Working Paper Series
ISSN 1170-487X

Proceedings of CBISE '98 CAiSE*98 Workshop on Component Based Information Systems Engineering

Edited by John Grundy

Working Paper 98/12
June 1998

Proceedings of

CBISE'98

CAiSE*98 Workshop on

Component Based Information Systems Engineering

June 8-9, Pisa, Italy

Edited by John Grundy

Contents

Foreword

Component-based information systems development is an area of research and practice of increasing importance. Information Systems developers have realised that traditional approaches to IS engineering produce monolithic, difficult to maintain, difficult to reuse systems. In contrast, the use of software components, which embody data, functionality and well-specified and understood interfaces, makes interoperable, distributed and highly reusable IS components feasible. Component-based approaches to IS engineering can be used at strategic and organisational levels, to model business processes and whole IS architectures, in development methods which utilise component-based models during analysis and design, and in system implementation. Reusable components can allow end users to compose and configure their own Information Systems, possibly from a range of suppliers, and to more tightly couple their organisational workflows with there is support.

This workshop proceedings contains a range of papers addressing one or more of the above issues relating to the use of component models for IS development. All of these papers were refereed by at least two members of an international workshop committee comprising industry and academic researchers and users of component technologies. Strategic uses of components are addressed in the first three papers, while the following three address uses of components for systems design and workflow management. Systems development using components, and the provision of environments for component management are addressed in the following group of five papers. The last three papers in this proceedings address component management and analysis techniques.

All of these papers provide new insights into the many varied uses of component technology for IS engineering. I hope you find them as interesting and useful as I have when collating this proceedings and organising the workshop.

John Grundy, June 1998

Workshop Committee

COMPONENT-BASED ARCHITECTURE: A Call for Pragmatism

Paul Allen
VP Methods
Select Software Tools

June 1998
CBISE 98 Pisa, Italy

Introduction

Businesses are becoming increasingly adaptive. This results in requirements that change rapidly, calling for software that is more and more flexible. Organizations continue to struggle with the increasing scale and complexity of software development and the practicalities of integrating their legacy systems with newer technologies. Component-based development is increasingly presented as the latest "cure" for these problems but itself depends on good design. Good design calls for effective architecture. One of the problems with developing an architecture is the number of different types of possible architecture: logical, physical, data, business, network, and so on. In a large organization this problem can be daunting: by the time you are half-way done the business and technology have changed so much that you have to start over. This paper recognizes that there is no "perfect" architecture and considers how to approach this problem in a pragmatic way.

The Need for Pragmatism

In the late nineteenth century a philosophy called "Pragmatism" developed in America (James, 1943) as a revolt against what some thinkers felt to be a sterile philosophical tradition then flourishing in Europe. Pragmatism is essentially a method for solving or evaluating intellectual problems and a theory about the kinds of knowledge we are capable of acquiring. It challenges purely theoretical activity by asking bluntly, "what's the point?". Before determining whether a philosophical claim is true, pragmatism takes the view that we must examine the **cash-value** of the claim. Our intellectual activity has as its purpose the attempt to resolve difficulties that arise in the course of our attempts to deal with experience. The cash-value of ideas is to be found in the use to which ideas can be put to work.

For example if I was walking in the woods and lost my way a solution might be to take into account such data as the sun's position, direction of my walk, my previous knowledge of the terrain. The cash-value of this theory is whether it helps me find a way out. In contrast, other philosophical theories have no cash-value. For example, what difference would it make if I believed the universe was really one vast mind, or if one believed such a theory false?

"So what?" you're probably asking. Well, in today's world of methods it's interesting to draw an analogy with the above developments in philosophy. With so much happening on the methods front we need to step back and ask whether the method is useful in a practical business setting, or does it result in a collection of academically impressive diagrams with no practical use. "A picture is worth a 1000 words" we are often told by the methodologists. However what if the picture is an illegible mess, what if the picture itself has a thousand words scrawled all over it? In a nutshell we must ask, in the spirit of pragmatism, "What's the cash-value of the method?". This means a new focus not so much on software as on business benefit and on an applications architecture that covers both software and business process.

What Are Components?

Currently there is much general confusion over the term "component". So before considering the pragmatics of architecture it is important to be clear on definitions.

A *component* is an executable unit of code that provides physical black-box encapsulation of related services. Its services can only be accessed through a consistent, published interface that includes an interaction standard. A component must be capable of being connected to other components (through a communications interface) to form a larger group.

A *service* is a type of operation, that has a published specification of interface and behavior, involving a contract between the provider of the capability and the potential consumers. Using the service description, an arms length deal can be struck that allows the consumer to access the capability.

A major problem with using objects as the building blocks of an application is that implementation dependencies are often exposed through OO programming language interfaces. Interaction standards, such as the OMG's CORBA and Microsoft's DCOM, help to resolve this issue by providing clean separation between specification and implementation(s) in terms of component interfaces. A third layer (or "form") of executable binary code completes a very powerful and flexible structure. Note that, significantly, the component code can be changed without disturbing its clients, provided the interfaces are kept consistent. This is particularly important for legacy software.

A component-based approach has the potential to remove the tight coupling of an application's parts and ease reuse of parts across different applications. The great is that application "islands" disappear: objects are packaged as components, which are configured in network fashion to meet business needs. However, giving developers *carte blanche* with such powerful technology magnifies the dangers of hacking all the application logic to the GUI. When this happens, business logic becomes fragmented across countless different applications, and there is an attendant loss of control of the business policy implemented by the software. Any opportunity for leveraging reusable business components is lost.

Pragmatic Architecture

An effective architecture must provide an overall structure and set of rules for managing the scale and complexity inherent in enterprise software development. It must help achieve software interoperability, adaptability and consistency of design. This means that it increases the chances of achieving a good design; it is not a substitute for good design!

A service-based approach makes use of the established software engineering principle of separating user interfaces, business logic and data management. Service categories bring this principle a step forward as follows:

- **User services** provide human-computer interfaces to meet the needs of a particular business process or department. User services link together with business services to deliver the business capabilities to users in terms of a visual interface.

- **Business services** convert data received from data services and user services into information by coupling related business tasks with the relevant business rules. Business services are technically neutral and may apply at different levels of generality across several business processes or departments.

- **Data services** are used for the manipulation of data in a way that is independent of the underlying physical storage implementation. Data services commonly apply to generic data access requirements across several business processes or departments.

Business components are abstracted out into their own layer ensuring the implementation independence that is so important for a component-based approach. This architecture also helps close the, all too prevalent, gap between business process improvement and software engineering: A user service delivers business capability through the software interface to a business process.

There are two further ground-rules that are followed in layering services:

- aim for increasing generality downward through each layer; for example a core business component like Product Rules provides services to a specific business component like Product Sale.
- take advantage of the best work of others. There are several important industry initiatives in this area, including OMG work through the business object task force (BOTF) as well as useful work on business related patterns (Fowler, 1997), not only design patterns which are now well-covered in the literature (Gamma et al 1995).

It is important to understand that the service-based architecture is not a panacea. Developing software is still hard! However at least we have a start: the service-based architecture provides an overall context for shaping modeling techniques as we seek to apply good design principles.

Modeling Software Architecture

UML package diagrams (OMG, 1997) are used for modeling software architecture as illustrated in figure 1. A ***service package*** is a user-defined stereotype of package that provides a set of services belonging to a single service category. Service packages provide a mechanism for grouping objects into cohesive units and achieving an effective granularity of reuse. The service packages are also used to effectively wrap legacy assets. One of the major benefits of this is that proposed reuse of legacy systems, software packages and databases is addressed as part of architectural design, and not left as an afterthought, as in many methods; for example the Product service package might represent a software package.

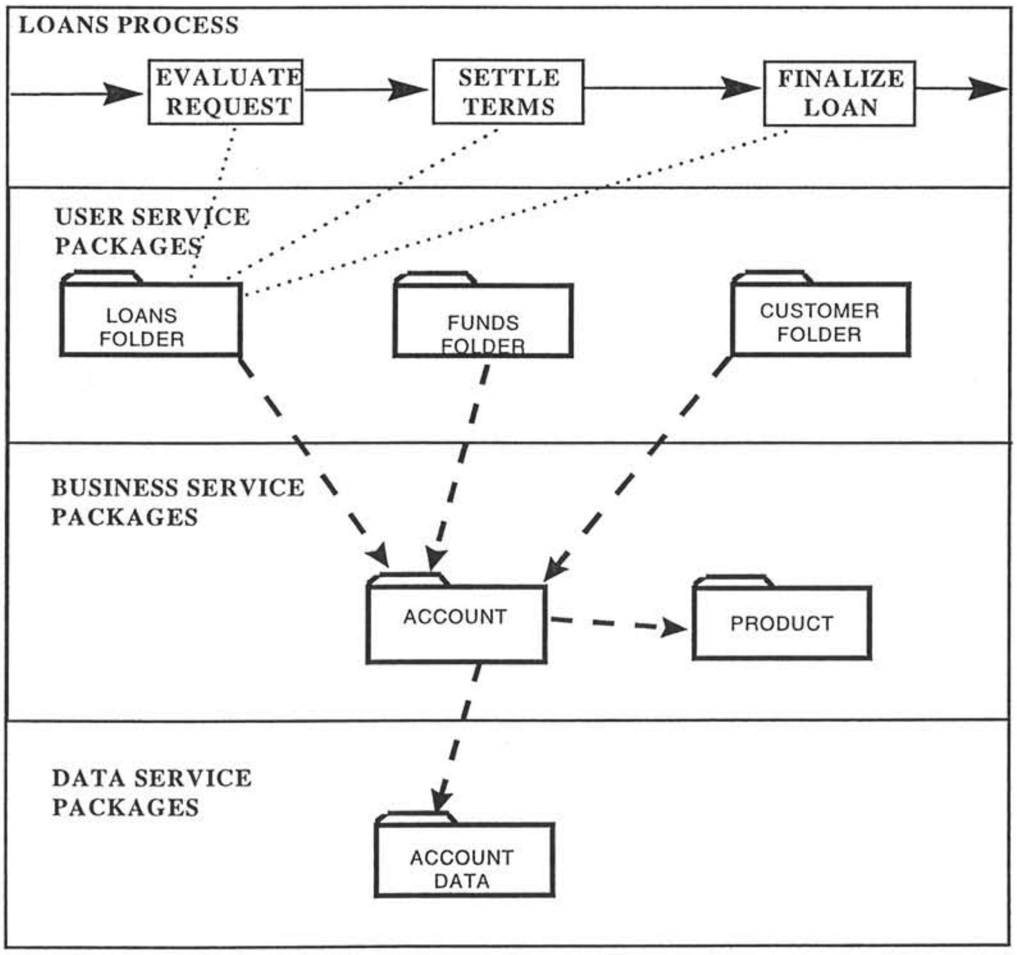


Figure 1. Part of a Service-Based Architecture for a Finance House.

Applying Established Good Practice

Consider for a moment the proposed solution to one of the most pressing problems in civil engineering - how to stop the Leaning Tower of Pisa collapsing. Experts are planning what they initially regarded as a pioneering operation to siphon soil from under its raised north side so that it will slip back into a more vertical position after computer models showed it to be on the verge of falling over. But a professor researching a history of civil engineering has discovered that the technique was first used by James Trubshaw, a British engineer working more than 150 years ago, to save a tilting church tower in Cheshire, UK.

Similarly the service-based architecture leverages investment in established techniques. For example, three stereotypes (OMG, 1997) of class are used; user, business, and data. User and data classes are technology classes. Another established idea: changes to user interfaces—which are quite common given ongoing developments in graphical user interface (GUI) technology—will not affect the underlying business classes. The data classes provide data services by controlling access to database storage and converting this data to a clean, non-database specific interface for use by business and user services. Again an established idea: this protects the business classes from the effects of changes to databases.

The service package provides the required services through one or more ***service classes***. A service class is a type of class, providing one or more services, which represents an abstraction of a component interface(s). A service class is used to help retain a domain focus in modeling without becoming embroiled in the potential complications of implementation: it gives the developer a mechanism for identifying interfaces "early in the process" in preparation for mapping to components. A service class marshals service requests and encapsulates access to a component in the spirit of responsibility-based design by contract (Wirfs-Brock, 1991). Typically, it is a control class (Jacobson, 1992) exclusively designed to provide services through one or more interfaces, although services can be allocated to any appropriate class, which then has the role of service class. The fact that a class is a service class is derived from the fact that at least one of its operations is a service.

Note how the architecture helps to shape and enhance established modeling techniques, rather than introducing radically new notations. This is very important for organizations who wish to protect their investment in existing skills.

Addressing the Legacy Issue

The service-based architecture also helps in leveraging existing software: Instead of rethinking the problem through afresh in terms of what software can be used to solve a business problem, sets of service features are examined to see which can be reused. As the bank of services is built up, so reuse of pre-existing code increases and the need to write new code decreases. Component code can be changed without disturbing its clients, provided the interfaces are kept consistent. This is particularly important for legacy software. For example, let's say one freight company acquiring another freight company needs to find a way to interface with the latter's billing system. The goal is to give the customer a "unified" bill so he or she would have all transport costs on a single invoice. Both billing systems are legacy systems operating on different hardware at different sites that are not practical to integrate. By wrapping each billing system in terms of the services it provides, a new "solution" that accesses the required services can rapidly be put together.

Another project involved a dealing system in which various *user services* provided visible functionality in the form of sets of window-based GUI forms. User services included placing deals, authorizing deals, setting up schemes, removing schemes and so on. A set of reusable *business services* supported the user services in their function. Business services included business tasks such as checking ability to proceed with a deal and verifying security types. The business services made use of a number of business rules, such as rules for determining a customer's credit rating or whether there were sufficient funds to cover pending deals. The business services used a set of reusable *data services* to access the necessary client and security data from various legacy databases transparent to the user.

Architecture Planning

Architecture planning employs a range of techniques (Allen and Frost, 1998), including scoping of business processes and analysis of domains, dependencies, patterns and frameworks, and legacy systems. The key activity is to identify service packages with minimal dependencies and clean, reusable interfaces. We follow the convention that a package is by default a service package if no stereotype symbol is shown. Service categories are used to guide the layering of service packages.

All layers are usefully partitioned by increasing generality, aiming for stable and responsible classes at the lower levels (Martin, 1995). Service packages at the top of the business layer often correspond to business processes (for example Account may correspond to an Account Management process) but may be used generically across different user service packages.

It is also useful to study similar work by others and to keep pattern-aware. Figure 2 illustrates a prevalent pattern observed in several industries - holidays, mobile phones and insurance - that might be applied to further develop the business layer in the previous example.

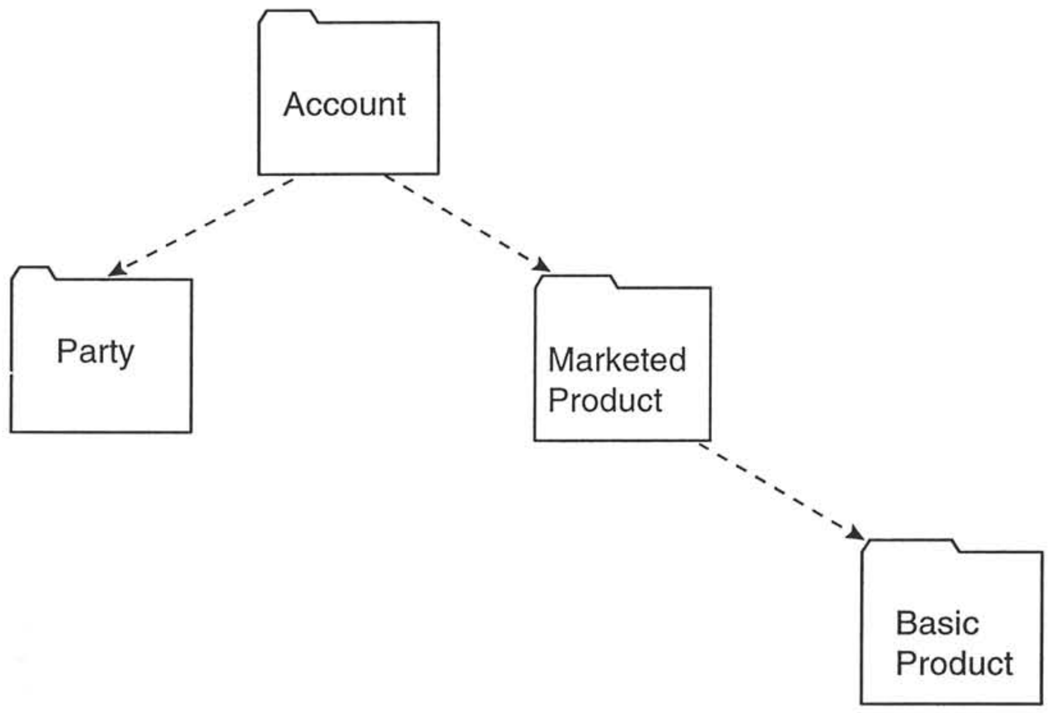


Figure 2. Refinement of the Business Layer using Patterns.

In parallel, we also approach bottom-up by assessing existing software assets. In particular, if reuse of a particular legacy database or software package is mandated up-front, then it is important to declare this within the overall architecture.

Typically, service packages are allocated to different teams. Project management is facilitated by architecting service packages in early phases of development. This also has the advantage that incremental design can focus on specific implementation details without being overloaded with wider architectural concerns.

Shaping the Architecture

No architecture is perfect - making trade-offs is a necessary part of the process. The price paid for a pure service-based architecture can be increased numbers of calls across the layers. This problem is exacerbated in the case of a highly constrained implementation, where services are fragmented across several platforms. These issues need to be balanced against the flexibility, maintainability and reuse that comes with a service-based approach. It is important to understand that service layers guide the partitioning but are sometimes sacrificed in the interests of non-functional requirements such as performance or security. The main theme is to partition to minimize undesirable dependencies and to critically appraise associations between classes in different service packages. It is also important to keep pattern aware, as discussed earlier.

Other trade-offs occur at the service-layer boundaries. For example if there is certain business policy that applies locally at different user sites to Account then it may well be expedient to hold the local business logic in the user service layer, especially if faced with a distributed implementation.

Regardless of implementation constraints it is a good strategy to separate out common interfaces within the business service layer. The reason is twofold: technical neutrality and reuse. It is important to model actors as roles, both in business process modeling and use case modeling. In the example there may be several different physical users of account information: the Loans Manager, Funds Manager and Customer Service Representative. However, they all share the same role of Account Manager. This suggests modeling a single service class, Account Controller, on to the Account service package.

