently developed components: some expression is needed to indicate an actual composition and the set of possible expressions forms a language. A useful notion of *components* is obtained by focusing on a particular form of commands. A component essentially expresses the semantics of a command. Ultimately one might want both an interpreter for that language as well as a compiler to combine components tighter than possible with an interpreter.

Checked integration of independently contributed components requires formal descriptions of the components. This corresponds to the description of syntax for commands of ordinary programming languages. Use of a type system in the formal description provides the basis for checking an actual composition before executing it. The set of possible descriptions forms a description language and the set of languages that may be described constitutes a language family.

The various tasks involved in the construction of a command language interface can best be compared to the design of a programming language and the implementation of a corresponding compiler/interpreter. In itself the design of a decent programming language is no simple task and neither is the construction of an interpreter or compiler.

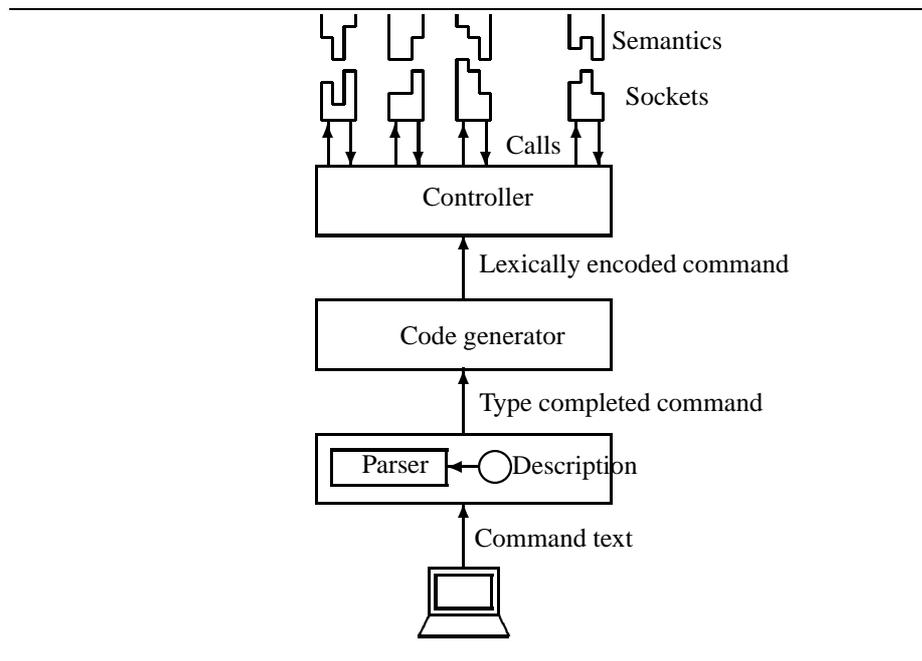


Figure 1: System structure

Tools exist for the construction part, but these are targeted primarily at computing science specialists.

This paper describes an effort to generate interpreters for composition languages. It is partly justified by the view that a portable interpreter construction tool with its language family may form a standard for a component market. Figure 1 shows the overall structure of the intended system, Figure 2 gives a view on the design space, and Figure 3 summarises the project achievements.

The generation process requires the availability of components in object module form and a description of each component. The generated interpreter dynamically combines components in accord with the way components might be combined statically in an ordinary program.

The paper is organised as follows: Section 2 introduces the language family with emphasis on use, Section 3 focuses on component descriptions, Section 5 contains various observations for discussion, and Section 6 relate this to some work of others.

2 Weak abstraction languages

A family of languages, called *weak abstraction languages*, has been identified and a tool for generating interpreters for them has been developed. The languages are intended to describe combination of commands with their semantics written in another language. In principle the components that implements the semantics may be written in different languages.

The notation for simple commands is closely related to other structured languages. Most weak abstraction languages will probably contain familiar commands for branch-

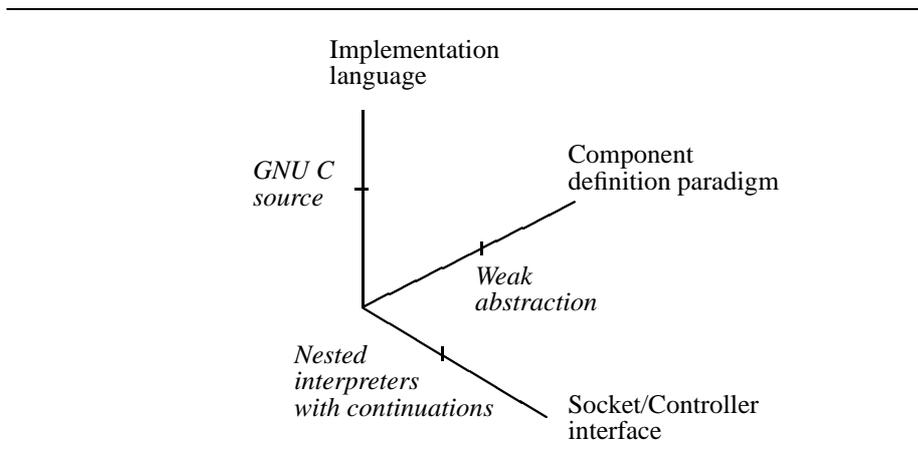


Figure 2: Outline of the design space.

ing and repetition like if- and while-statements.

An example:

```

program{
  listpackage a;
  induction(1::2::3::4::nil) {
    a.split(problem) {0} {hd+result(tl)}
  }
}

```

This is a small program that uses three components: a statement-like component `program{...}`, a class-like `listpackage` to introduce the type of lists with polymorphic operations on them, and finally `induction` with traits that makes it both class- and statement-like.

Members of `listpackage` is used to constructs a value `1::2::3::4::nil`. The member `split` is used to branch on a current list-value called `problem`: one way for an empty list and another for one that can be decomposed into the its head and tail (`hd` and `tl` respectively).

The independent components `listpackage` and `induction` interact as described in the `{...}` part of the `induction` command: the operation `split` from `listpackage` is used to break down a current `problem` (i.e. a list of integers), whereas the operation `result` obtained from `induction` is used to compute the inductively determined solution to a subproblem.

The example is a small toy program, primarily of academic interest. However, its basic principles have been used in serious applications like a compiler and a document preparation system.

Each language can be characterised semantically as a particular set of higher-order, polymorphic functions that can be freely combined within the limits determined by a type system. Syntactically the languages unify the notions of statements and objects, and they help users not to be aware of their function definitions.

A language is identified by a set of *signatures*, i.e. formal descriptions of commands, that typically determines an imperative, strongly typed, structured language. Each command has a semantics given by a component.

-
1. The family of *weak abstraction languages* has been identified.
 2. A *description language* describes introduction of type names, component and operator signatures, and operator classes.
 3. A *translator* maps source language commands and expressions into an internal representation with type information added.
 4. A *code generator* lexically encodes the internal representation of commands and expressions.
 5. An abstract *controller* interprets encoded commands as socket calls.
 6. A *dispatcher generator* generates socket routines, i.e. interfaces to the application's semantics for the commands.
 7. A *skeleton generator* can map signatures to empty semantics routines.
-

Figure 3: Summary of project achievements

3 Description of weak abstraction languages

Usually a language is described by a set of production rules that form the grammar of the language. A weak abstraction language is essentially described by a number of *signatures* which can be conceived as its grammar.

The term *weak abstraction* derives from an analogy between programs and proofs in formal logic. The principles applied here are by that analogy closely related to what is called *weak existence* in a higher-order logic. Abstract data structures and existence are related by the analogy, and thus justifies the adjective *weak*.

The important fact about weak abstraction is that a programmer's notion of an abstract data structure can be expressed in some languages without special constructs devoted to that purpose. Interested readers may look at the references [4, 5].

A signature is similar to the description of a routine in a typed, polymorphic programming language. It associates a name with a number of *types*, *places* and *parts* that must be filled in when it is used in a command. Types are filled in automatically. The others may be filled in with *values* or *command sequences*. A signature in a bracket describes a part.

Consider the description of the `listpackage`-component used above:

```
SIG listpackage OF W
  [Scope OF 1 List
    [nil OF A: A List]
    [ :: OF A(A,A List): A List]
    [split OF U,A(v:A List)
      [NilCase:U]
      [ConsCase(hd:A,t1:A List):U]: U
    ]: W
  ]: W;
```

Copy a language core to a new development directory.

Add signatures to a file and generate empty semantics.

Modify the semantics and generate a system.

Test for adequacy.

Figure 4: System development steps

The italics font is used for readability only. Parts similar to those of ordinary structured statements have names. Uses of `listpackage` may have the form `listpackage{...}` with further details given in the `...` part, named *Scope* in the description. The members `::`, `nil` and `split` may be used in the *Scope*-part of a command.

A special rule applies when `listpackage` is used last in another command, as in `program{listpackage{...}}`. It may then be expressed as a declaration of a named object: `program{listpackage L;...}`. This is how the notions of statements and objects are unified.

A powerful type system is used in the description language. The unary `List` type-constructor is introduced by `listpackage`. It allows new types to be formed, so `A List` denotes the type of lists with elements of type `A`.

Types are used in component descriptors only, but can appear in error messages. Type variability is supported: the `::` operator may be applied to operands that can be said to be of types `A` and `A List`, for any `A`.

The other essential part of the initial example is the `induction` statement, which has the description:

```
SIG induction OF DATA,V (value:DATA)
  [justification(problem:DATA)
   [result(subproblem:DATA):V]:V
  ]:V;
```

The signature emphasises the adequacy of deeper nesting. Note how the use of fonts alternate with the level of nesting. Names in italics are relevant for the semantics and very useful in an informal explanation:

```
Provided the justification, for some intended f, computes
  f(0) when problem=0 and
  f(problem), assuming result(i) = f(i) for 0<=i<problem
then induction(value){justification} will compute
  f(value).
```

This applies when `DATA` denotes `integer`. Note that the names `problem` and `result` relate to `induction` as members do to classes. Likewise `nil`, `::` and `split` can be considered members of a class `listpackage`, and in one branch of `split` members `hd` and `tl` show up — also like members. This is a general mechanism that is called *implicit name introduction*.

4 System development discipline

When a system can be identified with an interpreter for a particular weak abstraction language, an effective system development discipline can be provided. System and language design become closely connected: useful components correspond to semantics of useful commands.

The construction of interpreters for composition of components has been automated. Figure 4 indicates the steps needed to extend an actual system by adding new components. Once developed and tested within one system a component can be transferred to another in object module form.

As mentioned in Figure 3 semantics routines can be plugged into sockets and tools exists to generate both the base containing the socket interface and empty semantics routines that fit into the sockets. It means that a particular language can be tested as soon as it has been described formally, i.e. programs can be expressed and considered as pseudocode in an early phase. Furthermore, the tests become more and more meaningful as the individual components are developed.

Very few systems will probably be developed from scratch since most will contain some commands and operators of general interest, e.g. if- and while-commands as well as ordinary arithmetic, so incremental development is assumed to dominate.

The prototype interpreter construction tool that has been developed depends on components implemented in GNU C and is itself implemented in GNU C. Use of GNU C for the implementation ensures high portability.

Some sophisticated language mechanisms are needed both for the interpreter and for component implementation, but the actual C extensions in GNU C suffice.

5 Discussion

As illustrated a component may be considered (and defined as) class-like when that point of view seems adequate. A component's signature is similar to a template but signatures is a more powerful notion.

Some find that the notion of *object-orientation* involves a notion of *inheritance*, which is absent here because only component *composition* is considered. It seems a bit surprising that object-orientation has an essential characterisation in terms of 'implementation' rather than use.

A supplier may contribute to a system with a precompiled component in object-module form accompanied by a signature for it. Integration will be straightforward according to the description of system development.

A fixed number of component definitions is assumed in the described system. Modification of a component involves re-integration of the entire system. Further investigations are needed if modification, removal or addition of a component should be allowed on the fly.

The essential concepts that have been used in this work are the use of types as parameters in general and the use of routines as parameters. The notion of *type completion* has been used to relieve a user from giving types explicitly in commands.

The techniques used for composition of components combine the mechanisms of structured statement composition and implicit name introduction (ordinarily associated only with abstract data structures.) Both mechanisms can be explained simply in terms of higher-order constructs. The techniques are well understood and acceptable with respect to efficiency.

Late composition of components via an interpreter is particularly interesting when it conforms to general programming language principles, because *early composition* then should be an alternative choice. It will provide the freedom either to pay composition costs once for a fixed composition, or to pay it repeatedly for the sake of flexibility. In both cases a supplier might be able to contribute the same component in object module form only so that the choice can be left to the composer.

6 Related work

Any program that takes text as input can be considered an interpreter, although often with a very primitive input language, similar in nature perhaps to assembly languages. No sharp dividing line seems useful to distinguish those with a flexible command language. One conclusion could be that language interpretation should be essential knowledge for all programmers of systems that take text as inputs.