User service packages often correspond to particular individuals or departments. In the example the Loans Administrator and Funds Manager have different user interface requirements; the former needs a sophisticated GUI whereas the latter requires a simple menu-based interface. Therefore there are two folder service packages corresponding to these different needs. The Loans Folder is designed generically for use across all parts of the loans process.

Service packages are often "orthogonal" in the sense that it can be difficult to judge in which package a particular class resides. For example, does Account Payer belong in Account or Party or maybe a new service package of Payment? Such decisions depend on trading off generality against dependency. By putting Account Payer in Account the billing details are all in one place, but generality has been sacrificed; there may be duplication if the same person both pays for is also a sales contact or employee (and is therefore held within Party). Other guidelines that help in making such decisions include:

- avoiding circular dependencies
- minimal inter-associations between packaged classes
- maximum intra-associations between packaged classes
- aiming for cohesive packages, more by classes commonly reused together, rather than simply functionally cohesive
- achieving good granularity; for example, 5-15 classes per service package is a rough rule of thumb

Developing the Architecture

The main focus in this paper has been largely top-down and on raising awareness of some of the practical structural issues in developing a service-based architecture. However architecture does not happen in a vacuum. The pace of change, both commercial and technical, means that software solutions need to be delivered in timely fashion. Delivering effective software in parallel with developing an architecture is one of the major challenges we face today; above all else this calls for an effective software process, details of which are outside the scope of this paper but which is summarized in figure 3. The important point is that project-level development and architecture development work hand in hand. This requires trading-off short-term gain in terms of project development of software solutions that confer early business value, against the longer-term investment in architecture, as described in (Allen and Frost, 1998). Component management facilities are an important enabling technology for leveraging this approach; these are discussed in (Allen and Frost, 1997).

Software development projects follow an iterative incremental approach which is driven by use cases in support of business processes. The architecture is used to integrate services smoothly with business process modeling thus helping to close the, all too prevalent, gap between business process improvement and software engineering . Units of business process modeling are modeled using established event-driven techniques. These provide a natural pathway into use case modeling, though definitions must be applied carefully (Allen and Frost, 1998).

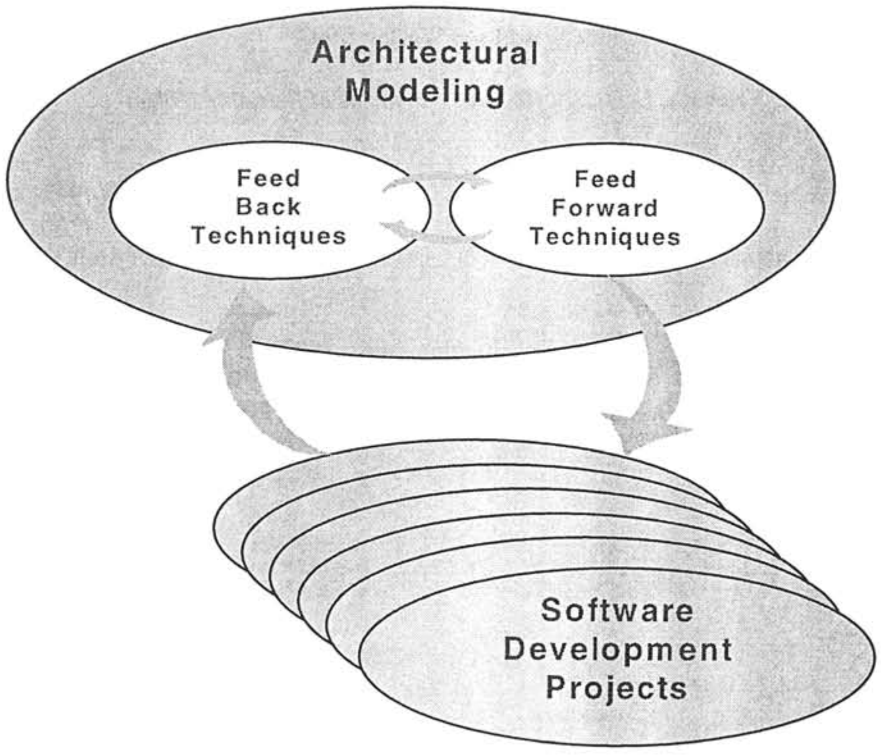


Figure 3. The Iteration Between Architecture and Software Projects.

The two approaches of business processes/use cases and service packages/domain modeling effectively come together by analyzing required object collaboration and interaction. It is here that feedback from development projects is effectively harnessed into the process. Use cases provide test cases for the architecture. Other well-tried techniques such as data modeling also have a key part to play in the renovation of legacy assets, particularly relational databases, to help get them in a fit state for wrapping.

As development unfolds so further UML diagrams are bought into play. In particular it is important to use deployment modeling (OMG, 1997) to explore distribution of service packages in terms of physical components across physical platforms to ensure the design meets detailed implementation constraints and quality criteria.

Summary

This paper has presented a pragmatic service-based architecture in which to leverage good design practice, patterns and established modeling techniques. The approach is grounded in business requirements. Domain modeling is used to drive the architecture. In the limited scope of this paper we touched on some basic principles including the concept of service layering, separation of operational from core functionality, the application of patterns and design guidelines. Use cases do not drive the development of the architecture but play an important part in driving system delivery at project level and in actually proving the architecture.

REFRENCES

Allen, P. and Frost,S., Component-Based Development for Enterprise Systems: Applying The SELECT Perspective, SIGS Publications, 1998.

Allen, P. and Frost,S., Component Manager, *Select Software Tools White Paper*, 1997.

Fowler, M., Analysis Patterns: Reusable Object Models, Addison Wesley, 1997.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

James, William, Pragmatism. A New Name For Some Old Ways of Thinking, Longmans, London 1943.

Martin, R.C., Designing Object-Oriented C++ Applications Using The Booch Method, Prentice Hall, 1995

OMG, *Unified Modeling Language Version 1.1,* OMG, Framlington, Mass.,1997.

Wirfs-Brock, R., Wilkerson, B. Wiener, L., *Designing Object Oriented Software*, Prentice Hall, 1991.

Why Model Software Products: Why, Guiding the CBISE Process, Of Course!

C. Ncube & N.A.M Maiden

Centre for HCI Design
City University, London
Tel: +44 171 477 8994
Fax: +44 171 477 8859
E-mail: C.Ncube@soi.city.ac.uk

Abstract

This paper proposes a software product model for guiding the Component-Based Information Systems Engineering (CBISE) development process, and in particular for guiding the requirement acquisition and product selection processes which must take place before system design and implementation. The product model is central to the PORE (Procurement-Oriented Requirements Engineering) method's requirements acquisition and product evaluation processes as it drives both processes. The process of requirements acquisition and product evaluation is iterative, part goal-driven and part context-driven, and exploits user-developed models of the candidate products or components as well as process goals to guide a requirements engineering team. The paper ends with a discussion of the current problems with the approach, and future developments to solve the problems.

Keywords

COTS software products, product model, requirements acquisition, product selection, process guidance, product-requirement compliance mapping, requirements model,

1: Introduction

As the next century approaches, the development of enterprise information systems is shifting away from what is known as bespoke systems development and moving rapidly to Component Based Information System Engineering (CBISE). Commercial-Off-The-Shelf (COTS) software products are now available to perform most of the functions that in the past required bespoke software. Procurement is the process of purchasing complex software systems from suppliers. These systems might be COTS systems or tailor-made for a customer. Requirements are a cornerstone for effective COTS software systems procurement. For example, requirements are used as a criteria for selecting the COTS system, are embedded in a legal contract and provide acceptance criteria to check when the developed system is delivered.

The use of COTS products introduces new problems for requirements engineers; e.g. deciding when to acquire new customer requirements and when to reduce the number of candidate products. However, this use of COTS products in CBISE has the potential for reducing the cost and time required to develop an information system. In today's competitive markets, the time-to-market or deployment is one of the most influential sources of competitive advantage open to organisations. The CBISE process enables organisations to maintain their competitive superiority. However, given the complexities of today's software intensive systems, the cost of procuring/purchasing wrong package due to inadequate requirements acquisition and product selection is large. The problem is that when building systems from COTS packages, new and different types of requirements need to be defined and new methods and techniques of acquiring these requirements and selecting candidate COTS products also need to be defined. We are developing a new method, PORE, which supports the requirements engineering and product evaluation/selection processes for CBISE development using an iterative process of requirements acquisition and product evaluation/selection.

One of the main features of PORE is that it encourages a requirements engineering team to acquire, model and analyse customer requirements at the same time as acquiring, modelling and analysing candidate software products. Although there are numerous requirements modelling languages available, there are very few approaches for modelling software products. The design of our method (PORE) identified three important problems for which there have not been any obvious solutions reported in the literature:

- how to interleave requirements modelling and software product modelling in order to provide effective process guidance?
- how can the software product model guide the requirements acquisition and product selection processes?
- what features of a software product must be modelled to guide these processes?

The software product model proposed in this paper attempts to solve these three problems and is the central component of the PORE's iterative process.

As mentioned above, one of the problems which arose during the design of the CBISE method (PORE), was what features of a software product must be modelled to guide the requirements acquisition and product evaluation/selection processes. PORE's solution is the software product model which represents the observable behaviour of the product and in particular, how the user interacts with it. To achieve this, we draw on the existing use case modelling approaches, and in particular the CREWS (Co-operative Requirements Engineering With Scenarios) use case model. The product model also represents the product's articulated goals using goal-based requirements analysis methods. It also models the product's architecture using models of software architectures reported in [6] and [11]. The main purpose of the product model is to improve the effectiveness of requirements acquisition and evaluation/selection processes and to aid requirements-product compliance mapping and checking.

This paper is structured as follows: section 2 outlines the requirements model, the software product model and its meta-model and the requirements-product compliance model, section 3 describes PORE's situated process and how requirement and software product models inform the requirements acquisition and evaluation/selection processes. The paper ends with possible criticisms of the current approach and future developments of the PORE product model which, the authors hope, will overcome these criticisms.

2: The Requirement Model, Product Model and Product-Requirement Compliance Model

2.1: Requirements

A requirement can be viewed as a capability that a software system must supply or a quality that a system must possess in order to solve or achieve an objective within the system's conceptual domain. A critical factor in successful acquisition of requirements is to understand not only what the system under consideration should do (functional requirements), but also the way in which it should provide its services . (non-functional requirements). A broader view of requirements acquisition, therefore, should go beyond the description of what the system is expected to do (system's functionality) and include system properties and constraints under which the system must operate, (non-functional requirements).

In the CBISE development process, this view is taken further to include information about product suppliers such as the supplier's technical capabilities, application domain experience, ISO standard certification, (supplier requirements) and legal issues involved in product procurement such as negotiating contract terms and conditions, licensing arrangements, (contractual requirements). When taken in this context, requirements represent both a model of what is needed and statements of the problem under consideration in various degrees of abstraction. The PORE method uses the requirement model to both elaborate the requirements statements and to check requirements-product compliance during the product evaluation and selection process as described in section 3.

2.2: The Product Model

To enable effective software evaluation and selection, there is a need to understand the software product. PORE's product model enables each software product to be modelled in three different ways [7, 8]. One of the problems for CBISE is that, it is very difficult to compare products since there is no understandable common vocabulary or taxonomy to describe a software product. To remedy this, the PORE approach proposes a software product model to be used as an instrument for comparison and evaluation and for driving the requirements acquisition process. The eventual selection of a product to be included in the development of the information system demands some form of precise mapping between the product's properties and the customer requirements. The mappings should cover both functional and non-functional requirements and cope with the product feature dependencies and incompleteness of requirements [5].

Typically, a software product possesses sets of interdependent properties. Properties are the characteristics that describe a product and are the features of the product model. Properties represent all that we need to learn about a product, its context and the relationships between the properties. The properties model the product's observable behaviour and how it provides its services to the users. Product's properties are intended for specific purposes and modelling the properties provides information to reason about the product's suitability for its intended purpose. Dependencies and relationships exist between the product properties; e.g. between actions and functions.

2.2.1: The Product Meta-Model

To precisely model the properties of the software product, the PORE approach uses a variety of abstract meta concepts such as goals to be achieved, objects to be used, actions taking place, events, agents to perform and control actions, components involved in actions, functions to achieve actions and relationships between meta-concepts. Figure 1 depicts the PORE's software product meta-model and its primitive concepts (e.g. agent, action, software component) and the meta-relationships linking the meta-concepts (e.g. performs, achieves, depends). The meta-concepts provide specific ways and strategies of traversing the product meta-model to instantiates its instances during requirement-product compliance mapping (see section 3). Each step in the product-requirement mapping becomes a validation of the product features against essential customer requirements. The main purpose of the meta-model is to drive the requirements acquisition and product evaluation processes and to enable effective requirement-product compliance checking (see Figure 2).

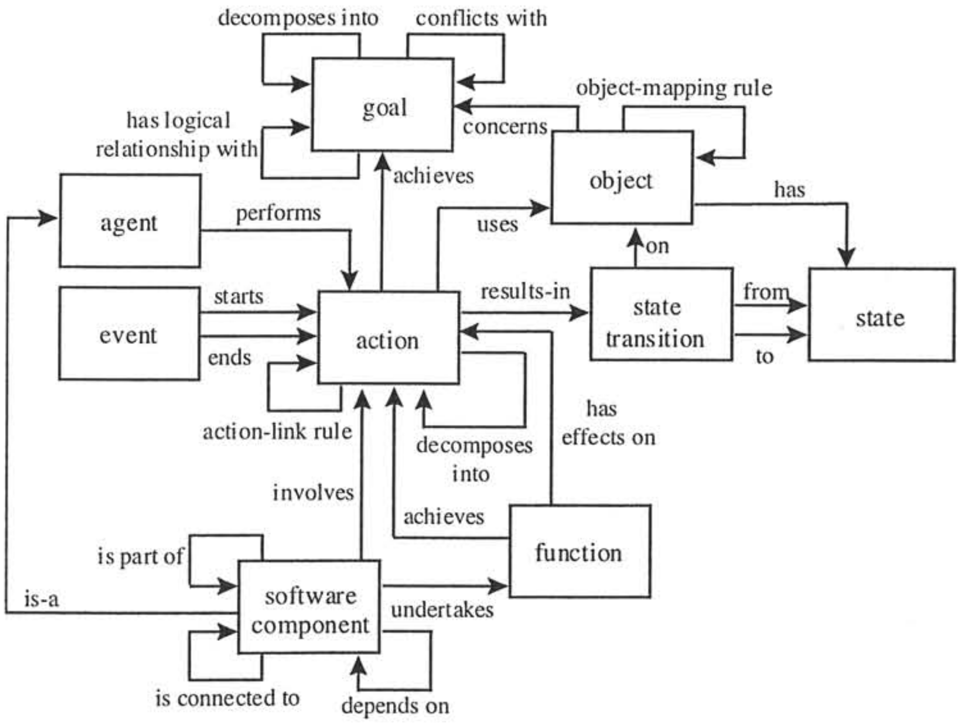


Figure 1: A meta-model for the product sub-model, showing the primitive concepts and relationships with which to model each software product

The following section briefly describes the main primitive meta-concepts of the product meta-model:

- a **goal** is a high-level objective that the system should meet [4, 2]. Goals are achieved by actions performed by agents. Goals can be decomposed into alternative combinations or logical sub groupings and may sometimes conflict with each other [2];

- an **action** is a process linked to the attainment of a goal. Each action can be cognitive, physical, system-driven or communicative. Each action can involve one or more agents, use one or more objects and may result in a state transition which may change the state of the object. Actions are linked to each other using action-link rules [9];

- an **agent** is a type of object which performs or processes actions [4]. Each agent has certain features that determine its capabilities to perform the desired actions. Each agent is either a human agent or a software component, thus enabling the requirements engineering team to model the system boundaries (degree of automation) of each product;

- an **object** is something of interest in the domain. Object instances can evolve from state to state through the application of actions. Each state of an object at some time is defined as a mapping from an object to the set of values at that time of all features of the object. The team models object use in a use case to describe both the product's information features and customer's information requirements;

- **functions** are a mode of action by which the product fulfils its purpose. They are the services and capabilities provided by the product and specify what the product is capable of performing. Functions define the behaviour of the product and the fundamental processes or transformations that the product and the hardware components of the system perform on inputs to produce outputs. The behaviour of a product's function can be mathematically characterised as a function which receives some input x and produces some output y.

- a **component** is an independently deliverable set of software services available to users or to other components [3]. The software package or COTS itself, is also a component. Components interact or collaborate with each other through connectors to accomplish solutions or to undertake complex functions. The structural features of components are collectively known as the architecture [6, 11];

- **connectors** facilitate interaction between components in order to form executable structures. Connector types include protocols, procedure calls, remote procedures calls, buffers, event broadcasts, constructs, etc. Connectors can be further divided into data-carrying connectors, control-oriented connectors and hybrid forms that exhibit both characteristics to some degree;

- **dependencies**: the product model also defines dependencies between components. One component X is dependent on another component Y if component X contains a call or connection to component Y, that is X is a dependent of Y [10]. The model identifies different types of dependencies between components. Examples are functional dependencies, trigger dependencies, precedence dependencies, constraint dependencies [8]. These dependencies in combination with component connectors, provide a simple but useful semantics for modelling a software product.

2.3: The Product-Requirements Compliance Model

The relationship between the product model and the requirements model is shown in Figure 2. The requirements model contains zero, one or more requirements. Each requirement has attributes and values to indicate its type, priority and owner. The second part of the model is a model of candidate software products. Likewise, a product model contains zero, one or more product features, and each product feature has similar attributes as requirements. The third part of the model represents the degree of compliance between the customer requirements model and the product model. Compliance is modelled as a set of relationships between one or more customer requirements and one or more product features. Attributes on each relationship have values to indicate whether or not there is compliance, the degree of compliance, and the degree of confidence in the compliance.

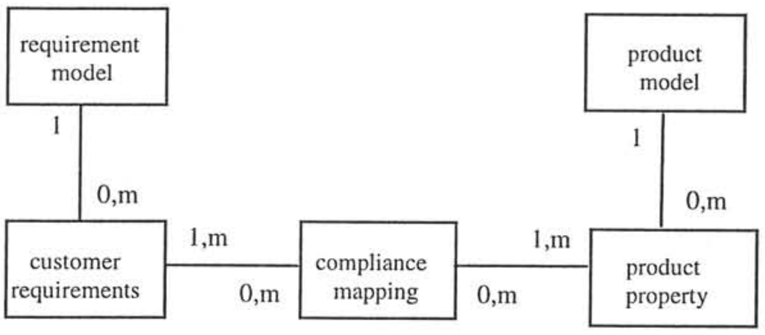


Figure 2: the relationships between the requirement model, product model and requirement-product compliance model

The compliance model is made of concepts that are instances of the product meta-model abstractions and atomic customer requirements. PORE uses meta-concepts of the product meta-model as a criteria for measuring and comparing conceptual similarities between candidate products and compliance with customer requirements during product evaluation and selection .The PORE's product evaluation and selection process is described next.