One example of a set of tools to provide a command language interface is Tcl [3]. The kind of languages it supports are not precisely representative for established views on programming language design. Its success, not the least to define the language Tk for the application domain of building graphical user interfaces, witnesses to the value of having a command language interface for an application domain.

An interpreted, general programming language may of course be used to develop application systems. This may lead to nice systems provided the language is seen as pleasing, that the possible cost of changing existing programming habits is not considered too high, and that the resulting programs meet the users' expectations with respect to performance. The application area of Tk has been approached in this way by using the languages Scheme [1] and ML [2] as interfaces to the semantics underlying Tk.

The form of signatures presented has its roots in the design of a general programming language that has been described in [5, 6].

Bibliography

- [1] W. Clinger and J. Rees. Revised(4) report on the algorithmic language scheme. *ACM Lisp Pointers IV*, 1991.
- [2] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [3] J. K. Ousterhout. *Tcl and Tk Toolkit*. Addison-Wesley, 1994.
- [4] J. Steensgaard-Madsen. Modular programming with Pascal. *Software – Practice and Experience*, 11:1331–1337, 1981.
- [5] J. Steensgaard-Madsen. A statement-oriented approach to data abstraction. *TOPLAS*, 3(1):1–10, Jan 1981.
- [6] J. Steensgaard-Madsen and Lars Møller Olsen. Definition of the programming language Modef. *Sigplan Notices*, 19(2), 1984.

On the Addition of Properties to Components

Jose M. Troya and Antonio Vallecillo

Dpto. de Lenguajes y Ciencias de la Computación.
Universidad de Málaga. Campus de Teatinos, s/n.
29071 Málaga, Spain.
{troya,av}@lcc.uma.es

The design of components for open systems has led to the study of new systems and their properties, like the ‘independently extensible’ systems. Most of these systems try to facilitate the design of components by incorporating new features, services and utilities that solve the basic problems of open systems: heterogeneity, partial knowledge of the components, and dynamic changes in the system’s configuration. However, our approach is not focused on the system, but on the components themselves. The present paper defines the properties that components of those systems should have, and introduces a components’ design methodology based on the modular addition of properties.

1 Introduction

There is now an increasing interest in the study of Open Systems, and in particular in the Independently Extensible systems [20] as the base for a component market. In this market users are able to buy or rent reusable components off the shelf and compose them to build their applications.

Extensible systems allow new functionality to be added at run time. Independently Extensible systems allow extensions of the system to be developed by different people in ignorance of each other [21]. In these systems, the key are the software components, individual units that can be reused by any end-user to build up his system. Usually users are third parties willing to compose them without modifications. The study of the properties that systems should have to allow late composition of components is an active area of research, and Component-Oriented Programming (COP) is proving to be a natural extension of Object-Oriented Programming (OOP) for these types of systems.

Regarding the components, each one can be considered as an *individual unit*, with a job to do: its *goal*. In order to achieve it, the component needs to adapt to the environment of its future user, use the particular resources and services of that system, and be able to deliver its own services to it. Since systems are open, distributed and heterogeneous, the component must face up to the particular problems of these environments: new servers and services may appear, disappear or stop working without previous notification; the component needs to coordinate with other components whose interface is unknown; and of course, delays and errors in the communication medium cannot be underestimated or ignored. There is no longer a global vision of the system; every component has its own local vision of it, probably different from other components’ vision.

So far, most of the efforts have been invested in improving the systems, incorporating the services and facilities that allow components to run on them more easily and to adapt to their changes. Examples of these systems are Hector [2] or ASX [19], or component architectures as JavaTM Beans [15] or Aglets [14]. In them, the system developer can count on a good set of tools that allow a better design and implementation of components. However, all the facilities not directly offered by the system have to be taken over by the components themselves, which forces each designer to add them to its components in an individual fashion and therefore with no guarantee of modularity, portability, or openness. In general, system restrictions must be incorporated, if possible, separately from the component's code. Hardwiring time or other constraints into the component's code may bring undesirable results, as mentioned for instance in [4, 18].

2 Our Proposal

Our approach is not so much focused on the system itself, but on the components. We need to incorporate the desired properties to them, in a modular and independent fashion. This leads to a methodology for the design of components, whereby components can be designed to concentrate *just* on their goal (i.e. their computational aspects) without taking into account other particular issues of the systems they will run on. When an end-user buys or rents a component to build up his application, he may buy as well some add-on properties to the component, like time control if his application is time-critical, robustness if his running environment is unsafe, or durability if he pretends to update the component from time to time. In this way, components are designed independently from the systems they will run on, and the components they will compose with. Later, the appropriated properties to cope with the system's requirements can be added to them.

With this in mind, our proposed solution is based on a tripartite structure "components/add-on properties/systems" instead of the standard pair "components/systems". Systems have just to provide the supporting infrastructure for components to work and interoperate among themselves. The add-on properties are abilities that components can incorporate to adapt to the systems. Separating these concerns allows a modular and independent addition of properties to components, contributing to a simpler design.

We have identified three properties: Autonomy, Robustness and Competitiveness. Each one deals with specific problems of the open and extensible systems, like dynamic re-configuration, error detection and recovery, maintainability and adaptability.

We have also simplified the system's requirements since many of the system's services can be now incorporated into the add-on properties. Nevertheless, this does not mean that properties do not have to be designed without taking into account the final system's features. In case the end-user system incorporates some features that are part of a property's function (i.e. service trading or time control), the property should make use of them in order not to duplicate jobs or responsibilities. For instance, the property of Independence for a component running in a CORBA platform should make use of the system's trading services [12], but we ought to be able as well to add this property to a component running on a system without these services.

The contribution of this paper is twofold. On one hand we have identified some basic properties that allow components to achieve their goals in open and changing environments. On the other hand we introduce an architecture to define and implement

properties in a modular and independent fashion. To support the model we have built a general framework that allows the definition, design and implementation of properties, that can be added to components by specialization. The framework not only permits the definition of new properties but also the modification of existing ones.

3 The Properties

The aim of this section is to describe, in a brief and informal way, the properties that components need in order to cope with the problems of open and distributed systems. We have identified three major ones: *Autonomy*, *Robustness* and *Competitiveness*. *Autonomy* can be defined as the ability of a component to take its own decisions in an independent way (however, by autonomous we do not mean self-sufficient or self-reliant). *Robustness* guarantees reliability and secure access. And *Competitiveness* provides the component with a general philosophy of survival in terms of a resistance and durability.

Being such a general and complex abilities, the key idea is to define them in terms of the composition of “smaller” basic properties. Thus, we can define the property of **Autonomy** as the composition of other three properties:

- *Independence*: The component should be self-governed, able to discover the services it needs and free to decide the solutions to hire in each situation.
- *Adaptability*: The component should be able to accommodate to different interfaces and protocols, and to changes in the requirements. It has to be composable, flexible, versatile and extensible.
- *Self-Protection*: The component should protect itself against external failures and unforeseen circumstances. The component cannot depend on the rest of the components’ behavior and well-functioning.

Robustness is achieved through the composition of three basic properties:

- *Integrity*: The component has to offer a robust behavior under a variety of circumstances, different inputs and different uses of its interface, valid or invalid ones. This property may check not only pre-conditions on the incoming messages, but also verify (and put right) the order in which they are received and the time interval between them.
- *Secure Access*: Every access to and from the component must be authorized. Unlawful entries and illegal outputs have to be detected and banned. Signatures can be added and later checked using this property, and encryption mechanisms can be used. *Laws* [16] can be not only defined with this property, but also enforced.
- *High Availability*: The component should protect itself against failures in the processes or machines executing it.

Finally, **Competitiveness** can be expressed in terms of other two basic properties:

- *Best Effort/Least Losses*: The component should try its best to satisfy its users in terms of response time, functionality and quality of the services provided [22], whilst maintaining its suffering and losses to a minimum when servicing requests.

- *Durability*: The component has to incorporate mechanisms to be able to be renewable, keep itself updated and improve over time.

4 The Model and its Implementation

4.1 Components

In general, components can be seen as encapsulation of programs. The “*capsule*” abstracts the program functionality, offers a common interface to the program services, hides their implementation and allows the composition and coordination of components.

The idea we use to implement each property is by adding “*layers*” to the capsule. Each layer acts as an active *wrapper* that captures and modifies the program’s inputs and outputs, offering to the outside world an interface with the given property. Please note that in our scheme the behavior of a layer is not passive or merely computational as if it were a filter: as a result of an incoming message the layer may send one or more messages to the system, wait for their responses and build up from them the message that will be finally passed to the component. Besides, the layers can be “composed” so the component can have multiple properties or abilities simultaneously.

4.2 Communication Mechanisms

To implement this scheme we need a system’s computational model that allow components to communicate, and define the minimum requirements to do so. We have chosen a very simple one that just contains the functionality required for our purposes. Having a simple and general model will facilitate its implementation in most of the known systems and platforms: Linda [6], Actors [1], CORBA [13] or Infospheres [8].

At a low level, our model is based on processes that communicate using message queues (mailboxes). Each component is a multi-threaded process with states. Each state is defined by a set of assignments of values to the variables of the component. The communication between components is asynchronous, and achieved by sending messages to mailboxes.

Each component belongs to its *domain*: the address of the machine (or net of machines) where it lives. We shall use Internet domains as valid component domains, and name them accordingly. Thus, a component may be running for instance on the domain “@lcc.uma.es”. This allows the easy integration of our model with the WWW and the usage of some of its services, like name servers.

Each mailbox will have a unique global address (`mb@domain`), given by its name “mb” and its domain name “@domain”. If the domain is omitted, the current domain name is used. If the mailbox name is omitted, we refer to *all* mailboxes currently at that domain. This mechanism allows sending broadcast messages to a domain.

There are two basic communication primitives: `Send` and `Receive`. The first one sends a message to a mailbox address, and the second one reads a message from a mailbox. `Receive` is a blocking operation in case the mailbox is empty, while `Send` is non-blocking. It is important to note that this model allows the use of formal reasoning methods, similar to the ones outlined in [7].

4.3 Controllers

Controllers are the special processes that *wrap* the component and modify its behavior. They intercept all incoming and outgoing events by accessing its message queues, and treat them according to their strategy. They are somehow analogous to the object decorators or object adapters [10], to meta-actors [1], or to filters [4].

To specify controllers we need to define the messages they deal with and the operations they use to handle these messages. We have defined a general scheme that can be gradually specialized: firstly to specify the controllers that implement each property, and then to particularize them when being added to a precise component in a given system. The more basic scheme of a controller is as follows¹:

```
public class Controller {
    ...
    // operations
    public void Deliver(Msg m){ // Captures outgoing messages.
        Outq.Queue(m);
    }
    public void Received(Msg m){ // Captures incoming messages.
        Inq.Queue(m);
    }
    public void TimeoutExp(Msg m){ // Captures timeout conditions.
    }
}
```

The first method is invoked by the system every time the component wants to send a message out to a mailbox. Method `Received` is invoked on the receipt of a message and `TimeoutExp` allows the controller to know that a sent out message does not get an answer when expected. In general, properties do not need to deal with time; however, they can be aware of these situations with this mechanism.

Once the controller has dealt with an incoming message, its result is put back to the component's incoming message queue (`Inq`). Analogously, the controller queues in the outgoing messages queue (`Outq`) the result of the treatment done to an outgoing message. In case of having several controllers chained together (due to a composition of properties), the end of each `Inq.Queue` operation causes the invocation of method `Received` in the next controller and, analogously, the end of each `Inq.Queue` operation causes the invocation of `Deliver` in the next controller.

4.4 A Framework for Adding Properties

Based on this scheme we have developed a framework for designing, implementing and composing the defined properties. Its structure is the following:

On one hand there is the communication basic model, that can be used by any component. Apart from sending and receiving messages, the model incorporates the possibility of attaching controllers to mailboxes.

To add a property to a given component it is enough to specialize its controller and attach it to the component mailbox. The framework handles the controllers' composition and all message passing among them.

Every controller specifies the messages it deals with, and implements the three methods above according to its strategy. Some of its classes may be abstract (in Java's terminology), and specialization is achieved through inheritance and parametrization.

¹We have used Java [3] to implement the first version of our architecture.

As an example, controllers of the property of Independence manage a weighed list of known solutions (pairs service/provider). Although the list and its operations are fully defined in the framework, the component's preferences (i.e. weights) and its known solutions must be configured when initially instantiating the controller.

5 Implementing the Properties

The extension of this paper does not allow the detailed description of the properties defined in section 3. However, we shall briefly mention here the underlying ideas supporting the implementation of each one.

Independence: The controllers of this property incorporate three mechanisms: *i*) a weighed list of known solutions; *ii*) broadcast queries about available services in the system; and *iii*) the use of publicity to advertise component's new services or changes in existing ones. These controllers behave as "service brokers" or "traders", but with additional intelligence: they know their component's preferences (in the form of weight functions) and use them to decide the services to contract in each situation.

Adaptability: The problem of adaptability is an integral part of open systems, and can be decomposed into two problems: interoperability and extensibility. The controllers of this property behave as "dynamic adapters", able not only to adapt their components to different interfaces, but to search in the system for other adapters to new interfaces.