3: Product Evaluation and Selection

The purpose of evaluating software products is to judge how well do they comply with the specific system's objectives and customer requirements. The PORE method prescribes a goal-based process to achieve product evaluation and selection [8]. Figure 3 graphically depicts this process. Candidate software products are rejected according to their compliance with essential customer requirements. In the early stages of the process, the products that are least compliant with essential requirements are rejected using the high-level atomic customer requirements. As the customer requirements become more complex and detailed, a small number of products are short-listed for a more complex and time-consuming compliance evaluation.

Central to the product evaluation and selection process is the product model. The product model drives both the requirements acquisition and the product selection processes. At each stage of the process, product properties are mapped to customer functional requirements and a compliance model is developed to check and to determine the degree of compliance between each candidate product and the customer requirements. The compliance model is a composition of compliant requirements and product properties. It is modelled as a set of relationships between one or more customer requirements and one or more product features. Attributes on each relationship have values to indicate whether or not there is compliance, the degree of compliance, and the degree of confidence in the compliance as depicted in Figure 2 and Figure 3.

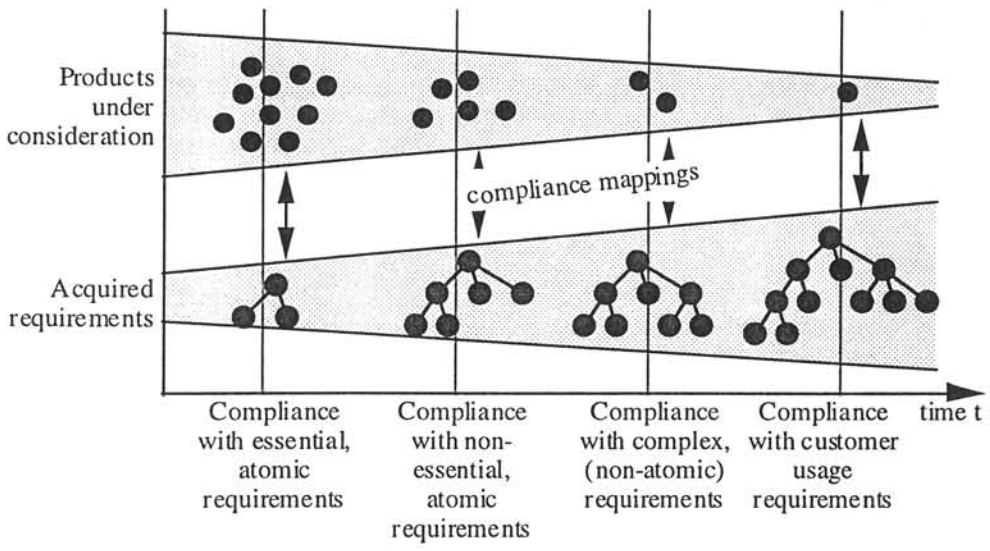

Products under consideration
compliance mappings
Acquired requirements
Compliance with essential, atomic requirements
Compliance with non-essential, atomic requirements
Compliance with complex, (non-atomic) requirements
Compliance with customer usage requirements
time t

Figure 3: Overview of PORE's goal-based process. The number of acquired customer requirements increases while the number of candidate software products reduces. Successive rejection of non-compliant products leads to a systematic reduction in the number of candidate products during the process.

To achieve the goal-based process of Figure 3, the PORE method prescribes a route map with four generic high-level processes, as depicted in Figure 4. The four generic processes are the (a) requirements and information acquisition process which acquires requirements from main stakeholders and information about suppliers and available candidate products in the market; (b) requirements and information analysis process which analyses the acquired requirements and supplier and product information for accuracy and completeness; (c) requirements-product compliance checking process which uses decision-making techniques and tools to analyse and check requirements-product compliance mapping; and (d) candidate product selection or rejection process which selects one or more compliant products and rejects non-compliant ones. The order in which the four processes are undertaken is context-driven and is determined by the current state of the situation model.

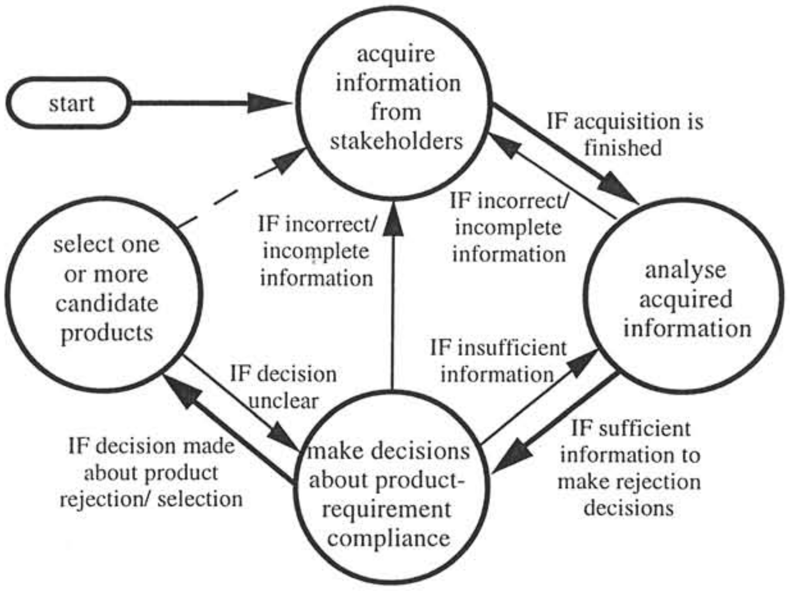

start
acquire information from stakeholders
IF acquisition is finished
IF incorrect/ incomplete information
IF incorrect/ incomplete information
select one or more candidate products
analyse acquired information
IF insufficient information
IF decision unclear
IF decision made about product rejection/ selection
make decisions about product-requirement compliance
IF sufficient information to make rejection decisions

Figure 4: graphical depiction of a route map showing PORE's high-level processes. The achievement of each essential goal is a broad sequence, in which the first process is acquisition of information from stakeholders and the last is selection of one or more candidate products, but the sequence of the intervening processes is not predetermined and each process can be repeated many times.

As shown in Figure 3, as the team goes through the process iteratively, guided by the route map depicted in Figure 4, the number of candidate products reduces and the number and detail of customer requirements increases. The PORE goal-based process model is the central component of the PORE approach. Each process is modelled as a collection of process chunks [8]. PORE combines the process chunks together in different sequences to form different processes to achieve different goals. Each process chunk has 6 attributes as shown below:

```
Process-Chunk:
  Name: Unique-identifier
  Goal: Verb (Result | Object) (Manner)
  Process: {Goal}
  Situation: {property (sub-model)}
  Input-information: {content (situation sub-models}
  Technique: {technique in PORE method box}
End-Process-Chunk
```

An example of the application of PORE process chunks is shown below. The goal is to reject from the candidate products list, those products which do not comply with the essential goal-1:

```
Process-Chunk:
  Name: 1.0
  Goal: Reject software products non-compliant with essential-goal-1
  Processes:   Acquire atomic-functional-requirements
```

```
                Acquire product-information
                Acquire product-contract-information
                Acquire supplier-information
                Determine product-requirement-compliance
                Determine supplier-requirement-compliance
                Determine contract-requirement-compliance
                Analyse product-requirement-compliance
                Analyse supplier-requirement-compliance
                Analyse contract-requirement-compliance
                Determine non-compliant software products
                Reject non-compliant software products
  Situation: None
  Input-information: None
  Technique: None
End-Process-Chunk
```

As depicted in Figures 3 & 4, the product evaluation and selection process is very complex and at any point in the process, there are a large number of possible situations which can arise. PORE enables the requirements engineering team to model these situations based on Suchman's definition of a situation [12]. PORE models a situation in three parts. The first part models the current state of the customer's requirements; the second part models the degree of compliance between the customer model and the product model and the third part models the candidate software products, (also see Figure 2). PORE's context-driven process is made more complex by the large number of techniques from different disciplines which are available to achieve each process. Maiden & Ncube [12] identified numerous techniques to acquire and analyse information, and to make complex decisions and select candidate products. It is rare that a requirements engineering team will be familiar with all of the techniques, so PORE makes them available through its method box. For each technique, the method box gives information sources about the technique, advice about the technique's use and prototypical examples of its use.

4: Discussion and future work

We believe that the software product model is a novel approach to acquire requirements and evaluate and select products during the CBISE development process.

However, the PORE approach still has significant limitations that we aim to overcome in its future developments. First, although it is important to model software products, the current approach is labour-intensive. One solution is to guide the requirements engineering team to model product features needed to infer properties about the product sub-model and its contents. In particular, this will ensure that each product is modelled to a sufficient level of detail. Another solution is to parse natural language descriptions of software products. A software tool (CREWS-SAVRE tool) is currently being developed so that the requirements engineering team can enter natural language use case descriptions which are then parsed to infer actions, agents, objects, states, state transitions and other features which are central features of PORE's product sub-model [1]. We are looking forward to reporting the results from this work in the near future. A detailed description of the PORE method is available at its dedicated web site: *http://www.soi.city.ac.uk/pore/welcome.html*. We welcome any feedback and suggestions.

5: References

[1] Achour C.B. & Rolland C., 1997, 'Introducing Genericity and Modularity of Textual Scenario Interpretation in the Context of Requirements Engineering', CREWS Technical Report, Centre de Recherche en Informatique, Universite de Paris 1, Paris, France

[2] Anton A. I. & Potts C., 1998, 'The Use of Goals to Surface Requirements for Evolving Systems', to appear in Proceedings IEEE International Conference on Software Engineering, IEEE Computer Society.

[3] Brown A.W & Short K, 1997, 'On Components and Objects: The Foundations of Component-Based Development', Proceedings Fifth International Symposium on Assessment of Software Tools and Technologies (SAST'97), 112-121

[4] Darimont R. & van Lamsweerde A., 1996, 'Formal Refinement Patterns for Goal-Driven Requirements Elaboration', Proceedings of 4th ACM Symposium on the Foundations of Software Engineering (FSE4), San Francisco, Oct. 1996, pp 179-190

[5] Finkelstein A.C.W., Spanoudakis G. & Ryan M, 1996, 'Software Package Requirements and Procurement'. Proceedings 8th International Workshop on Software Specification and Design, IEEE, Computer Society Press, 141-145

[6] Garlan D., Allen R. & Ockerbloom X., 1995, 'Architectural Mismatch or Why it's hard to build systems out of existing parts', Proceedings 17th International Conference on Software Engineering, IEEE Computer Society Press

[7] Maiden N.A.M., & Ncube C., 1998, 'Acquiring COTS Software Selection Requirements', IEEE Software Journal, March/April 1998.

[8] Maiden N.A.M. & Ncube C. 1998, 'Guiding the Component-Based Software Engineering Process', Submitted to IEEE Software Special Issue (September)

[9] Maiden N.A.M., Minocha S., Manning K. & Ryan M., 1998, 'CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements', Proceedings 4th International Conference on Requirements Engineering (ICRE98), IEEE Computer Society Press.

[10] Ray M., 1996, 'Dependency Diagrams, Cross Roads', The ACM Student Magazine, vol. 2(3), February 1996.

[11] Shaw M., 1996, 'Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does', Proceedings 8th International Workshop on Software Specification and Design, IEEE Computer Society Press, March 1996

[12] Suchman L., 1987, 'Plans and Situated Actions: The Problems of Human-Machine Communication', Cambridge University Press.

Business Objects: The Next Step in Component Technology?*

Ernst Ellmer, Wolfgang Emmerich
Dept. of Computer Science
University College London
London WC1E 6BT, UK
{w.emmerich | e.ellmer}@cs.ucl.ac.uk

Abstract

Component technology seems to be a promising approach towards more efficient software development by enabling application construction through "plug and play". However, the middleware supporting this approach is still complicated to use and distracts the attention of the component developer from the application domain to technical implementation issues. Business objects are intended to hide the complexities of middleware approaches and provide an easy to use environment for application developers. We conceptualize business object approaches by presenting a common model and survey some major players in the marketplace. We conclude by identifying implications of business objects on information systems engineering.

1 Introduction

Component-based software development is intended to facilitate and speed up application development. Reusable components are supposed to be plugged together in a distributed and inter-operable environment. Middleware systems such as CORBA, DCOM or Java RMI provide the necessary infrastructure for this purpose. Middleware systems offer a number of facilities important for information system development, including transactions, persistence and security. They are, however, rather complicated and difficult to use. It is desirable to hide the complexities from application developers and allow them to deal with concepts they know from their application domain rather than with implementation issues. An approach towards achieving this goal are business object facilities and common business objects.

In this paper we summarize a study that presents an overview and evaluates business object approaches that are currently be developed or are on the market already. We characterize business object facilities and common business objects and give a brief overview of six competing approaches, namely the Combined Submission to the OMG Business Object Domain Force (OMG approach), the SSA Business Object Facility (SSA approach), Sun's Enterprise JavaBeans (Sun EJB), IBM's San Francisco Framework (IBM SF), Microsoft ActiveX (MS ActiveX) and the SAP Business Framework (SAP BF). Finally, we conclude with some important implications of the business object approach for information systems engineering in order to encourage discussion on their advantages and disadvantages. A detailed description of our study focusing on an in-depth analysis and comparison of the approaches can be found in [2].

*This work was partially funded by the Austrian FWF (Fonds zur Förderung der wissenschaftlichen Forschung) under grant J1526-INF.

2 Business Object Concepts

In its 1996 Request for Proposal the OMG defined Common Business Objects and Business Object Facilities as follows [4]:

Common Business Objects (CBO): Objects representing those business semantics that can be shown to be common across most businesses.

Business Object Facility (BOF): The infrastructure (application architecture, services, etc...) required to support business objects operating as cooperative application components in a distributed object environment.

In the sense of these definitions, the Common Business Objects can be regarded as components providing core application functionality for a certain business domain. They are intended to build the kernel of applications tailored to the needs of a certain organisation. Business Object Facilities on the other hand represent the enabling technology for creating and executing business objects. They aim at hiding middleware complexities from application developers. The business object approach promises to realize the following achievements:

Simplicity: Business object developers can concentrate on business solutions. Infrastructural concerns, such as transactions or persistence are entirely transparent for them.

Reusability: A business object is able to support different applications. The business object facility supports the construction, integration and execution of such reusable components.

Extensibility: Business objects are extensible with mechanisms, such as inheritance and delegation. In this way, specialized solutions can be created quickly from common business objects.

Scalability: A key feature enabling scalability is distribution. Business objects are therefore distributeable across different servers.

Heterogeneity: Business objects are able to communicate with each other although they may reside on heterogeneous hardware and operating system platforms and may be written in different programming languages. The BOF hides any heterogeneity from the developer.

Portability: Business object implementations are platform independent. BOFs hide the underlying operating system and legacy applications and make business objects portable between different deployment platforms.

Interoperability: There will be several BOF providers in the market. Business objects that reside in one BOF are able to communicate with business objects residing in other BOFs.

In order to understand and compare the various business object approaches, we developed a common model for describing their features and characteristics. It can be considered as a layered architecture. While the top three layers are ordered according to the level of abstraction they achieve from the underlying hardware and software infrastructure, the lowest layer deals with conceptual issues. Figure visualizes the model.

The **concept layer** is intended to provide the "big picture" of an approach. It describes how an approach manages business objects and their interoperability at a high level of abstraction without technical detail.

The **infrastructure layer** is concerned with the underlying middleware of an approach. Two issues are addressed. The *basic technology* issue covers the infrastructure used for brokering messages. Examples are OMG/CORBA, Microsoft DCOM, or Java RMI. Furthermore it covers operating system platforms and programming languages supported. The *services* issue is concerned with system-level services a business object developer is provided with in order to create applications. Examples are naming, events, life cycle, persistence, transactions, concurrency control, relationships, externalisation, query, licensing, properties, time, security, object trader, and object collections.

The **facility layer** deals with business object facilities. According to our model, this layer contains five issues. First, we consider *interfaces*. On the one hand, clients or other business objects need to access a business object;

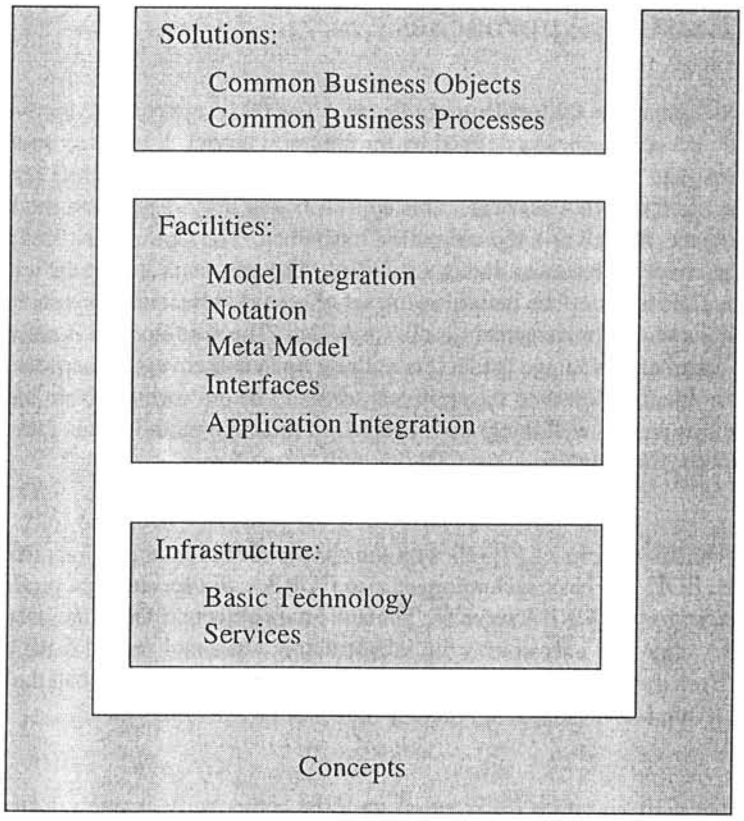


Figure 1: Common Model of Business Object Approaches

on the other hand, a business object has to access BOF services. Access to clients can for example be granted via static or dynamic CORBA interfaces or through RMI-compliant Java interfaces. Services can for example be made accessible by letting a business object class inherit from a system provided super-class. A very important feature of business objects is their ability to access legacy applications such as databases or application suites. The *application integration* issue focuses on the access mechanism rather than on the currently available set of accessible applications. Mechanisms for accessing legacy databases are for example schema mappers linking object attributes to tables in a relational database. For the development of applications based on business objects, it is important to provide modelling support. Two issues are concerned with modelling. First, modelling concepts are needed in terms of a *meta model*. It defines the constructs available for specifying business objects. Examples of such constructs are business object types, business system domains, or events. Beyond a meta model, a *notation* for representing business objects is needed. The representation has to cover the semantics of the business objects rather than a technical description of the interface specification. Last but not least, we evaluate the degree of *model integration* of the approach with business analysis models on the one hand and interface definition or implementation languages on the other hand. Integration with analysis models means that there should be a mapping between analysis model such as a UML specification and the business object model. On the other hand, a mapping between business object model and interface definition languages such as CORBA IDL or implementation languages such as Java is desirable.