Self-Protection: The controllers of this property are in charge of watching that the component never waits for a event that will never happen. To achieve this task, they have the information about how long the component can wait for each message, and manage all pending events.

Integrity: To implement this property, the controllers manage a list of valid conditions for the incoming and outgoing messages. They do all the checkings and also the calling to the rejection procedures in case of invalid entries.

Secure Access: All accesses to and from the component are checked and authorized by the controllers according to the laws they know. Apart from the laws that the controllers handle, they can also search for "local advise" in certain cases; this mechanism allows the implementation of the local and adaptive nature of some laws, since they may vary from system to system. The controllers may be also in charge of adding security mechanisms (signatures, encryption).

High Availability: This property is achieved by the clonation of the component (one or more times), and the monitoring of the availability and activity of the component all the time. The controller may be also in charge of saving/restoring the component's state when these mechanisms are available.

Best Effort/Least Losses: In case the component offers several method bodies to deal with a message, the controller should decide the body to call in order to minimize the associated cost and to optimize both the response time and the quality of the service.

Durability: The controller should allow the component's maintainability and updating processes, diverting all component's events to its designated substitute during these operations.

6 Related Work

The idea that originated this work was born from the original papers from Tokoro and Takashio [22], and from Minsky and Leichter [16]. The former one mention real time, asynchrony and autonomy as key issues in open and distributed systems, but only deal with the first two concepts in their paper, without paying attention to autonomy. On the other hand, the law-governed architectures are introduced in [16], a one-property scheme that we have extended and generalized to give birth to our general model.

The use of a Component Framework as a "whitebox component that reveals part of its internal configuration, establishes configuration rules, and may enforce some of these rules" [21] fits like a glove to our proposed model. It is not the typical framework for the development of open and distributed systems and applications, as in the case of Hector [2], ASX [19], Java Beans [15] for stationary systems, or Aglets [14] for mobile ones. Our framework goes in a different direction: it provides the support to our methodology for designing components.

Regarding the properties, there exist a lot of literature about them in the field of Artificial Intelligence, where heated discussions take place about the precise definition of autonomous agents and their properties [9]. However, apart from various informal definitions we do not know of any work in that or any other field that formally defines them.

Neither the layered model or the communication mechanism based on mailboxes try to be a novelty, but to be functional, general and simple enough to be implemented without major problems in other systems. Layered models are well known and widely used: apart from the "Home Processes" for Linda [16] and the time control for objects [22] in the two aforementioned papers, we can also cite the time control for actors [18], the Composition Filter model [4], the layered Object Model LayOM [5], COM's aggregation or the usage of message handler procedures in the Oberon-2 [17] system. It is important to mention that our layers are not mere filters with delay capabilities at most, but more active, able to interrogate the system's components and to take decisions based on the component's preferences, more in the style of software adapters [10]. The model of processes that capture and modify the communication events of the components is completely reflective, as meta-actors [1] or filters [4] are. Finally, the only special concept used in the communication mechanism is the broadcast facility for domains, in the line of the ideas of Gehani [11] and ActorSpaces [1].

7 Conclusions

COP aims at producing software components for a component market and for late composition. While most of the efforts are now put on the development of better systems that facilitate the design and composition of components, we have focused on the components themselves and presented here a component framework that supports a design methodology based on the late addition of properties to components. Components are then designed to achieve only their core functionality, leaving the rest of the concerns

to the add-on properties that will be incorporated at a later stage if needed. This not only simplifies their design but makes them more reusable.

A set of basic properties has been defined, each one dealing with a specific problem in open systems. Besides, a general model for defining and specifying properties has been introduced. Having such a framework not only offers a platform to formally define components' properties, but it also allows to reason about them.

Currently the properties mentioned here have been specified and a prototype of the complete framework in Java with their main strategies is almost ready. We have chosen Java and the Internet as its first platform because of its wide use, the facilities it offers, and for being a good example of a real open system in which to test our model.

Acknowledgements

We would like to thank the anonymous referee for his careful and valuable comments and the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) for partially funding this work through grant TIC94-0930-C02-01.

Bibliography

- [1] G. Agha, W. Kim and R. Panwar. *Actor Languages for Specification of Parallel Computations*. In DIMACS, 1994.
- [2] D. Arnold et al. *Hector: Distributed Objects in Python*. Proceedings of the Fourth International Python Conference, Livermore CA, 1996.
- [3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley Longman, 1996.
- [4] L. Bergmans and M. Aksit. *Composing Synchronization and Real-Time Constraints*. In JPDC, No. 36, pp.32–52, Academic Press Inc., 1996.
- [5] J. Bosch. *Language Support for Component Communication in LayOM*. Presented at the ECOOP'96 Workshop on Component-Oriented Programming (WCOP'96), 1996.
- [6] N. Carriero and D. Gelernter. *Data Parallelism and Linda*. In LNCS, No. 757, pp. 145–159, Springer-Verlag, 1992.
- [7] K.M. Chandy and A. Rifkin. *Systematic Composition of Objects in Distributed Internet Applications: Processes and Sessions*. Proceedings of the 30th HICSS, Hawaii, 1997.
- [8] K.M. Chandy et al. *A Framework for Structured Distributed Object Computing*. California Institute of Technology. Submitted to Comms. of the ACM, 1997.
- [9] S. Franklin and A. Graesser. *It is an Agent, or just a Program?: A Taxonomy for Autonomous Agents*. Proc. of the 3rd International WS on Agent Theories, Architectures and Languages, Springer-Verlag, 1996.
- [10] E. Gamma et al. *Design Patterns*. Addison-Wesley, 1995.