The **solution layer** comprizes the highest layer of the comparison model. It is concerned with concrete implementations of business objects that can be used by a developer as a basis for a business application. Two issues are considered by our model. *Common business objects* on the one hand represent semantics common to a whole business domain. Their intend is to relief an application developer from implementing features that can be provided by pre-built components because of their generality. *Common business processes* on the other hand go even further than common business objects and provide a skeleton application for business domains. That means not only common objects are identified and implemented but also their interactions in order to perform a broader business task.

3 A Brief Survey of Existing Approaches

Combined Submission to the OMG Business Object Domain Force The OMG approach [5] provides a wide coverage of features of business object approaches as defined by the common model. It is based on a type manager for managing business objects. The basic technology is of course CORBA and five services are supported that are slightly different from the basic CORBA services. This approach also provides a meta model defining concepts for modelling business objects as well as a representation formalism. The Component Definition Language (CDL) is a textual language covering business object semantics. The approach integrates with UML as an analysis approach and CORBA IDL for interface definition. A set of common business objects for business financials, order management, and warehouse management is also provided. The specification is currently being implemented by several vendors. Its main advantage is that it is entirely open and covers all aspects from basic infrastructure to common business objects. However, the approach seems to be put together from various ideas and the three specifications it contains are not well integrated. Moreover, the meta model seems rather complex and the practicability of the textual specification language CDL has still to be proven.

SSA Business Object Facility The SSA approach [7] relies on the concept of executable object (XO) as business object that are managed by the BOF. The basic technology is also CORBA. Six services are provided by the approach, which again differ from the basic CORBA services. Semantic data objects (SDOs) are used for wrapping legacy applications. The SSA approach only covers the infrastructure layer and provides the application integration and interface features from the facilities layer. It appears more compact and clear than the Combined OMG submission. SSA is currently implementing the approach within ESPRIT project OBOE.

Sun's Enterprise JavaBeans The EJB approach [8] is based on three components, namely Enterprise JavaBeans classes implementing business objects, Enterprise JavaBeans containers responsible for life-cycle management of the beans, and Enterprise JavaBeans servers for providing system-level functionality. Sun's Enterprise JavaBeans extend Java technology with component server capabilities based on Jave Remote Method Invocation (RMI). The approach covers the same range of features as the SSA approach. Services, however, are weak and need to be specified in more detail.

IBM's San Francisco Framework The IBM initiative [1] is based on Java technology but extends the basic RMI mechanisms. The services provided are based on the OMG specifications but simplified or extended where considered to be necessary. Furthermore Java features are included. Database applications can be integrated via schema mappers. San Francisco is well integrated with existing modelling approaches. Rational rose models can be directly translated into Java by the means of a code generator. Common business objects as well as common business processes are provided by the framework. IBM's San Francisco is already available as Version 1.0 and provides all features except a modelling approach. Its strengths are undoubtedly the common business objects and processes that are supposed to provide about 40% of an application.

Microsoft ActiveX Microsoft provides a set of component technologies marketed under the name ActiveX. However, business objects as defined in this paper are not explicitly addressed. The Microsoft Transaction Server [3] provides some basic facilities that might be of benefit for business objects and thus is included into this survey. It hosts ActiveX components and provides distribution, life-cycle, and security services for them. The base technology is COM/DCOM. MTS is available in Version 2.0 and needs a Windows NT Server for running. The services provided as well as the mechanism for integrating (Microsoft) legacy applications can compete with the other approaches. An advantage of the Microsoft technology is that it is already available and in use while many of the other approaches are still in a specification state.

SAP Business Framework An alternative approach to the ones introduced above is SAP's Business Framework [6]. It is not intended to support the development of business objects but provides a business object interface to SAP R/3. An object repository layer is established on top of R/3 providing access to non-SAP applications.

The approach therefore is especially strong in the area of common business objects and processes as well as application integration (with R/3). Another advantage is that the business objects can be accessed from CORBA, COM/DCOM and Java. The participation of SAP business objects in global services, such as transactions remains to be demonstrated.

4 Conclusions

This paper was devoted to the next step in component technology represented by business object approaches. We developed a conceptual model consisting of four layers, namely concepts, infrastructure, facilities, and solutions. We furthermore provided a brief overview of six competing approaches. To conclude this paper, we want to point out some implications of business objects on information systems engineering as a basis for discussing pros and cons.

Business object approaches will have impact on the *development processes*. Applications will be built in a bottom-up style rather than top-down by plugging together various existing and newly developed business objects. The processes are more likely to follow a incremental approach rather than a waterfall model; applications can be assembled rapidly, enabling cross-checks with the users. Furthermore, development processes are supposed to be shorter. Executables can be reused instead of high level design documents or code.

Business application developers will need less implementation knowledge and can concentrate on the application domain. Services, such as transactions, persistence, security, life cycle management will be provided by the business object facility. "Intelligent IT users" can be integrated in the process as developers rather than users stating requirements that then have to be interpreted and realized by implementation specialists.

Development tools will need to be sophisticated in order to hide implementation complexities from application developers. This is a challenge for tool providers. Integrated toolkits covering all phases from analysis to implementation will be needed. In combination with business object approaches, appropriate development tools will make component technologies usable.

A *market for common business objects and processes* will be established. Applications will be assembled from bought as well as self-developed objects. The current "reinvention of the wheel" phenomenon can be avoided by making independent objects inter-operable and deployable in different environments.

Another interesting implication of business objects are enhanced *integration possibilities* with other information systems such as workflow systems. Current systems suffer from a lack of integration of application functionality. Based on component technology, workflow systems could evolve into "business operating systems" providing a graphical user interface as well as process support for application components.

References

[1] K. Bohrer. Middleware Isolates Business Logic. *Object Magazine*, November 1997.

[2] W. Emmerich, E. Ellmer, Birgit Osterholt, and Roberto Zicari.

[3] S. Hillier. *Microsoft Transaction Server Programming*. Microsoft Press, 1998. To appear.

[4] OMG, 492 Old Conneticut Path, Framingham, MA 01701-4568. *Common Facilities RFP-4: Common Business Objects and Business Object Facility*, January 1996.

[5] OMG. Combined Business Object Facility: Business Object Component Architecture. Technical Report TC Document BOM/98-01-07, Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA, JAN 1998.

[6] SAP. Benefits of the Business Framework. `http://www.sap.com/bfw/media/pdf/50016302.pdf`, 1998.

[7] *OMG BODTF RFP-1 Submission - Business Object Facility.* 500 West Madison Avenue, Chicago, USA, November 1997.

[8] A. Thomas. Enterprise JavaBeans: Server Component Model for Java. Technical report, Patricia Seybold Group, 85, Devonshire Street, Boston Mass. 02109, DEC 1997.

A Component-based Integration Framework for Technical Information Systems

G. Paul, M. Endig, K.-U. Sattler
Department of Computer Science
Otto-von-Guericke-University Magdeburg
P.O. Box 4120, D-39106 Magdeburg, Germany
EMail: {paul|endig|kus}@iti.cs.uni-magdeburg.de

Abstract

Today's software market is characterized by the existence of a multitude of atomic software systems for various application areas. These systems are usually "closed" solutions for certain functions. In general, a communication between the systems is only possible through system-specific interfaces. An integration (communication) of these software systems is enabled by recent developments in the area of computer science. For example, by the application of special middleware platforms an integration of software systems is possible. The aspect of integration is essential for software systems in the area of the technical product development. Currently, software systems are available for the support of the individual steps of the product development process. Such systems are denoted as technical information systems. However, due to the inherent complexity, an integration of all these systems over the whole product development process is not possible. This paper presents a general approach to configuring user-specific technical information systems based on a four level integration framework.

1 Motivation

The current software market is characterized by the existence of a multitude of large and complex information systems for various application areas. The current systems offer functionalities which are necessary for the solution of certain tasks. Even in very complex applications, like the computer supported product development, it is no longer possible to offer all necessary functionalities in *one* information system. Due to the very high inherent complexity an implementation of well structured and operatable systems is not realized at the same time. In today's information systems this is attempted with the result that the systems are large monolithic software packages which are difficult to operate and hardly to grasp. These systems have additional deficits, among other things, with regard to their integration behavior. They are usually "closed" solutions with no or only limited cooperation possibilities. They can only be acquired in theirs "totality" or in special modules. This leads to two further substantial deficits of today's information systems. On the one hand, in a certain software environment functionalities may be available redundantly. For example, for a special enterprise this means that the costs of software packages are very high without any visible effects. On the other hand, functionalities which are never needed are also part of this "large" software (system) package. For example, an enterprise bought a special software package and only 60% of it are used. These systems are hardly scalable to special situations. In an enterprise a special software environment is necessary. The provision of technical information systems scalable on this special situations is very difficult.

Following [9], these deficits can be avoided by the use of a *component-oriented approach* for the development of information systems. In this case, the information systems are divided into their *core components*. Each core component comprise all functions, software modules, and necessary data structures which are required to solve *one* particular task. For example, a special CAD-system contains different kinds of core components (viz. components for 2D-/3D-functionalities, a component for displaying the CAD-model etc.). This kind of the usage of software systems appropriate interfaces for the summarization in special components have to be provided by the systems. By configuring the core components, component-based information systems can then be implemented.

In the recent years, new technologies have been developed to provide a basis for realizing manufacturer-overlapping information systems which are available in a heterogeneous network. Such an information system contains different kinds of components from various software producers. Two technologies for different hard- and software environments have been established: *the Common Object Request Broker Architecture* (CORBA) of the Object Management Group (OMG) [8] and Microsoft's *(Distributed) Component Object Model* (DCOM) [2, 14]. Both technologies represent a *Broker Architecture* in the sense of [7]. Whereas CORBA already represents a completely standardized integrated technology, the specification of DCOM is not completed in the sense of a "standardized integrated" technology. In our opinion, to use the advantages of both technologies for "new" manufacturer-overlapping information systems a combination (integration) of both technologies is necessary. So, the application of "all" software systems based on such technologies can be used directly for a "new" information system. The methods and concepts required for such an integration are already available in commercial products and can be applied. For example, ORBIX from IONA Technologies provides an appropriated interface for such an integration.

The CORBA and DCOM technologies can be used as a basis of *component-based* information systems. However, there is the problem that these technologies are "completely" or directly applicable only for new developed systems. In the companies a lot of monolithic information systems are available. It is very important that such systems will be adaptable to the new technologies. In this case, the systems can be directly used continues in "new" information systems. The adaptation to the new technologies can be realized, with the help of a re-engineering process. This process is very time-consuming and cost-intensive.

For the development of complex products (for instance technical machines etc.) a multitude of technical information systems are used. In this case, on the one hand, there are systems which claim to support total solutions, for example CAD/CAM. Such systems are usually very complex in size and handling and, thus, represent large monolithic software systems. On the other hand, there are applications which supply "niche-solutions" or represent small special applications for certain subtasks. Component of the further considerations should be the presentation of an integration framework. By the application of this framework, technical information systems on the basis of a component-based approach can be used better without the necessity to perform a "complete" re-engineering process.

2 Fundamentals

All technical information systems available on the market serve for the creation, administration and representation of certain information, for example data or documents. For this, special functionalities in a certain order are necessary and have to be supplied to the current user of the system. Due to this fact, all technical information systems can be divided into four layers. On the *data layer* the administration of all data objects necessary for the system is realized. All functionalities of the system are offered on a *function layer*. These functionalities are coordinated in a certain order on a *process layer*. For the user the total functionalities of the system are (then) offered on a *user layer*. In technical information systems these four layers are available. As an example, it is referred to a system for the administration of product data (e.g. a EDM- or PDM-system). In general, in such a system a database for the administration of the product data exists (on the data layer) and a lot of functionalities for the manipulation of product data are provided (on the function layer). The creation of a product structure is a typical function in such a system. Certain functionalities must be executed in a certain order (on the process layer) and have to be offered on a user layer.

The decomposition of software systems into four different layers contains still an another aspect which is particularly important for the development of modern technical information systems. As a matter of principle, the different layers of a software system correspond to the possible integration dimensions of a software systems following [16]. In this case, the integration of information systems can take place on the following layers:

- *Data Layer:* On this layer the necessary data objects, for example documents or product-describing data, are summarized and administered in a uniform model. Depending on the current application systems, the objects can thereby be stored either directly or indirectly using a reference to the actual data objects. As storage medium, databases can be used. On the data layer a system-wide data integration can then be implemented by the application of federated database concepts [15]. Other integration concepts are conceivable, too e.g., a uniform product data model on the basis of product data standard AP214 and AP212 from the STEP-standardization line ISO10303 [6].
- *Tool Layer:* Following [9], on the basis of the component-oriented use of software systems special software components for user-specific software configurations are applied. On this layer the decomposition of software systems into their core components and an integration (communication) of these a better usefulness of the tools is possible. The software tools represent the virtual modules or *components* with a granularity from simple modules with few functionalities to complete complex applications. The introduction of a com-

ponent model the mapping from the configuration of an engineering environment as a set of correlated tool components can be enabled.

- *Process Layer:* Component-dependent functionalities on the Tool Layer which are available in the frame of a tool configuration shall be used on this layer in order to configure task-specific sequences. For that, special modeling concepts and methods for the flexible configuration of sequences are necessary. For example, an appropriate method for the mapping of parallel processes is very important. The methods of modeling have to be implemented in a special tool for an execution environment of configuration. In the simplest case, a Petri-net approach can be used [11]. On this layer an integration of functionalities of the Tool Layer into any task-specific process sequences is possible. The introduction of a special process model and appropriate concepts and methods make it possible. For example, in [4] different kinds of software process modeling and execution methods are described. However, there is the problem that these methods are only applicable through an appropriate adjustment process. The process is very time-consuming.

- *User Layer:* On this layer all functionalities of the complete system must be offered or represented in an appropriate/uniform format. The independence from the hard- and software, which is used, has to be guaranteed.

On the basis of these four layers an integration framework for technical information systems can be specified. The information/software systems available in a special software environment of an enterprise can be used better through the application of this framework. Through the supply of system-wide functionalities within this framework, a higher efficiency of functionalities of individual software modules in the whole system can be achieved. A possible integration framework is the subject of the following considerations.

3 Integration Framework for Technical Information Systems

In our opinion, for the implementation of modern component-based information systems a uniform system-wide integration framework have to be used. For example, systems based on such a framework "standardized" system-wide interfaces are provided and all other systems can used these. For the implementation of such a framework appropriate technologies like CORBA or DCOM are necessary. In order to also include existing monolithic information systems into such a *new* system, the supply of an integration framework is meaningful. The general goal of this special framework is to provide the foundation of a component-based technical information system. The advantage for an enterprise is that the available software systems can still be used. Additionally, a higher total functionality can be obtained by the supply of system-wide functions. We suggest an integration framework for a component-based technical information system, as shown in Figure 1.

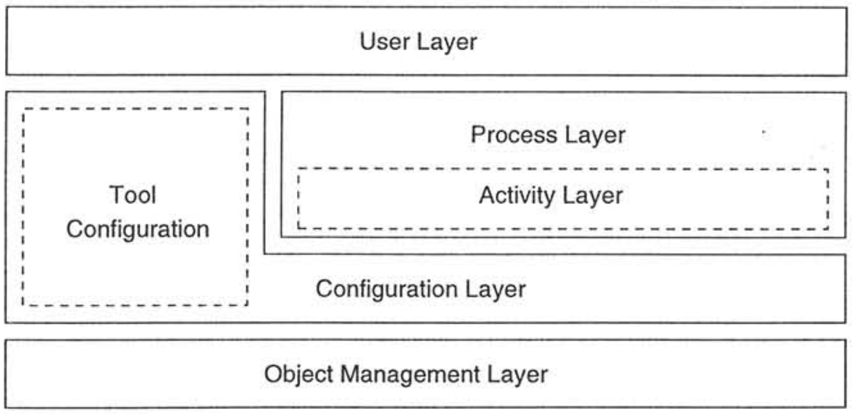


Figure 1: PACO - An Integration Framework

As discussed in the preceding section, in order to realise a constant component-based information system, our framework distinguishes the four different integration layers. The necessary (available) software systems are partitioned into components which belong functionally together (core components). By the use of a component model the components are combined to enterprise-specific software configurations. A component is a software module with well-defined interfaces which represent the generalization of the behavior and the internal representation and implementation "covered". Thereby each component belongs to a component class which defines the characteristics (in the form of interfaces) and implementation. In principle, a distinction between two different kinds of component interfaces is possible:

- *Operational interfaces:*
  This interface defines a set of operations which can be executed by a component or which a component can invoke at another one. An operation is described by means of a *signature* consisting of an operation name and a set of parameter definitions.
- *Event interfaces:*
  This interface defines a set of events which a component can be announced on the environment. Events will be described again by means of a signature.

An interface can be derived from other interfaces. Moreover, an interface can extend the derived interfaces with new operations or events. The derivation is only allowed within the scope of the same kind of interfaces. A component has to implement at least an operational interface. Furthermore, a set of abstract points of interaction, so-called *Slots*, can be defined for a component. So a component can uses services of another component. Slots are defined by an system-wide unique identifier and the necessary interfaces. The interfaces allow the explicit connection of a component with the component which provides a special service. A connection of Slots of different components by means of *Interactors* special configurations of components can be supplied. The model is allowed to define interactions independently of the implementation of the components and to reduce the dependencies between the components. A configuration of components represents the clipping from the tool supply of the environment, necessary for the processing of a task. Further, a configuration of any software components or core components to component-based information systems is possible [12].

For example, with this model the functionalities of an Assembly-Editor based on a CAD-system and the services of a Product-Browser could be connected. The several components will provide their typical special functionalities. The connection between this components the application of functions from other components is possible directly. In this case, the CAD-system do not have to offer a special component for the administration of product structures. The necessary functionalities are provided by an other component from the special software environment. Additionally, the model can be extended through a component for the document selection or administration (see Figure 2).

Functionalities provided by a configuration can be used to model and execute task-specific process sequences. Due to the sometimes very high granularity of these functionalities (for example methods of concrete objects), the supplement of the virtual Process Layer is meaningful. The goal of flexible modeling of technical process sequences an appropriate summarization of such functionalities is necessary. Therefore, the Process Layer is extended by an *Activity Layer*. On the Activity Layer, the "atomic" functionalities of the underlying configuration are combined with high-order functionalities of the total system. The combination of special functions to high-order functions *Compound Activities* are used on the basis of *Activity Nets*. On the Process Layer the technical process sequences then can be configured based on these (Compound) Activities.

Based on the suggested integration framework, the introduction of an enterprise-wide technical information system is supported by the use of implementation-independent specification languages. On the several layers all relevant objects and object relations are specified (for instance components and relations between this components). Thus, an enterprise-specific system can be described on an abstract layer. Then, the concrete systems can be generated from this description. For the specification of necessary data objects on the Object Management Layer generically *Data Definition Languages* can be applied [5]. For a flexible adaptation of the software environment on the Configuration Layer the individual software components or the relations between these components are specified with the help of the language *Tool Interconnection Language* (TIL). From this TIL specification of a special software configuration with the help of one TIL compiler the internal representations of a configuration readable by the runtime environment and the adapter code required for the interconnecting of the software components are generated. Based on the configuration description the runtime environment is able to instantiate and connect the components directly or via the generated adapters. In the context of a TIFRAME-Framework the TIL specifications, the TIL compiler and the necessary adapter code was summarized [12]. Based on the Configuration Layer for the configuration of task-specific sequences on the Activity Layer an appropriate description model was defined for

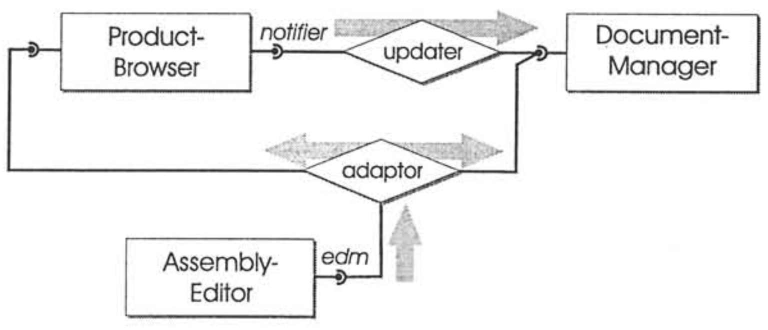


Figure 2: Definition of a Configuration

the specification of all relevant information (e.g. the introduction of a role model to the access management). This include, among other things, the definition of model- and execution-specifications of Compound Activities. From the internal representations of a configuration which is readable by the runtime environment, this specification with the help of a special compiler are generated and for runtime environment provided for the instantiation and the execution [3].

Advantages of this integration framework are in particular the possibility of the component-oriented use of available, specialized information system components and the associated possibility of the configuration of special technical information systems. Another advantage is the possibility of configuring (describing) necessary process sequences. The technical information systems available in a special enterprise environment can be used in such a manner that first the possible system components are combined into an enterprise-specific configuration of components. Then, the functionalities supplied by the components are used for the configuration of task-specific sequences on the Activity Layer. The special sequences can be supplied on the Process Layer. In this way, the framework concept can be realized gradually. At first, the concepts of the Object Management and Configuration Layer are implemented. In a second phase, the methods and concepts of the Process and Activity Layer can be provided. In general, the stepwise introduction of systems based on this framework concept has two advantages. On the one hand, after the introduction of the Configuration Layer concepts the higher functionalities from the whole system are used directly at the same time. On the other hand, the understanding of the "new" system environment by the people in an enterprise will be better through a stepwise introduction.

For the realization of these concepts, first a framework for the component-oriented tool integration in design environments has been specified and prototypically implemented [10]. The configuration of special technical information systems is possible. For the configuration of necessary (Compound) Activities, first concepts for specification were implemented. In the future, these Activities will be used for the implementation of a product structure dependent process sequence on the Process Layer.

4 Related Work

The improvement of the support of the product development process is a current subject of research where two different research areas can be recognized. Whereas the first area deals with the improvement of the support of individual development phases, the second area tries to find a general approach for the computer supported product development process. In this context, it shall be referred to the *KOMFORCE*-initiative which aims at the development of methods, reference models, and tools for integrated development workplaces [13]. It is, similarly to the PACO[1] integration framework, based on different layers of integration. Other works, e.g. "Architecture for a frame system in the product development", include the tools which are available in an enterprise, in their totality [1]. In contrast to the described concept, our approach is based on a component-oriented use of software tools.

The development of component-based software systems is subject of current research in the educational area as well as in the commercial area. Thus, with the introduction of the OLE2-technology (Microsoft) the basis for the development of component-based software systems were already lied. In this case, a document-oriented approach

[1]PACO: stands for *P*rocess, *A*ctivity, *C*onfiguration, and *O*bject Management Layer

has been pursued. However, there is the problem that the components of the user interfaces was in the focus. Other large software manufacturers try to establish their component technologies on the software market, such as Java Beans (Sun). These technologies are likewise aligned to components of the user interfaces. All components must support the interfaces given by the current environment in order to be used in an application environment.

5 Conclusion

Due to the deficits of (technical) information systems which are currently in use, technologies like CORBA or DCOM have to be used. The development of such technologies had the goal of creating a manufacturer-wide standard for cooperating objects in distributed, heterogeneous environments. In general, the implementation of component-based, manufacturer-wide information systems in heterogeneous networks is possible based on such technologies. An approach for the development of technical information systems is especially necessary due to the current market situation.

A set of engineering applications is used during the product development process with the described deficits. However, the current market situation enforces the enterprises to use new methods and concepts in order to be use these systems more flexibly. Modern technologies like CORBA can directly be used only for the development of new software systems. An adjustment of available systems to these new technologies is realizable only by a complex and cost-intensive re-engineering process.

The suggested integration framework for the use of available technical information systems both, a complex re-engineering process and a completely new software life cycle process can be avoided. This approach attempts to adapt the available software system components to the new technologies. The implementation of the framework is done using by an adapter concept. Thus, the functionalities of "old" reliable software components can be used for "new" technical information systems.

For the implementation of the integration framework tools on the different layers are necessary. On the one hand, any configuration of software components on the Configuration Layer is realizable, in order to produce enterprise-specific software system configurations. On the other hand, in order to fulfill special user-specific requests, tools which enable a flexible configuration of process sequences are necessary.

References

[1] K. Bender, K. Bindbeutel, M. Glander, and A. Karcher. Framework technology for tool integration in integrated product development. In *International Conference on Manufacturing Automation (ICMA'97), Hong Kong*, pages 1001—1006, 1997.

[2] D. Chappell. *Understanding ActiveX and OLE.* Microsoft Press, 1996.

[3] M. Endig. Approach to a Framework for a Flexible Configuration of Base Functionalities in Technical Information Systems. In T. Grust, editor, *10. Workshop Grundlagen von Datenbanken*, Konstanzer Schriften in Mathematik und Informatik, Konstanz, 02.06.-05.06.1998, 1998. (In German).

[4] A. Finkelstein, J. Kramer, and B. Nuseibeh. *Software Process Modelling an Technology.* Research Studies Press Ltd., 1994.

[5] International Organization for Standardization (ISO). *Industrial automation systems and integration – Product data representation and exchange – Part 11: Description methods: The EXPRESS language reference manual*, 1994. ISO 10303-11.

[6] B. Machner and N. Lotter. Cooperate Product Development in a Network of Automobile Industrie and Supplier. In R. Anderl, J. L. Encarnação, and J. Rix, editors, *Tagungsband CAD'98 — Tele-CAD Produktentwicklung in Netzwerken*, pages 212–226, Informatik Xpress 9, 1998. (In German).

[7] F. Manola and S. Heiler. A "RISC" Object Model for Object System Interoperation: Concepts and Applications. Technical report, GTE Laboratories Incorporated, 1993.

[8] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.0*, 1997. OMG Document 97-02-25.

[9] G. Paul, F. Kreutzmann, and K.-U. Sattler. Integration Architecture for Systems in Scope of CAD/CAM. *CAD-CAM REPORT*, (3):130ff, 1996. (In German).

[10] G. Paul, K.-U. Sattler, and M. Endig. An Integration Framework for Open Tool Environments. In H. König, K. Geihs, and T. Preus, editors, *Distributed Applications and Interoperable Systems, Proceedings of International Working Conference (IFIP WG 6.1)*, pages 193–200, Chapman & Hall, 1997.

[11] W. Reisig. *Petri Nets. An Introduction*. Springer-Verlag, Berlin, 1985.

[12] K.-U. Sattler. *Tool Composition in Integrated Design Environment.* PhD thesis, Otto-von-Guericke-University Magdeburg, 1998. (In German).

[13] W. Schneider. Reference paper for an integrated development work place. http://wwwhni.uni-paderborn.de, 1997.

[14] R. Sessions. *COM and DCOM : Microsoft's vision for distributed objects*. Wiley computer publishing, 1998.

[15] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[16] A. I. Wasserman. Tool Integration in Software Engineering Environments. In F. Long, editor, *Software Engineering Environments: Proc. International Workshop on Environments*, pages 137–149, Springer–Verlag, Berlin, 1990.

Component Based Development and the Role of Workflow

Roger Tagg
Department of Information Systems, Massey University,
Palmerston North, New Zealand
Email: R.M.Tagg@massey.ac.nz

Abstract

According to some commentators, one reason for Object Oriented Development having been overtaken by Component Based Development has been the absence in the former of an effective approach to scripting the composition of objects. Workflow Management Systems offer visual scripting facilities for defining task dependency in processes involving manual participants. The same approach could be considered for composing software elements, which are effectively automatic participants, into a compound application.

1. Introduction and Background

This position paper represents a synthesis of ideas developed in earlier work on both Workflow Management and Cooperative Information Systems. The approach taken is that of a designer or user-developer, rather than that of a programmer or software engineer.

Component Based Development (CBD) is defined by Spritzer [9] as "... the practice of delivering solutions by building or buying interoperable components. Components' ability to interoperate results from their adherence to a widely accepted software infrastructure that defines a common mechanism for such bundles of functionality to work together within a common container." Spritzer points out that this definition does not depend on either of the words "object" and "application".

It is perhaps ironic that the above ideal is very similar to the claims made for the Object Oriented (OO) paradigm. So, how are components different from objects, and what has caused OO development to have been overtaken by CBD?

The lack of an effective composition mechanism for objects was highlighted some time ago by Nierstrasz, Tsichritzis et al [5]. The authors claimed that the difficulty of achieving significant levels of re-use in the OO approach is "... because object-oriented languages are still largely perceived as programming technology rather than as a software component production technology". They went on to say "Object-oriented languages typically provide a very limited binding technology for composing software at the level of expressions and statements. In order to compose objects one must program some new objects" and "Software composition is made more difficult by the ability to define rich interfaces to objects. The most successful examples of reusable components rely on the existence of fairly simple, standard interfaces". This last statement was justified by reference to the work of Wirfs-Brock and Johnson [14].

Nierstrasz et al's solution was to introduce Scripts, and their colleagues de Mey et al, as part of the Esprit II ITHACA project, proposed a visual scripting language called *Vista* [1]. Another initiative, from the same research group was the use of "Gluons", ie objects acting to mediate collaboration between heterogeneous objects [6].

As is widely known, the commercial market in CBD is dominated by two *de facto* standard approaches, namely ActiveX Controls and Java Beans, and it is doubtful whether there can be room for many more competing standards.

According to Eidahl [2], assembling a new ActiveX Control from existing Controls using Visual Basic (VB) CCE is typically a matter of pasting copies of pre-existing Controls into position on a window on a user's screen. The scripting aspect is provided by a series of VB event subroutines which respond to mouse clicks and other events. However some components, e.g. an internal interest calculation, may not have a direct user interface, and so the means of its invocation may only appear in the code. As with any CBD approach, components are assembled into a "container" which may be the whole application as defined by the user's screen interface, or just another component.

JavaBeans are an extension of Java classes with the key additional capability of "introspection". Introspection allows the JavaBean to be queried to determine its methods or properties, which is required to support cooperative working [8]. As with ActiveX Controls, a JavaBean needs to be placed within a container application; scripting is done in the Java for this container. A more recent development, Enterprise Java Beans, allows full multiuser security and resource management.

Scripting for both ActiveX Controls and JavaBeans currently requires coding, although the market is moving fast to provide development environments. Examples are Symantec's Visual Cafe Pro, IBM's Visual Age PartPaks and Sybase's PowerJ.

2. The Nature of Reusable Components

It is possible that one reason for the failure of OO approaches to achieve widespread re-use has been that they have been too theoretical. Insufficient weight has been given to consideration of what units of software form naturally-reusable chunks. While top-down requirements-driven development tends to lead to solutions appropriate to only one particular application, many of the bottom-up, programmer-oriented views of object re-use result in objects that are too small and require heavy coding to assemble.

Tagg & Freyberg [12] propose an intermediate "component engineering" approach. Clustering techniques, which take account of the natural affinity between process and data elements, are used to decompose a large application requirement into a set of "rightsized" potential components which are then "matched" against available libraries in a process which starts at the largest object frameworks and descends through software agents and compound classes down to basic classes.

Tagg & Freyberg also discuss component engineering from the point of view of bringing existing components, which may already be in operation in distributed "islands" of automation, into cooperation within a combined system.

In practice, the availability of potentially reusable components must depend on what it is worth to software entrepreneurs to package their software into reusable form. "Entrepreneurs" may be speculators hoping for a financial return, or they may work within an organisation with a remit to increase the level of re-use in order to save development effort, but the economics will be similar in both cases.

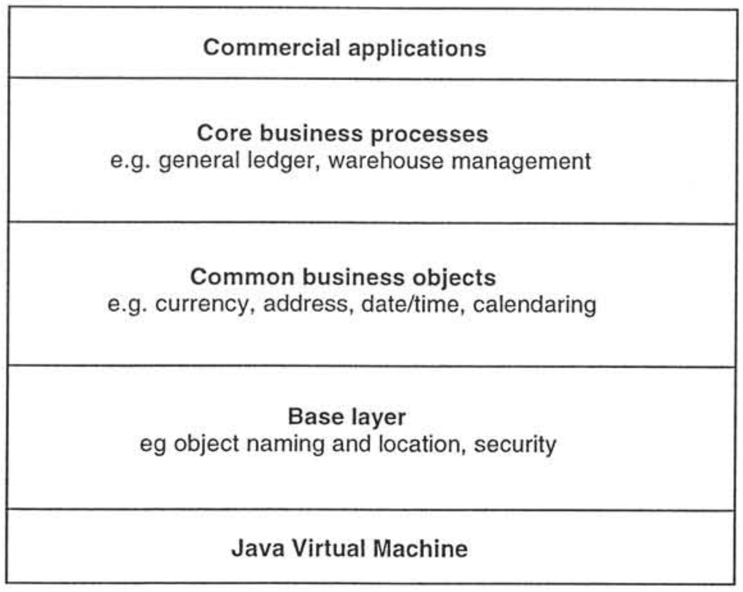

Commercial applications
Core business processes
e.g. general ledger, warehouse management
Common business objects
e.g. currency, address, date/time, calendaring
Base layer
eg object naming and location, security
Java Virtual Machine

Figure 1: IBM's San Francisco Project Framework (after [7])

The commercial world has recognised the entrepreneurial nature of component production. Examples of some of the component initiatives now being offered can be seen at a number of Web sites, eg [10, 4, 7]. The first of these sites, produced by Sun Microsystems as part of its Gamelan project, shows a number of Java Beans that have been developed and submitted to the developer.com reference library. The second shows the initiative being carried out by Sybase Inc. to encourage CODE (Cooperative Open DEvelopment) partners to contribute to a Component Gallery for components developed in PowerJ, PowerSite and PowerBuilder.

The third of these sites demonstrates the progress of the IBM San Francisco project. IBM has invested heavily in Java developments, but has made the market decision to play to its traditional strengths which lie on the server side. It has involved over a hundred software vendors in the building up of a multi-level framework of Java components. It has also built a generator to translate from Rational's ROSE (OO CASE tool) to JavaBeans. The multi-level framework is shown in Figure 1 above.

Pountain & Montgomery [7] discuss the natural affinity between the Web and related 3-tier client/server architectures with the world of components. As well as discussing the layered approach of San Francisco, the authors also point towards a more multi-dimensional arrangement of components, as shown in Fig 2 below.

To sum up, real available componentware is likely to exist in a number of formats serving a variety of application needs. The level at which these components exist will be determined largely by the economics of the cost of packaging in relation to the expected payoffs in terms of component sales revenue or development cost savings. The need remains to assemble these components. Pasting visual things on a form shows those components with which the user interacts, but without coding does not capture the dependencies in terms of control flow or data flow.