- [11] N. Gehani. *Broadcasting Sequential Processes (BSP)*. In *Concurrent Programming*, Gehani and McGettrick (eds.), Addison-Wesley, 1988.
- [12] Y. Hoffner. *A Designer's Introduction to Trading*. APM Technical Report APM1387.01, Architecture Projects Management, APM Ltd., Cambridge, UK, 1994.
- [13] ISO/IEC 10746–1 to 10746–4. *Reference Model of Open Distributed Processing*. Draft Rec. X.901 to X.904 ISO/IEC JTC1/SC21/WG7, May 1995.
- [14] D.B. Lange and M. Oshima. *Programming Mobile Agents in Java – With the Java Aglet API*. IBM Research, 1997.
- [15] Sun Microsystems, Inc. *Java™ Beans: A Component Architecture for Java*. <http://splash.javasoft.com/beans/WhitePaper.html>, 1996.
- [16] N. Minsky and J. Leichter. *Law-Governed Linda as a Coordination Model*. In LNCS, No. 924, pp.125–146, Springer-Verlag, 1995.
- [17] H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer, 1995.
- [18] S. Ren, G. Agha and M. Saito. *A Modular Approach for Programming Distributed Real-Time Systems*. In JPDC, special issue on Object-Oriented Real-Time Systems, 1996.
- [19] D.C. Schmidt. *ASX: An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Systems*. BSC/IEEE Distributed Systems Engineering Journal, 1995.
- [20] C. Szyperski. *Independently Extensible Systems —Software Engineering Potential and Challenges—*. Proceedings of the 19th Australasian Computer Science Conference, Melbourne, 1996.
- [21] W. Weck. *Independently Extensible Component Frameworks*. In Special Issues in Object-Oriented Programming –Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP'96 in Linz. Max Muehlhaeuser (ed.). Dpunkt Verlag, Heidelberg, Germany, 1997
- [22] M. Tokoro and K. Takashio. *Toward languages and Formal Systems for Distributed Computing*. In LNCS, No. 791, pp.93–110, Springer-Verlag, 1993.

Inheritance Using Contracts & Object Composition

Wolfgang Weck

Turku Centre for Computer Science (TUCS) & Åbo Akademi
Turku, Finland
Wolfgang.Weck@abo.fi

Normal class-based code inheritance across component boundaries creates a dependency between the involved components. To avoid this, a specification of the inherited class must be part of the respective component's contract and the inheriting class must be specified with reference to this specification only. With this, inheritance can be replaced by object composition without sacrificing the possibility of static analysis, yet being more flexible.

1 Introduction

One of the distinguishing properties of component-oriented programming is the notion of *late composition*. This is to say, that component manufacturing and component composition are two separate steps, carried out one after the other. During component manufacturing, other components are referred to by interfaces, or contracts, only. Actual implementations are selected at composition time [12].

Object-oriented programming is a foundation technology for components. A typical component will specify a couple of classes or objects. To access services, other components will obtain objects from the providing component and send requests to them. In a running system, the hierarchy of components is complemented by a mesh of objects. These objects and the references between them are constructed, changed, and destructed at run time.

Object reuse and modification is a key tool for component reuse. In this paper we investigate language support for inheritance across component boundaries under the aspect of late composition.

We do, however, not discuss the semantical problems of inheritance, such as the fragile base class problem. We are interested solely in technical support of late composition. The semantical problems of inheritance can be treated separately, since our proposals apply also to various kinds of disciplined inheritance. In the extreme, the compiler may restrict the power of inheritance to that of forwarding.

2 Object Composition versus Class Composition

Two notions of classes and inheritance exist in the object-oriented programming community. Many programming languages and their underlying models are *class-based* (e.g. Eiffel, C++, Java, or Smalltalk). Others, such as Cecil [5] or Self [16] are *prototype-based*.

Class-based approaches abstract from the many instances of objects by grouping them into classes. All the objects of one class have the same attributes, accept the same messages, and exhibit the same behavior. Every new object is created as an instance of some specified class, and it will remain an object of this class throughout its life time.

With prototype-based approaches, objects are created by cloning an existing object, the prototype, and modifying the clone. Here, classes are sometimes seen as a dynamic equivalence relation that can be inferred at run time. By modifying an object's state, or structure, objects can be migrated from one class to another at run time. This approach has the advantage of being more flexible. It allows to change an object's behaviour or to assign class membership via predicates on the state [4].

The flip side of this flexibility is that static checking and reasoning becomes almost impossible. It may not even be clear, which messages a given object accepts, unless its complete history is examined. In a modular environment, this makes static analysis very difficult. Still, in a closed system, complete flow analysis may allow to make up for this [1], but in an extensible system this is not possible anymore [15].

With object-oriented programming, it is common to express composition and reuse by means of *inheritance*. In short, some inheriting entity inherits from one or several inherited entities by copying their implementation. Additionally, the inheriting entity may specify some modifications of the copied material. In principle, inheritance is equivalent to copying and modifying source code.

The above two views on object-orientation use inheritance between different kinds of entities. Class-based approaches support inheritance between classes, whereas prototype-based approaches support inheritance between objects. The latter is, for instance in Self [16], implemented through reference to a *parent object*, to which the handling of unknown messages is delegated.

Class-based approaches fix the inheritance relations at compile time. Since inheritance relates to implementation, class-based inheritance fixes the implementation to be inherited at compile time, i.e. before composition time in a component-oriented context. This makes state-of-the-art class-based inheritance unsuitable across components, because we want to delay the selection and binding of the base-class implementation until composition time.

We can conclude that, depending on the school you follow, you will get from object-oriented programming either support of static analysis or the possibility to compose implementations later than at compile time, but not both. A question of interest is, whether a middle ground can be found, on which you get both static analysis and late composition of implementations. Such a middle ground would be necessary for component-oriented programming to allow for inheritance across components.

3 Contracts

On the component level, the above dilemma between static analysis and late composition is well known. There, the answer is the definition of contracts, which specify the obligations component providers must meet and the expectations component clients may have. For every component, it must be documented according to which contracts it offers or requires services. Two components can be composed, if one offers services that are requested by the other component according to the same contract. Each component can be analysed separately, based on the contracts it participates in. At composition time, one only needs to check whether the two components actually claim to stick to the same contract. If they don't, the composition can be rejected.

In short: at compile time only specifications are bound, whereas implementations are bound at composition time. The contract provides the necessary separation of the specification from the implementation.

The practical effect of this separation is that static reasoning is still possible, because the yet unknown partner can be substituted by the contract. Still, bindings between implementations are established only at composition time, retaining full flexibility of selecting components to the composer. Thus, with components, we managed to eat the cake and have it too.

4 Class Composition With Contracts

The same technique can be used with inheritance. Instead of referring to an implementation, the inheriting class refers to a contract only. The contract states what to expect from the inherited class; the inheriting class can be statically analysed. We get the safety we want.

Only when an object is instantiated, the binding to a concrete base class implementation must be established. Any class meeting the required contract can be bound. Like with component composition, the contract can be used to check at composition time, whether the specific composition is feasible. In addition to safety, we get the late composition we want.

Note that it is because of the separation between specification and implementation that the semantical problems of inheritance become so acute with component-oriented programming. The inheriting class must simply be able to cooperate with *any* base class that meets the specification. In this paper, however, we postulate specifications to be detailed enough and/or one of the many approaches to disciplined inheritance to be in place.