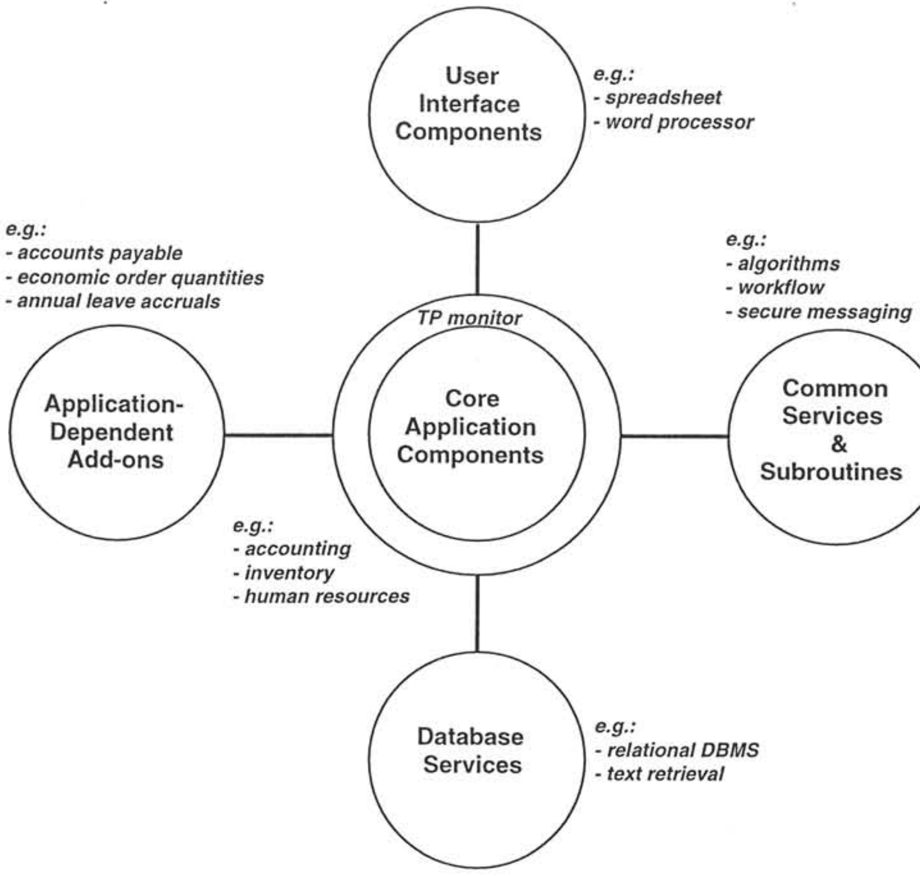


Figure 2: An Alternative Component Framework (partly derived from [7])

3. Workflow as a Potential Scripting Approach

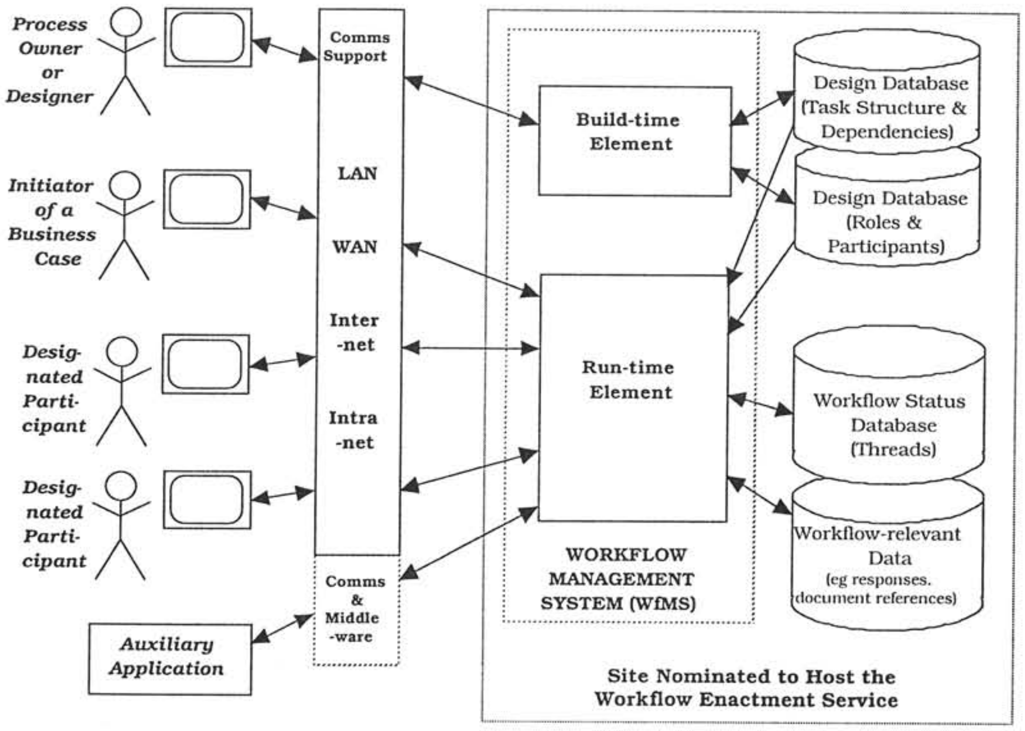
As can be seen from Figure 2, commercial component models envisage the use of a TP monitor as a container for application components to handle the so-called ACID properties [3]. Microsoft are recommending their own MTS system, while for JavaBeans CICS (IBM) or Tuxedo (BEA) are envisaged.

However TP monitors can become complex to use, and their orientation towards high-concurrency, immediate-response applications may amount to overkill for many potential applications of CBD. One alternative is to use a Workflow Management System (WfMS) as the container or scripting mechanism.

Workflow is the passing of administrative work (typically documents, but also other media including computer files) between participants in an administrative process. Workflow management is simply the control of this passing or routing activity. A Workflow Management System is "a system that completely defines, manages and executes workflow processes through the execution of software whose order of execution is driven by a computer representation of the workflow process logic" [15].

Workflow management can be exercised in both a production mode, where a business process is to be followed in many similar business cases following a repeated pattern, and in an ad hoc mode, where the process is defined newly for each business case.

Participants in most applications of workflow management are typically assumed to be humans, but most WfMS allow for automatic invocation of "auxiliary applications" which act as if they were participants. This is not the same as the manual use by a human participant of a desktop tool (such as a spreadsheet) or even of a transaction processing system, to support their execution of their assigned tasks. A typical WfMS system structure is shown in Figure 3.



Process Owner or Designer
Initiator of a Business Case
Desig-nated Parti-cipant
Desig-nated Parti-cipant
Auxiliary Application
Comms Support
LAN
WAN
Inter-net
Intra-net
Comms & Middle-ware
Build-time Element
Run-time Element
WORKFLOW MANAGEMENT SYSTEM (WfMS)
Design Database (Task Structure & Dependencies)
Design Database (Roles & Participants)
Workflow Status Database (Threads)
Workflow-relevant Data (eg responses. document references)
Site Nominated to Host the Workflow Enactment Service

Figure 3: Structure of a Typical Workflow Management System

Figure 3 suggests a workflow application where the participants are predominantly human, and automated participants (components) are the exception. In a CBISE environment, however, the balance would be the other way round. The dashed extension of the "Comms Support" bar would assume greater importance, with Middleware such as 2-way Messaging APIs.

The build-time element of most commercial WfMS software packages incorporate a visual tool which is used by a process owner, or his/her delegated designer, to specify a task dependency structure for a single or repeated business case. Figure 4 shows a typical extract from one possible visual model [13], in which workflow tasks are linked by dependency rules and embedded in supertasks.

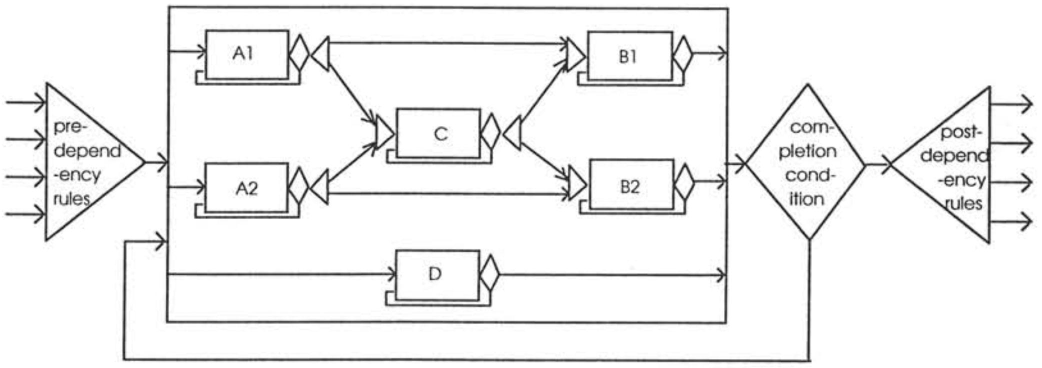


Figure 4: Example Extract from a Visual Model of a Workflow Task Dependency Structure

The enveloping box represents a task or subworkflow which consists of the subtasks A1 thru D. In the CBISE environment, the enveloping box is the container component, the smaller boxes the components being composed, and the intervening arrows and shapes are the script. The triangle to the left of a box indicates the synchronization or JOIN logic, ie AND, OR or combination; its absence implies simple sequence. Similarly the triangle to the right of a box indicates the "what next", "emission" or SPLIT logic, ie AND, OR (with specified predicate) or combination; again absence implies sequence. The diamond shows repetition of a task until a completion condition is met. Within the large box, the arrows starting from the left hand side lead to all the subtasks which can start as soon as the parent task is started; similarly the parent task is complete when all the subtasks with arrows pointing to the right hand side are completed. This diagram is intended to be equivalent to a multi-level predicate Petri Net.

The WfMS will also include a mechanism by which data can be passed between participants. In some systems data may be passed physically in data containers; in others a shared file system or database may be assumed, and only file locator references passed. This mechanism is also required for parameter passing in the CBD case.

Some sophisticated WfMS support Transactional Workflows, where the ACID properties [3] are enforced across the "container application" which is the WfMS. The thing to remember about Transactional Workflow is that, if human participants (who perform their tasks manually and in their own time) are involved, then any rollback of a potentially inconsistent system state will also have to be manual and in the participants' own time! For this reason, Transactional Workflows appear more appropriate where the participants are entirely or nearly all automatic.

The advantages of using a WfMS in this way are:
a) providing a clearer presentation of the script logic for discussion by designers and process owners, and
b) handling direct interaction with human users acting as manual components.
Even if not used directly, design models prepared using the WfMS's build-time tool could, with use of a generator, be used to produce the script code for an object-oriented tool such as Rational ROSE.

This use of a WfMS is probably best limited to applications where there is a mixture of manual and computerised components. WfMS run-time elements are also typically message-oriented and might be inappropriate for a highly automated real time system.

4. Workflow as a Component in its own Right

A WfMS can be looked on in another way in connection with CBD, namely as an invokable service for orchestrating the human participants involved in a computer system.

Suitably designed, a workflow component could be provided as an add-on to any TP system, allowing control of the status of user replies, so that the system administrator can be made aware if any human participant is holding up the system. The software vendor SAP has taken this approach by offering its Business Workflow product.

In production workflows, a workflow component could offer the appropriate user interfaces for initial design and modification of the task dependencies; for user worklist processing; for notifying the process owner of performance status; for over-riding the normal task sequence to cope with exceptions and crises; and for system manager monitoring and statistics collection.

Ad hoc workflow could be included as a tool which could be visually pasted onto a user's desktop so that control could be applied to a one-off set of tasks.

A final application is where a process needs to be enforced across a set of participants from independent organisations. In such a case the role of workflow controller might best be handled as an independent one, and could be delegated to a third-party software agent.

5. Conclusion

Most Workflow Management Systems incorporate a visual tool for task dependency definition, which could provide the scripting which is a necessary feature of any component-based approach. The advantage of the WfMS approach to scripting is that the logic of the script can be clearly seen by the designer and users, whereas when using programming languages it is lost in the code. With Visual Basic, for example, the script logic is hidden in the interelationships of the event subroutines. WfMS might also be a good solution where control of manual human activities is important.

However current commercial trends suggest that the role of scripting will be taken by the use of languages such as Java, VB, Delphi and PowerBuilder within the container application in which the components are to be assembled, although the code could be generated from a design-level CASE tool such as Rational ROSE.

It is perhaps more likely that Workflow will find its role in Component Based Development as a component in its own right.

References

[1] de Mey, V., Junod, B., Renfer, S., Stadelmann, M. and Simitsek, I. (1991). The Implementation of Vista - a Visual Scripting Tool. In *Object Composition*, Annual Report of Centre Universitaire d'Informatique, Université de Génève, pp 31-56.

[2] Eidahl, L. (1997). *Visual Basic 5 Control Creation Starter Kit.*. Indianapolis, QUE Corporation.

[3] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. San Mateo, CA. Morgan Kaufmann.

[4] IBM (1998). San Francisco Project Release 1.2. http://www.ibm.com/Java/Sanfrancisco

[5] Nierstrasz, O., Tsichritzis, D., de Mey, V. and Stadelmann, M. (1991). Objects + Scripts = Applications. In *Object Composition*, Annual Report of Centre Universitaire d'Informatique, Université de Génève, pp 11-29.

[6] Pintado, X. and Junod, B (1992). Gluons: Support for Software Component Cooperation. In *Object Frameworks*, Annual Report of Centre Universitaire d'Informatique, Université de Génève, pp 311-329.

[7] Pountain, R. and Montgomery, J (1997). Web Components. *Byte Magazine*. August, pp 56-68.

[8] Rahmel, D. (1997). Java Beans in Action. *Internet Systems*, May pp S9-S13.

[9] Spritzer, T. (1997). Component Architectures. *DBMS Magazine*, September, pp 56-66

[10] Sun Microsystems (1998). Gamelan: The Official Directory. http://www.developer.com/directories/pages/dir.java.javabean.general.html

[11] Sybase Inc (1998). Sybase Component Gallery for Powersoft Tools. http://www.sybase.com/partners/gallery/index.htm

[12] Tagg, R. and Freyberg, C. (1997*). Designing Distributed and Cooperative Information Systems*, London, ITP Computer Press.

[13] Tagg, R (1998). *Process Modelling - History and Future*, Technical Report no. 4/98, Department of Information Systems, Massey University, Palmerston North, New Zealand.

[14] Wirfs-Brock, R. and Johnson, R. (1990). Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, Vol 33 No 9 (Sept), pp 104-124

[15] Workflow Management Coalition (1996). *The Workflow Management Coalition Specification: Terminology and Glossary* http://www.aiai.ed.ac.uk/WfMC/DOCS/glossary/glossary_ToC.html

Synergies in a Coupled Workflow and Component-Oriented System

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

Abstract
Current workflow systems allow often only modeling and execution of business processes. Application programs used in activities are integrated as black boxes. These programs can neither be created nor modified by business specialists. One must be a program developer. The use of composite applications in workflow systems can eliminate these drawbacks. These applications are built of components and are modeled in a way similar to workflows. A coupling between a workflow and a component-oriented system has advantages compared to their combination. Context-sensitive applications and more flexible workflow models are becoming possible.

Keywords: *business objects, components, component-oriented system, composite applications, workflow management system*

1. Introduction

To provide a better understanding of this paper, some basic terms of workflow management and component-oriented systems will be introduced.

A *workflow management system* is a software system for coordination and cooperative execution of work according to a specification (see also [14, 1, 2]). The use of a workflow system can be divided into a modeling phase, an execution phase, and an analyzing phase.

In the modeling phase, the organizational structure of an enterprise or administration and the order of all business procedures are modeled as *business processes*. The whole work is splitted into several steps, which are connected by a relationship. This chronological order is forming the process. A process step, also called *activity*, is a piece of continuous work performed by one person or machine—the *actor*. Each activity is associated with consumed and produced work objects (i.e., data and documents) and with organizational (e.g., a *role*) and technical resources (e.g., an application program) that are necessary for performance. The modeled business process is specified as a *workflow scheme* (often called a *workflow model*) in a formal textual or visual workflow language.

In the execution phase, these schemes are used for controling, supervising and recording of activities performed. The executable instance of a workflow scheme is called a *workflow instance*. A *workflow* represents a running workflow instance in the enterprise. The actor of an activity can use interactive application programs to fulfill 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.

In the analyzing phase, recorded data are evaluated. The results extracted serve as a basis for improvement of workflow schemes. Passing all phases several times in a cycle, business processes are optimized little by little.

A *component-oriented system* [11] is an architecture or an implementation for a software system providing infrastructure for components. An implementation consists of an application development environment and a runtime environment for components. A *component-based system* [13] uses components only during development. In the runtime environment (i.e., the compiled application), components do not exist anymore. They are hidden from the user. The term *composite application* defines an application that is built using components as building blocks. The term is used to distinguish between composite and monolithic applications that are built conventionally.

In the following section, some problems will be described resulting from the restriction and deficient flexibility of monolithic applications that are used in workflow systems. Subsequently, a design of a system is introduced that couples a workflow and a component-oriented system. This coupled system can solve those problems. After that, the additional benefit of a tight coupling compared to a pure combination of individual systems is explained. The paper ends with a description of related architectures and a conclusion.

2. Problem Description

The engagement of current workflow systems reveals some problems linked with creation, use, and adaptation of applications. The use of component-oriented systems as application programs in activities can eliminate or at least reduce these problems [8].

2.1 Creation of Application Programs

Workflow systems support the (partial) automated handling of business processes. However, support is restricted to the process-oriented part of a business case. The function-oriented part, which comprises applications realizing business functions, is usually supposed to be given. These applications are executed in activities. If there is no suitable application for the activity while implementing a business process, the application program has to be realized in a conventional way. For example, an order is given to an external or internal software vendor. A workflow system offers usually no support for realization of application programs. Traditionally, the realization of workflows and applications are kept .

2.2 Adaptability Problem in Activities

Using a modeling tool, a user can easily adapt a workflow to changing business situations. However, adaptability ends at the border of activities. Programs executed in activities can only be adapted by software developers. This can be expensive and time-consuming. Component-oriented systems allow the user to adapt composite applications. Indeed, this feature is not supported very well in current systems [5].

2.3 Heterogenous System Environments

Workflow systems are often used in a very heterogenous infrastructure of computer equipment, which is grown by and by. Workflows are usually based on a platform independent specification. Nevertheless, the invoked application are often platform dependent. This problem can be solved if platform independent components are used to build composite applications.

2.4 Incomplete Reuse of Workflows

Reusing a workflow in a different enterprise requires the workflow to be adaptable by the user. Additionally, each application have to be executable on each platform used enterprise-wide. As a consequence, without the availability of adaptations and of platform independent applications, reuse of workflows is not really possible.

2.5 Unspecific Applications

In particular, if workflows are executed often, workflow systems are very useful. Therefore, applications should be accommodated very well to their operating area to achieve high productivity. However, creation of customized applications is usually expensive. Component-oriented systems allow to create and modify activity-specific programs simply and quickly. Moreover, composite applications can easily be customized to specific application domains.

3. Architecture of a Coupled System

Workflow systems and component-oriented systems have similar assignments and use similar technical approaches to facilitate development of applications. However, they differ in application domain and in magnitude of application. Workflow systems control enterprise-wide applications consisting of activities. Component-oriented systems control local applications consisting of software building blocks. This similarity can be explored for a two-way exchange of architectural concepts. A design of an architecture [9] coupling both systems is introduced in this section. This proposed architecture can solve the problems mentioned in the previous section. The specific architectures of the individual systems can be found in [2, 6]. The design concentrates on the question how to create activity-specific applications. Legacy applications can be used in the workflow system, as usual.

In the coupled system, the workflow system is responsible for the process-oriented part of the business process and coordinates the order of activities in a typically distributed, heterogenous computer environment. The component-oriented system is responsible for the function-oriented part inside an activity and coordinates working steps. The component-oriented system serves not only for execution but also for creation and modification of application programs in activities. Besides the coupling in the execution phase, there exists also a coupling in the modeling phase.

The principle for modeling workflows is adopted and applied to composite applications. By analogy with workflows, applications are divided in already existing building blocks (i.e., the function part). The blocks are connected by a modeled control flow (i.e., the process part). This approach is better than the more conventional approach to code an application in a programming language . Composite applications can be created and modified by the user himself more easily. Programming code, however, is only understood by system developers.