Class inheritance with contracts is implemented in IBM's SOM and in modular object-oriented programming languages, such as Modular Smalltalk [19] or Oberon-2 [9], an extended version of Oberon [18]. These languages feature separate constructs for modules and classes. Modules are separately compilable, similar to Modula-2. It is possible to compile several modules implementing the same interface. This allows for alternative implementations of the same specification.

In Oberon-2, for instance, classes are implemented as record types. The latter are extensible inside as well as outside the module in which they are defined. Furthermore, procedures can be bound to such record types. Such type-bound procedures resemble methods. In an extending type, they can be redefined, otherwise they are inherited from the base type.

Exported record types resemble contracts for classes, because module interfaces contain only link information to be exploited at binding time. To avoid confusion, note that the contract syntactically consists only of signatures. However, we postulate some semantics being attached, e.g. as a comment, i.e. we use types in the spirit of behavioural types [8]. This semantics specification just happens not to be checked by the compiler.

New classes can be specified, programmed, and compiled referring to a base type. This happens whenever a record type is extended in a separate module, because compilation relies on the imported module's interface. The implementation is bound at load time only and therefore needs not to be selected earlier.

With modular object-oriented programming, only one component (module) per contract can be used in a running system. As a consequence, all classes meeting a

given contract have the same implementation. It is just that this implementation is selected very late.

Still, this has been used effectively for system refactoring, done one module at a time, as long as the module's interface had not to be changed. In some cases of modules implementing device drivers, alternate implementations of a single contract were provided to allow adoption to different hardware. (These modules seldomly actually define classes, but they could do so.)

5 Can We Go Further?

Can we get rid of the aforementioned restriction and support different implementations of a contract simultaneously? One could allow several modules implementing the same interface to coexist in a running system. This would collide with some assumptions being generally made about modules. Also, we would need to find a way to refer to the different module implementations. Currently, identifications are made by referring to the module name, i.e., the contract, which is implemented by exactly one module.

Another, probably better, approach could be based on separating subclassing from subtyping, as for instance done in Sather [11]. Types would be employed as contracts whereas classes would be bound as implementations later. At component manufacturing time, subclasses would be specified by referring to a type instead of to a class. (Sather does not support this to avoid the semantical problems of inheritance. It rather defines class inheritance to be exactly equivalent to textual inclusion and application of text editing operators. Obviously, this binds an implementation at compile time.)

With this, the base classes to be used with an object would need to be specified at object allocation time; the allocation procedure would need to accept the respective additional parameters.

To make this useful, we would have to turn classes into first order objects. Otherwise, the programmer of a component that contains a creation statement would have to wire in which implementation to use. This would again create a dependency between components, this time between those containing the creation statement and implementing the base class.

6 Object Composition With Contracts

This can be taken one step further by composing objects instead of classes: one can specify a base object instead of a base class when allocating an object.

If a contract is used to specify statically the properties required from the parent object, and if the parent object is bound for ever when the referring object is created, static analysis is possible to exactly the same degree as before, i.e. as with class-based inheritance with contracts. Though each object may be of a class of its own now, the same information as before is available from the object itself and the contract specifying the parent.

To retain the full amount of static information as with class inheritance, we must prohibit to re-assign the parent or base object. Otherwise, unexpected changes of state and/or behaviour of the composed object could occur.

This construction is indeed on a middle ground between static class inheritance and full dynamic inheritance as used with prototype-based object-orientation. Compared

to the former, we gain flexibility, even more than with class composition based on contracts. Compared to prototype-based object-orientation, we get more static information because of two restrictions. First, only objects satisfying the required specification can be used as a parent. Second, the parent, once assigned, cannot be changed anymore.

Still, we get much of the flexibility of prototype-based object-orientation. For instance, the user can interactively specify a parent object to be wrapped. An example are wrappers for editors that add some functionality. The user can compose these graphically. For instance, a normal text editor can be extended to send notifications about text changes to other users. Note that this wrapper would work with *any* text editor that implements the required contract. Other applications of such wrappers can be found within the BlackBox component framework [10].

We take it as strong support of our proposal that it implements the formal model used by Cook and Palsberg to describe inheritance [6]. There, inheriting classes are specified as *wrappers* that only refer to the base class' signature. A concrete base class is bound at instantiation time only.

An implementation inbetween our proposal and Self was proposed as "Delegation Through a Pointer" to be added to C++ [14]. Compared to Self, the pointer to the parent object is typed. Viewing types as approximations for specifications again, we see that one of our two restrictions applies. The parent has to meet a certain specification. In contrast to our's, Stroustrup's proposal would still allow to re-assign the parent object.

An implementation that fixes the parent object can be found in Modula-3 [3]. Its allocation procedure allows to specify a list of methods to be bound to the new object. However, the base object's class (i.e. its implementation) is fixed statically by the type of the variable passed to *NEW*. Thus, Modula-3 does not allow a dynamic selection of the parent object at run time. In this sense, it is less flexible than our construction.

In the rest of this section we sketch a simple implementation for proof of concept. More efficient implementations may be possible, in particular to shortcut in deeply nested compositions. One way to approach this problem can be found in Microsoft's COM aggregation.

The wrapping (inheriting) object can refer to the parent (inherited) object. Method calls, not handled by the wrapper, are forwarded to the parent. Cecil [5] translates an inheritance-like syntax to such object composition. To achieve the same kind of self-recursiveness as with inheritance, this scheme can be enhanced to delegation by passing an extra self parameter as shown in [7]. [13] shows that delegation is as powerful as inheritance. With our proposal, the extra parameter needs not to be visible to the programmer. The compiler would generate what elsewhere the programmer would need to do explicitly.

A contract may specify how to access instance variables of a base class. The compiler would translate such access to superclass variables to dereferential access of the parent object's variables. Further, the compiler could easily enforce certain restrictions, e.g. granting access to subclasses but not to clients.

The benefit of compiling inheritance into delegation is that objects can be composed instead of classes without losing the possibility of static analysis. For the latter, the compiler asserts that the reference to the parent object cannot be changed after creation.

It would further be possible, that the compiler enforces some kind of restricted inheritance to avoid the semantic problems caused by inheriting an invisible implementation (see above). In the extreme, the run time data structure could support plain forwarding only, but the programmer can still use the more convenient inheritance notation. Such an approach is also attractive for a programming language to be compiled to Microsoft COM's aggregation.

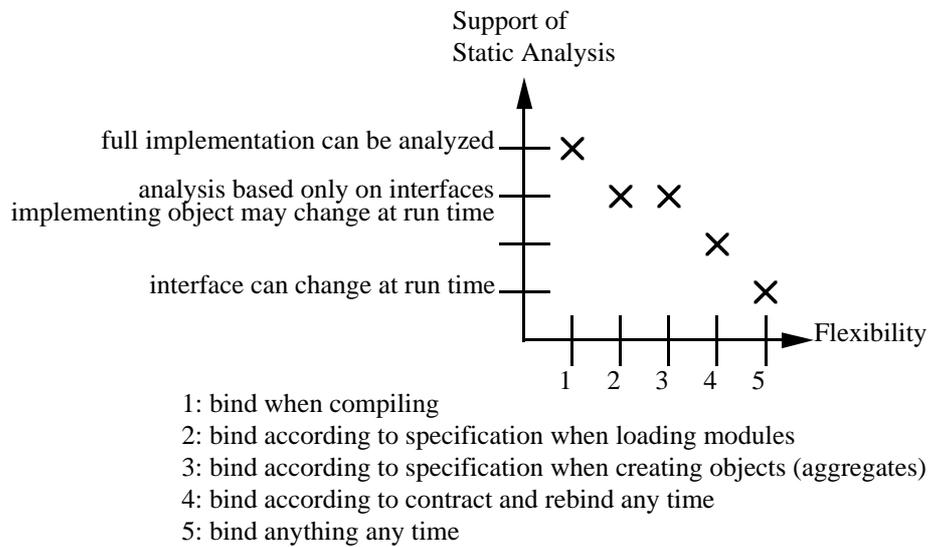


Figure 1: Static Analysis versus Flexibility

7 Summary

To be applicable as a foundation technology for component-oriented programming, object-oriented programming with inheritance must support a middle ground between class-based and prototype-based object-orientation. Traditional class-based object-orientation is not flexible enough, unless class specifications rather than implementations are bound at component manufacturing time. On the other hand, prototype-based object-orientation is too flexible, thereby prohibiting effective static analysis.

As the above middle ground we suggest syntactical support for inheritance but using only a specification of the base class together with an implementation as object composition "under the hood". The compiler would have to hide the details of the latter to assert that the objects are not composed arbitrarily. In particular, the parent object must match the specification. Also, the compiler would have to prohibit that the parent object is re-assigned, once the composed object has been created.

This scheme allows for full static analysis, limited only by the amount of information stated with the specification of the inherited object. It gives the best possible flexibility, since the selection of the inherited code is delayed not only until component composition time, but until object generation time. The latter allows even the user to pick the code to be bound (see Fig. 1).

Our proposal allows both to compose objects using delegation, somehow restricted delegation, or plain forwarding. Delegation has the same power as class-inheritance. Depending on the future solutions to the semantical problems of inheritance and encapsulation, restrictions up to plain forwarding may become appropriate. Our suggestion can be adopted as needed.

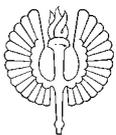
Bibliography

- [1] Agesen, O., Palsberg, J., Schwartzbach, M.I.: Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In O.M.Nierstrasz (ed.), Proc. ECOOP'93, LNCS 707, Springer-Verlag Berlin, ISBN 3-540-57120-5, pp. 247-267.
- [2] Brockschmidt, K.: Inside OLE 2. Microsoft Press, ISBN 1-55615-618-9, 1994.
- [3] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 Report (revised). SRC Report 52, Digital Systems Research Center, Palo Alto, California, 1989.
- [4] Chambers, C.: Predicate Classes. In O.M.Nierstrasz (ed.), Proc. ECOOP'93, LNCS 707, Springer-Verlag Berlin, ISBN 3-540-57120-5, pp. 268-296.
- [5] Chambers, C.: The Cecil Language, Specification and Rationale, Version 2.1. Department of Computer Science and Engineering University of Washington, Seattle as available at April 25, 1997 from <http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html>.
- [6] Cook, W., Palsberg, J.: A Denotational Semantics of Inheritance and its Correctness. In Norman K. Meyrowitz (ed.), Proc. OOPSLA'89, SIGPLAN Notices 24:10.
- [7] Johnson, R.E., Zweig, J.M.: Delegation in C++. Journal of Object-Oriented Programming 4:3, November 1991.
- [8] Liskov, B., Wing, J.M.: A New Definition of the Subtype Relation. In O.M.Nierstrasz (ed.), Proc. ECOOP'93, LNCS 707, Springer-Verlag Berlin, ISBN 3-540-57120-5, pp. 118-141, 1993.
- [9] Mössenböck, H.: The Programming Language Oberon-2. Structured Programming 12:4, 1991.
- [10] The Oberon/F User's Guide. Oberon microsystems, Inc., Basel, CH, (<http://www.oberon.ch/customers/omi>), 1994.
- [11] Szyperski, C., Omohundro, S., Murer, S.: Engineering a Programming Language: The Type and Class System of Sather. In Proc. International Conference on Programming Languages and System Architectures, LNCS 782, March 1994.
- [12] Szyperski, C.A., Pfister, C.: Proc. first international Workshop on Component-Oriented Programming (WCOP'96). In M. Mühlhäuser (ed.), Special Issues in Object-Oriented Programming, dpunkt Verlag Heidelberg, ISBN 3-920993-67-5, 1997.
- [13] Stein, L.A.: Delegation is Inheritance. In Proc. OOPSLA'87, October 1987.
- [14] Stroustrup, B.: Multiple Inheritance for C++. In Proc. EUUG Spring Conference, May 1987.
- [15] Szyperski, C.: Independently Extensible Systems - Software Engineering Potential and Challenges. In Proc. 19th Australasian Computer Science Conference, Melbourne, Australia, 1996.

- [16] Ungar, D., Chamber, C., Chang, B.-W., Hölzle, U.: Organizing Programs Without Classes. In *Lisp and Symbolic Computation*, July 1991 4:3, Kluwer Academic Publishers, pp. 223-242, 1991.
- [17] Wirth, N., Gutknecht, J.: *Project Oberon. The Design of an Operating System and Compiler*. Addison-Wesley New York, ISBN 0-201-54428-8, 1992.
- [18] Wirth, N.: The Programming Language Oberon. *Software - Practice and Experience* 18:7, pp. 671-690, July 1988.
- [19] Wirfs-Brock, A., Wilkerson, B.: An Overview of Modular Smalltalk. In N.K.Meyrowitz (ed.), *Proc. OOPSLA'88, SIGPLAN Notices* 23:11, 1988.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

<http://www.tucs.abo.fi>



University of Turku
• Department of Mathematical Sciences



Åbo Akademi University
• Department of Computer Science
• Institute for Advanced Management Systems Research



Turku School of Economics and Business Administration
• Institute of Information Systems Science