From the workflow system's point of view, an activity has an inner structure. Instead of being a black box, it consists of working steps connected by a modeled control flow. One could say, the component-oriented system can be seen as a simplified and specialized 'mini workflow system' inside activities. In some aspects, it provides the same functionality as a workflow system.

The component-oriented system is based on an architecture that is a variant of the compound document concept [6] and is adapted to workflow management domain. The user arranges data that represent business semantics into a uniform collection, called *composite application*. The arrangement of data is done similar to that in a compound document.

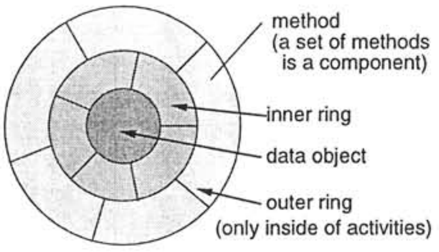


Figure 1: The structure of a business object

A composite application consists of *business objects* [6, 5] representing entities in a business domain. Each object has a specific meaning commonly used by business specialists. An example is the business object *Customer* in a composite application *Tender Management*. Business objects are divided in pure data objects and atomic or compound component objects.

The data itself or links to it are stored in *data objects*. Data objects establish a data integration level providing a uniform view on workflow-relevant data. In the example, the business object *Customer* consists of a data object that is a link to a tuple in the relation *Customers* stored in a relational database management system.

Atomic components offer common usable functions restricted exactly to one type of data object. Typical functions are creating, displaying, or editing of data. As a general rule, atomic components can only be configured but not created by users. The development of atomic components should be taken over by an independent software vendor. Thus, the vendor can hide its expert knowledge from the user. In the business object *Customer*, an atomic component can be used, for example, to exchange one tuple against another tuple of the same relation. *Compound* *components* are used to embed other composite applications.

Components of the inner ring (see Figure 1) are usually associated with a data object during the whole life cycle of a composite application. They shall be used to implement common functions. *Components of the outer ring* only exist inside of activities. The components are associated with the data objects of a composite application. They shall be used to realize activity-specific functions on certain data objects.

The interactive or automatic creation or modification of any data objects through a component function is called a *working step*. Inside an activity, a control flow is defined on working steps.

A workflow scheme describes a workflow separated in different aspects [1]. Thus, modifications in one aspect do not propagate inevitably into other aspects. Some independence can be achieved. This principle can be applied to modeling of composite applications, too. Because some aspects are modeled in both systems (i.e., control and data flow), it suggests itself to couple both models in these aspects.

One or more composite applications can be associated with activities. A control flow is defined for each composite application used in an activity. Additionally, it is specified in which state a composite application must be if the activity shall be finished.

The user is able to modify a composite application on two levels. On the first level of modification, the user needs no programming skill. He or she can modify data objects by the means of components. Business objects can be configured. Even more, the arrangement of business objects in the composite application can be changed by adding new ones or deleting old ones. On the second modification level, the user can modify the control flow between business objects. Calls to component functions (working steps) can be added or deleted. Though, depending on the scope of modification, he or she needs programming skill in the (scripting) language the control flow is modeled with.

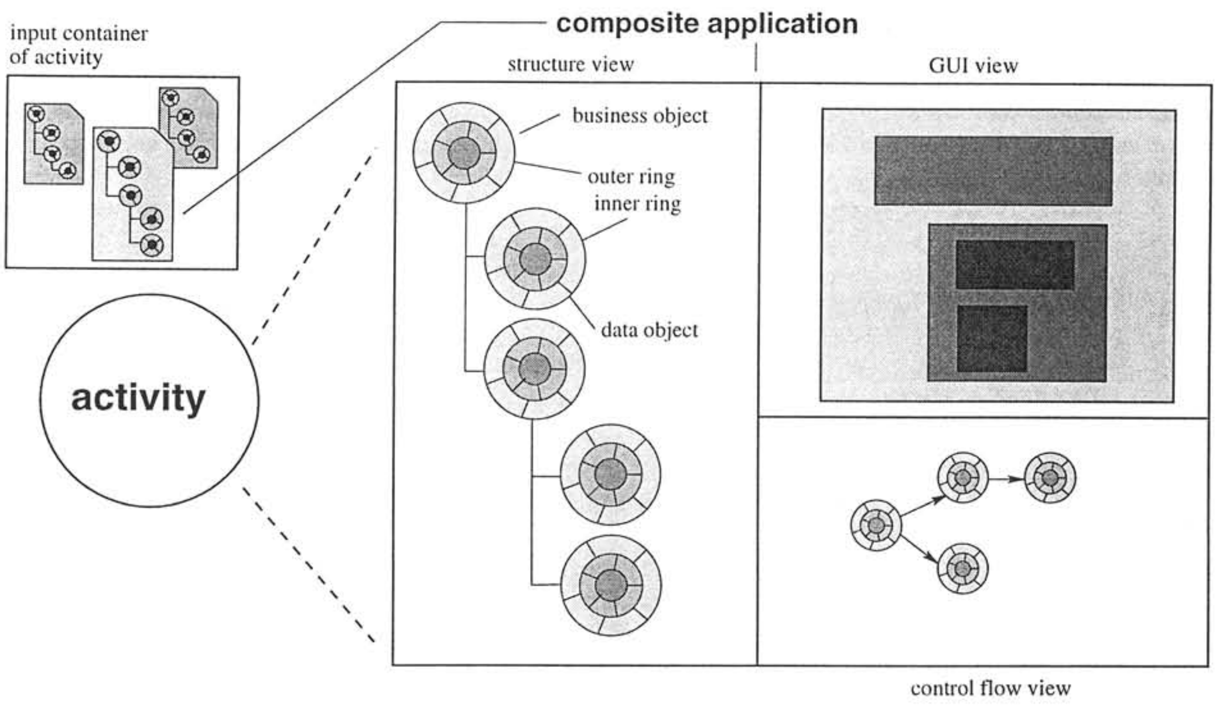


Figure 2: Sketch of a composite application in a workflow activity

An activity with a composite application is shown in Figure 2. The workflow system puts several composite application in the input container of the activity. These composite applications can be used in the activity. The structure of a composite application comprises several business objects. Inside of activity, business objects are complemented with an additional ring of application-specific components. The order of working steps on business objects is described by a specified control flow. This control flow is used for guiding the user during his or her work.

This coupled system architecture enables the end user to create and modify whole business processes. He or she can adapt the workflow model as well as the model for the composite application according to changing business situations.

4. Synergy Effects through Coupling

The use of a coupled system has advantages compared with the combined (i.e., independent) use of individual systems. Three advantages are described in this section. The workflow system can automatically adapt composite applications according to their use. The coupling supports an enhanced communication method using events, which can be used to realize more flexible workflow control constructs. Common modeling aspects can be used to modify the workflow scheme more easily.

4.1 Flexible Automatic Adaptations

An application does usually not change its behavior or its outlook depending on the activity in which the application is invoked. Only input data differ in activities. If a user changes the configuration and so the behavior or outlook of an application, this adaptation can be stored either globally for all users or locally for himself. Different scopes of adaptation are not possible (e.g., a scope referring to the use). Although this problem is independent of the use of a component-oriented system, it is intensified by using composite applications. These applications consist of many, well configurable components. This is in contrast to current workflow systems using few, less configurable application programs.

The introduction of a *process context* allows flexible definition of adaptation scope based on the definition of organizational structure and process organization in a workflow system. Thus, an application can automatically be adapted according to its specific use in the activity. From the user's point of view, applications have a 'process consciousness': They 'know' in which context they are used and adapt automatically. Depending on its use, an application can behave and look different. Thus, the realization of *context-sensitive applications* will be possible.

A composite application allows several adaptations, for instance, setting attribute values of a component or changing the set of accessible functions. This is called *configuration*. It is possible to fade out certain data objects or components or to change the associated component of a certain data type.

The modeling tool of a workflow system must allow to define contexts in the definition of organizational structure and process organization. A process context consists of a set of activities that are associated semantically. A context is not restricted to activities of one process. It can comprise activities of several processes. For example, the activities are related because of their common application domain or use. A context can consist of activities performed by certain persons or roles, too. The affiliation to a process context can also workflow-relevant application data. A workflow system can manage several contexts. An activity can be a member of one or more contexts. Therefore, a relationship on contexts must exist that defines an order in which the adaptations of the composite applications have to be done. A specific part of the workflow system is responsible for the adaptation of a composite application. Before an activity is started, the adaptation has to be done in the activity. In addition, a database for configuration data is necessary.

Process contexts can be applied diversely. For instance, a context can supply pre-selected data values in input masks or it can restrict the history of input values to input made in earlier workflow instances of the same workflow scheme. By defining a workflow as *uses-foreign-language*, the user interface can be configured automatically for a certain language by the means of a context. Composite applications can show different views on included or referenced data the person or role (e.g., an expert) who is working with the application. Certain properties (e.g., protocoling all actions) can flexibly and automatically be switched on or off via the context.

4.2 More Flexible Control in Workflows

Typically, an activity is integrated in a workflow system using a model called *black box integration*. The model is restricted to an application, supplying or transferring parametric data, and waiting for completion of the application. An application can be wrapped with an *envelope* [12] to make the invocation interface of application compatible to the interface of the workflow system. There is no need for modifying source code. From the workflow modeler's point of view, a conventional application is built monolithically—even if source code exists. Internal states or occurred events of an application cannot be made public without having high programming skill.

In composite applications, the user can modify some aspects, in particular control flow and communication. The boundary between black box and white box integration . So to say, a composite application consists of 'black' components and 'white' control flow.

Cooperation between the workflow and component-oriented system can be facilitated through the mutual exchange of events. This communication method can be used for more fine-grained synchronization or for externalization of application states. Therefore, descriptive control flow constructs [2] can be implemented more easily. That helps to model the control flow of a workflow more accurately.

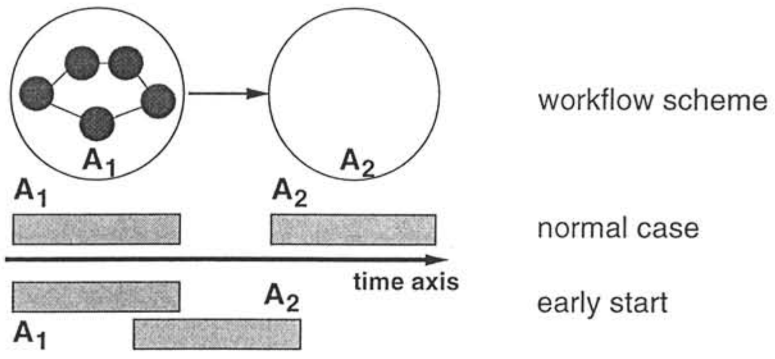


Figure 3: Early start of A_2

For example, the control construct *early-start* allows to start a succeeding activity A_2 before the preceeding activity A_1 is finished. In some situations, this construct can increase parallelism. A_1 possesses an inner structure—the process consisting of working steps. If a certain state in this inner process is reached, the actor of A_2 can start A_2 earlier by its request. If the workflow system should support this, it must be informed about the progress of processing in the activity.

Flexible communication methods are important for the interoperation between workflow and component-oriented system. For example, the realization of interrupting, aborting, or delegating activities is easier. *Delegation* is the migration of an activity to another user. In current workflow systems, delegation can often only be done when the activity has not been started. Delegating while the activity is being performed requires sophisticated communication methods. Thus, in a coupled system, individual working steps can be executed by another user.

The communication methods can also be used to realize a more fine-grained version management. In long-living activities, several versions can be delivered up the workflow system which can associate these versions directly with the workflow. If an application had used an individual version management inside the activity, it should had associated the versions to the workflow explicitly.

4.3 Shifting of Application Parts

Whenever business increases, new workers are engaged. If the new workers have the same assignments as the old ones, the workflow system can allocate the work load to the workers fairly. If the assignment is different, the old worker's assignment has to be divided and a part has to be assigned to the new worker. This partition of work assignments requires that an activity must be divided, too. If there is an activity-specific application used, it has also to be splitted and parts have to be shifted into a new application of a new activity.

Shifting is a modification in the workflow scheme in which parts of an application are separated and merged into another application. In the introduced architecture, an application part consists of working steps associated with control flow.

The architecture supports this kind of modification well because the coupled modeling in the modeling phase can be used. A composite application is structured in a collection of business objects and control flow. These structure allows identification of application parts. Therefore, even unskilled user can separate parts and insert them elsewhere in other applications. The workflow system facilitates the performance of shifting because it can use information about the whole process to reveal consistency problems. It can point the modeler to activities or applications in which 'holes' have occurred. The modeler has to 'fill' these holes—by doing some further modifications. In the case of inserting, the workflow system can also warn of problems, like missing data objects.

5. Related Work

Besides the introduced architecture of a coupled system, some more architectures exist that support inner structure in activities. These are the integration approach and the distributed workflow system approach.

In the integration approach, the inner structure is modeled as a subordinated workflow. Each activity has exactly one component associated [3, 10]. This approach rises sharply the number of activities that have to be performed by the workflow engine. It is not likely that a centralized workflow engine can cope with this additional load and deliver the required performance for interactive applications. Therefore, scalability of this approach is bad.

As an alternative, workflow systems can be realized in several more decentralized ways [4]. An architecture can consist of several interoperating workflow engines whose realizations are central. The coupled system is also an architecture in which workflow systems interoperate. However, the systems are not equivalent but specialized for a certain purpose. Another architecture, a fully distributed workflow system [7], has no centralized engine anymore. Each activity has its own part of control functions. The distributed architecture matches the inherent distributed character of the execution phase very well. But the modeling and analyzing phase have a inherent centralized character. The realization of these phases is becoming more expensive.

To be comparable with the introduced system architecture, all approaches must be enhanced with methods to adapt a workflow. This can also be expensive if the workflow system is mainly distributed.

6. Conclusion

The use of a component-oriented system for creation of applications, which are invoked in workflow activities, increases flexibility of a workflow system. The basic principle user's self-organization in business processes is going to be real. The end user can create business-wide applications and he or she can modify the whole business process according to changes in business situations. The end user is not restricted to modifications of the pure workflow model anymore (i.e., the workflow without the applications). The user's working method can be adapted and arranged to his or her own needs more efficiently. The coupling instead of simply combining those two systems increases the benefit even more. Synergy effects are becoming possible that are not possible in individual systems.

The coupling and the synergy show diverse forms: A process context can be used to adapt composite applications to their specific application domain . According to its intented purpose, an application can react differently. Events occurring in composite applications can trigger events on the workflow level and vice versa. Synchronization mechanisms are becoming more flexible. Thus, real business processes can be modeled more accurately in workflow systems. Certain aspects in composite applications and in workflows can be modeled in common. This simplifies consistent modifications, like shifting of application parts.

References

[1] Stefan Jablonski. Workflow-Management-Systeme: Motivation, Modellierung, Architektur. *Informatik Spektrum*, 18:13–24, 1995.

[2] Stefan Jablonski, Markus Böhm, and Wolfgang Schulze. *Workflow-Management: Entwicklung von Anwendungen und Systemen.* dpunkt Verlag, 1997.

[3] Frank Leymann. Workflows Make Objects Really Useful. In *Proceedings of the 6th International Workshop on High Performance Transaction Systems (HPTS)*, September 1995. http://www3.hursley.ibm.com/hpts95/proc95.htm.

[4] J. A. Miller, A. P. Sheth, K. J. Kochut, and X. Wang. Corba–based Run–time Architecture for Workflow Management Systems. *Journal of Database Management*, 7, 1996. Special Issue on Multidatabase.

[5] OMG. Common Facility RFP-4 — Common Business Objects and Business Object Facility. Technical report, Object Management Group, 1996.

[6] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. Wiley & Sons, New York, NY [u.a.], 1996.

[7] Rainer Schmidt. Component-based Systems, Composite Applications and Workflow Management. In *Proceedings of Foundations of Component-based Systems Workshop*, pages 206–214, September 1997.

[8] Stefan Schreyjak. Coupling of Workflow and Component-oriented Systems. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the* 2^{nd} *International Workshop on Component-Oriented Programming (WCOP'97)*, TUCS General Publications No 5, pages 77–86. Turku Centre for Computer Science, September 1997.

[9] Stefan Schreyjak. Using Components in Workflow Activities. In Jeff Sutherland and Dilip Patel, editors, *Proceedings of the* 2^{nd} *and* 3^{rd} *International Workshop on Business Objects (at OOPSLA)*, Springer Verlag, 1998. to appear.

[10] Stefan Schreyjak and Hubert Bildstein. *Beschreibung des prototypisch implementieren Workflowsystems Surro*. Universität Stuttgart, Software–Labor. Fakultätsbericht Nr. 1996/19.

[11] Clemens Szyperski. Independently Extensible Systems — Software Engineering Potential and Challenges. In *Australian Computer Science Communications*, 18(1):203–212, February 1996.

[12] Giuseppe Valetto and Gail E. Kaiser. Enveloping Sophisticated Tools into Process-centered Environments. *Journal of Automated Software Engineering*, 3:309–345, 1996.

[13] Peter Wegener. Concepts and Paradigms of Object–oriented Programming. *OOPS Messenger*, 1(1):8–87, 1990.

[14] Workflow Management Coalition. *The Workflow Referenz Model*. Technical report, November 1994. Document Number WFMC-TC00-1003, Issue 1.1, http://www.wfmc.org/wfmc/DOCS/refmodel/rmv1-16.html.

Considerations on the Nature of Dynamic Structures in Component Schemas

Sari Hakkarainen
Swedish Institute for Systems Development, SISU
Electrum 212
S-164 40 KISTA, Sweden
e-mail: sari@sisu.se

Abstract

In the development of component-based information systems, existing specification methods in traditional information systems development and requirements engineering can be applied. Our approach focuses on the dynamic aspects of such specifications. The problems addressed in this position paper are twofold. First, the metadata suggested so far to describe the structure and content of associated resources do not support description of dynamics of components. Second, it is difficult to establish correspondences between dynamic parts of schemas in information modelling. The characteristics of events are analysed and two problems are identified as causes to the difficulties in comparison of dynamic schema structures.

1. Introduction

The development and maintenance of internet services based on new technology would benefit of applying existing traditional methods of information systems development and requirements engineering. The hard problems in schema integration overlap with problems in interoperability of loosely coupled configurations of components, characterised as heterogeneity of information and services. The position taken here is that schema comparison techniques can provide means to support the development, selection and exchange of dynamic information in component-based information systems.

In order to specify internet services supporting user interaction and pluggable dynamic components, a notion of events and interconnections between the dynamic aspects of the information has to be established. So far, the information obtained on WWW has been regarded as basically static. However new technologies supporting embedded (Dynamic HTML) or pluggable (Java scripts) software components provide the possibility of viewing at the information as changing over time and depending on the context. However, the metadata [12] suggested so far to describe the structure and content of resources on the internet, (RDF, and PICS, as well as Dublin Core [2] and URCs [11]), do not support description of dynamic components. It is not sufficient to view the content of components in the same way as URL, title, or paragraph, nor as image, sound, or animation. Similarly, as the need for a special kind of meta-concept describing an anchor has been recognised, also the particular temporal semantics of components should be specified.

The problems in information modelling concerning the comparison of dynamic schema structures are difficult to handle at the specification level [6]. There is no existing method nor theory directly applicable to solve the particular problems which arise in comparison of dynamic schema parts. However as trying to find a solution to the problem [7] it becomes evident that there is no adequate analysis of what the problems exactly are, found in the related literature [8, 5, 9, and 10]. In this work two types of problems are analysed and identified as characteristics of dynamic component specifications and causes to the difficulties in developing a technique for comparison of dynamic schema structures. The problems consider implicit properties, and temporal dependency.

Events and dynamic rules have implicit internal properties such as tense and sequence. These properties are not accessible on the specification level where the types of events are defined. There is no explicit notion of time nor causal order in a declaration or rules of an event. The absence of reference to these internal properties is identified and referred to as a *problem of implicit properties*. A simulation of occurrences of evoked events reveals the existence of these properties. In order to compare dynamic schema structures, the implicit properties should be embedded in the comparison mechanism in order to perform a meaningful comparison.

It is not sufficient to compare an event with another as a whole. The analysis of temporal nature of event constitutes reveals *a problem of temporal dependency* when two events are compared. The truth values of the different parts of an event are based on different system states (information bases). When analysing the semantics of an event, it becomes evident that the truth values of the rules in the different event constituents are known in a specific sequence and at different points of time. Any comparison algorithm analysing correspondences between dynamic schema structures should consider the semantics of tense inherent among the constituents of the structure.

2. Related work

An event specification intends to describe possible changes in the state of an associated information base. An event can be regarded as a concept with specific properties. An event head denotes associated attributes, a condition part determines when the event is valid, and a conclusion part specifies the intended effect. When it comes to the conclusion part the focus in existing modelling methodologies has shifted from specifying how the effect can be materialised [1] to what is effected [3]. The modelling languages are becoming more implementation independent and task-oriented, yet supporting adequate description of dynamics.

There is no directly applicable existing method nor theory known to the author to solve the particular problems which arise in comparison of dynamic schema parts. The schema structures describing the dynamics of a schema have implicit properties inherent, such as tense, and sequence. In this work a notion of temporal context is introduced in order to make part of that semantics explicit. In the NATURE process model [8] the notion of context has a prominent role. A process is defined as a set of actions performing a transformation on a product (static schema) part and materialising decisions in the context a given situation in a product. A context is an objectified association between a particular situation and the decision which can be taken on it.

There are some promising approaches to introduce temporal aspects to databases [9] and systems development [10]. However, the pragmatic approaches to temporal aspects have not been able to penetrate to IS modelling practitioners. A notion of time is introduced as an explicit property of concepts and rules enabling historical analysis of a schema both in the ORES and TEMPORA methods. The technique of objectified relationships, enabling time stamp of relationships, has become an accepted method of collecting historical data. In this work, the aim is to be able to handle the temporal semantics of the constitutes of events at the specification level rather than to collect historical data over their occurrences at the instansiation level. Hence, there is no need to introduce explicit notion of time to component schemas.

3. Implicit properties of dynamic schema structures

In order to discover means to support the schema comparison process of dynamic schema parts on specification level we have studied the particular characteristics of events apparent in the instansiation level. An event (type and occurrence) is constituted of smaller parts with specific semantics. We say, an *event constituent* is one of event head, condition or conclusion. A dynamic rule groups conditions and conclusions into a meaningful unit. It is an objectification of the many to many relationship between a set of conditions and a set of conclusions enabling the connection of the relationship to a specific event. Thus, a dynamic rule in itself is not an event constituent. Based on the reduction of events into event constituents, a meta-model of event occurrences is proposed as depicted in figure 1. It is intended to cover the temporal semantics of the mutual relationships between the constituents of event occurrences specifying the parts of the system state concerned.

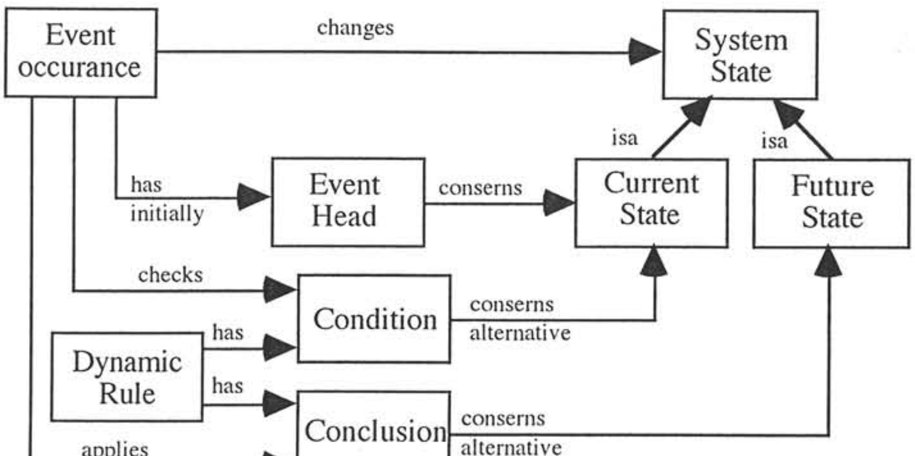


Figure 1. An event meta-model at the instansiation level describing the temporal semantics of event occurrences.

A simulation of schema execution is often used as means of schema validation. Validation concerns locally specified schemas, and event specifications in a global schema after a manual comparison and integration of dynamic schema parts in local schemas. In a simulation the schema is instansiated with variable information bases and the events are evoked in variant sequence. The purpose is to check that the conditions hold as desired before the event is evoked and that the system state is the desired after the event execution. The validation of a global schema is made in order to rule out any interferences caused by conflicting conditions or conclusions.

Consider the following example. An event, eventX is judged to have similar structure, conditions and conclusions as another event, eventY. Therefore, eventX and eventY are integrated in a global schema. The explicitly specified semantics of eventX has a prerequisite that a condition, C1, defined as: aState(anEntity), holds. However, the semantics of eventY implicitly has a prerequisite that C1 does not hold. In the local schema specifying EventY, aState is not recognised as a possible state of the type of entity, which anEntity instansiates. The particular type of entity state is not even declared in the static part of the local schema. A simulation of execution of the global schema detects this interference in that eventY does not perform as expected. For example, in the case eventX has been evoked before it, i.e., C1 holds after eventX in the system state, C1 also holds at the evoking time of eventY.

The above case illustrates how an analysis of events on the instansiation level, where constants are assigned to variables, supports an analysis on the specification level, where types are specified using variables, in making explicit the sequence and tense inherent in event declarations. The simulation technique is however time consuming. The population of cases and the sample of possible order of event execution has to be carefully selected. The selection prerequisite domain knowledge as well as testing expertise. Also the analysis of the output of a simulation process is performed manually. Simulation can be motivated when specifications have reached a final state and released after the validation. However, in order to compare dynamic schema structures, the implicit properties, arising from the absence of reference to sequence and time, should be embedded in a comparison mechanism. The event constituents need to be analysed independently and their temporal roles should be specified.

4. Temporal dependency of dynamic schema structures

Based on the above analysis of event occurrences, a notion of tense of dynamic schema structures is introduced below. An event occurrence at the instansiation level has an event (type) at the specification level. An event head is specified for each event and instansiated at the time of evoking an event. For each event we assume the life time of all instansiations of that particular type are known. Thus the time of event head instantiation is known as the point of time of execution of the event occurrence. The truth value of an event head is known as the point of time of execution of the event occurrence. The time when the truth value of the other constitutes of an event specification; event head, dynamic rule, condition, and conclusion is known, is an interval of time relative to the point of time of execution of the event occurrence. We say, *tense* of an event constituent, C, of an event Ev is described by a *time referent*, t, denoting when the truth value of C is known and relative to the virtual evoking time, t, of Ev.

A virtually fixed point of time at the specification level is introduced. It refers to the evoking time (of event occurrences) of an event and positions the event constitutes in a virtual time axis. A vocabulary consisting of three time referents; at, before, and after, is used to specify relative time in that time axis. A *time reference* is one of at, before, or after. *At* is a point of time, virtually fixed to a possible evoking time of an event, *Before* is an interval of time partially specified by at, where at closes the before interval and the starting point of before is not known. *After* is an interval of time partially specified by at, where at starts the after interval and the ending point of after is not known.

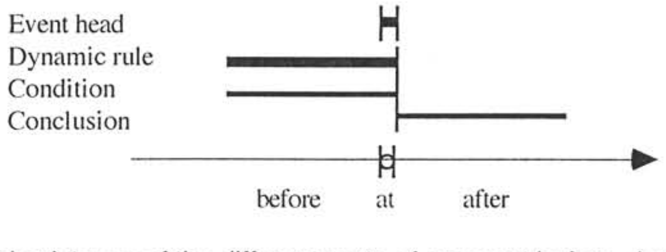


Figure 2. Temporal existence of the different parts of an event before, during, and after an occurrence of the event.

Figure 2 schematises the tense of event constitutes at the specification level. The relative point of time when the truth value of an event head is known, is the same as the evoking time. The relative time interval when the truth value of a dynamic rule is known, is at the evoking time. The relative time interval when the truth value of a condition is known, is before and at the evoking time. The relative time interval when the truth value of a conclusion is known, is after the evoking time.

Note that the notion of time is not introduced as an explicit property of concepts and rules as in [9 and 10]. In this work, an explanatory notion of time implicit in event specifications is sufficient to be able to handle the temporal aspects in schema comparison. The purpose is to enable analysis of temporal semantics of event constituents at the specification level rather than to collect historical data over their occurrences at the instansiation level.

The temporal semantics of conclusions is drastically different from dynamic rules and conditions, as illustrated above. It is not sufficient to compare an event with another as a whole. When two events are compared, the truth value of a condition is not based on the same system state (information base) as a conclusion nor event head. The truth value of the rules in the different event constituents is known at different points of time. The truth values of conditions and conclusions are known in a sequence and at distinct points of time. When the former is assigned a truth value the latter is unknown, and vice versa. The truth value of a dynamic rule is known only at the event occurrence time. The problem of temporal dependency arising from the conflicting temporal semantics should be considered. The different parts has to be compared independently in a well defined context.

5 Conclusions

Two hard problems arising in comparison of specifications of dynamic components have been identified and analysed in this work. Following the problem specification, requirements for comparison of dynamic aspects of component schemas become visible. The problems of implicit properties and temporal dependency of dynamic schema structures can be managed when the event is divided into smaller constituents for which the temporal semantics is specified and a notion of temporal context established. Following that line of argument, it is recommendable that the event constituents should be analysed rather than the whole of the compared events, yet there is no need to introduce explicit notion of time to component schemas. An event meta-model should also be related to UML [4] as a response to the recent standardisation efforts.

An attempt to compare the event constituents rather than the whole of the compared events gives rise to a third kind of a problem, namely *a problem of referents in isolation*. The unbound variables contained in event constituent specifications can not be substituted when comparing two constituents as such. Intuitively, the unbound variables have more semantics when related to other event constituents. The inherent, relative semantics of variables in isolation needs to be considered, when comparing constituents of two dynamic schema structures.

References

[1] J. Bubenko and E. Lindencrona, Conceptual Modelling, Information Analysis (In Swedish), Student literature, Lund, 1984
[2] Dublin Core, can be found at URL: http://purl.oclc.org/metadata/dublin_core/
[3], Delphi, an Overview of the Language (in Swedish), F H. Höök, F 91 0881, Ellemtel, 1991
[4] H. E. Eriksson, and M. Penker, UML Toolkit, John Wiley & Sons, 1998
[5] From Fuzzy to Formal, ESPRIT Basic Research P6621, Final Report, 1995
[6] S. Hakkarainen, Analysing and Evaluating Heuristic Support for Schema Comparison, in H. Kangassalo et al. (Eds.), *Information Modelling and Knowledge Bases VII*, IOS Press, 1995
[7] S. Hakkarainen, 'Correspondence Analysis of Dynamic Schema Constructs Using Case Grammar', R. P. van de Riet et al. (Eds.) *Applications of Natural Language to Information Systems,* IOS Press, 1996
[8] Novel Approaches Underlying Requirements Engineering, ESPRIT Basic Research P6354, Final Report, 1996
[9] Towards First Generation of Temporal Data Base ESPRIT Basic Research P7224, Final Report, 1994
[10] Integrating Database Technology, Rule-Based Systems and Temporal Reasoning for Effective Software, ESPRIT Basic Research P2469, Final Report, 1994
[11] Universal Resource Characteristics, can be found at URL: http://www.hypernews.org/HyperNews/get/www/URCs.html
[12] W3C Metadata Activity, can be found at URL: http://www.w3.org/Metadata/

Using Viewpoints for Component-Based System Development *

P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena
Department of Computer Science
University of Waterloo, Waterloo, ON, N2L 3G1 Canada
{alencar, dcowan, lucena }@csg.uwaterloo.ca

Abstract

We propose a systematic domain-oriented approach to the development of component-based software (CBS) systems based on viewpoints. In this research we focus on design and implementation of pre-built components and frameworks constructed from these components. Both the components and the frameworks are viewed as black boxes where the software developer should have no knowledge of their internal structure.
Keywords: *Component-based software engineering, viewpoints, off-the-shelf components, frameworks, design patterns, unification, consistency, transformations, file systems, software process life-cycle applications, Web-based education systems.*

1 Background and Motivation

Business, government and the software development community have all recognized the inadequacy of current software development techniques as the demand for more complex applications continues to grow without bound. Business and government have become aware of this problem through their inability to hire enough qualified software-personnel, while software developers have discovered they must increase their productivity in order to meet the increased software development demands of their respective organizations.

This crisis, along with other factors, including the appearance of technologies [12] such as the Internet, the World-Wide Web (Web), CORBA, the Distributed Component Object Model (DCOM), and JavaBeans, has caused a re-examination of the concept of constructing software components rather than rewriting each new application from "scratch". This approach, often called component-based software development, has been discussed in the literature since 1968, and has been pursued for many years, most recently by the object-oriented development (OOD) community.

Leading software developers believe that OOD has failed to deliver on promised productivity gains through component reuse [16]. However, components and frameworks encompass more than object technology, and initial experience with this broader group of software components [12], has provided strong evidence that we may finally realize the productivity gains originally expected from OOD.

2 Previous Work

Our research team has spent most of the last decade examining how to define and assemble components into operational software systems, while minimizing the handcrafting inherent to programming. Specifically, we have been developing theories, methods, and tools to support this process and then testing them on real-life examples.

This research program encompasses theory and methods in object-oriented design and programming, frameworks, design patterns, transformations and viewpoints. A framework is a set of components and rules for

*The work described here has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Information Technology Research Centre of Canada (ITRC), the National Research Council of Brazil (CNPq), and Sybase Inc.

collaboration [20] among the components. Frameworks can be viewed as mini-architectures or sub-assemblies often representing the kernel of an application for a specific domain. Frameworks or components can be plugged together at specific connection points to create a broader application. In certain contexts these points are called "hot spots." Black box frameworks have the same properties and advantages as blac box components. However, many current frameworks [10, 20] require some knowledge of internal structure in order to plug them into applications. The theoretical notions include the abstract design view [14], and the corresponding "views-a" operation [4, 9, 8], both conceived by our research group. The views-a concept is a new object-oriented paradigm similar to inheritance (is-a), and containment (holds-a) that clearly defines the semantics for interfaces. We believe that views-a provides a model for the software "glue" that adapts and combines components. The methods are based on transforming the abstract views-a operation into the concrete representation of object-oriented design patterns and the tools to support this process. Finally, we have been experimenting with transforming the object-oriented models into available components such as SQL and multimedia databases, and legacy software systems [7, 6].

We have applied the lessons learned from this research to the construction of several network-centric software systems including the application services for a community network, and a Web-based education environment. These applications were designed using the theories, techniques and tools described previously, and were assembled from pre-built components. During these projects, we also identified and built components that are not available from a supplier. These specific components were built in a style that allowed composition with other pre-built software components [5].

3 A Viewpoint-Based Development Approach for CBS

Our research objective is the creation of a complete domain-oriented development approach for component-based systems based on viewpoints. We have several years of experience building component-based software systems [5] in specific domains and creating and extending several related abstractions [14, 4, 7, 6]. This background has provided us with insights into the analysis, design and implementation of this type of software. We are specifically interested in using pre-built components to create black box frameworks in a more methodical fashion.

The use of pre-built components in software development has generated a substantial interest [15, 16, 17, 19, 12, 11, 20]. However, current methods are based on ad-hoc techniques, and at best adopt general descriptions such as the component-based system reference model [11] developed at the Software Engineering Institute. Reported research has focused on the description of systems and associated problems, but there has been little investigation of a design and implementation process.

A model for the specification, design and implementation process for component-based software that we propose might be viewed as consisting of five steps.

1. Determine the perspectives or viewpoints of an application - viewpoint analysis and consistency is applied to component-based software design.

2. Determine the kernel or framework of an application using unification of viewpoints - unification and domain specific languages are used to determine black box frameworks, and the points at which components can be glued to frameworks.

3. Glue the components and frameworks together to create a complete application - the glue semantics is characterized through the views-a operation. Pre-built components are connected to frameworks by using this glue model.

4. Map the glue into design patterns - the views-a semantics is used to guide the selection of appropriate design patterns. Object-oriented descriptions of the applications are produced.

5. Transform the resulting object-oriented design into "real" components - the interface and functionality of some components from an object-oriented perspective is described and the corresponding transformations are developed.

We have partially tested each of these steps by describing aspects of several applications. Thus, we believe this process is feasible, and will make significant contributions to software engineering, as described in the next few paragraphs.

In the first step, we determine how to describe and use viewpoints and create methods for checking consistency among viewpoints as illustrated in [22]. A "viewpoint" is a view of a system from a suitable perspective where we focus on a specific set of concerns. We could view a distribution system from a user or supplier perspective, or from the business rules that have been implemented. In contrast, we could also view a software system from a structural, functional or sequencing perspective. In the first case, we are examining the system from the functionality presented to the external world, while in the second case, we are examining the different relationships of the same aspect of the system.

The two types of viewpoints are called inter- and intra-viewpoint representations respectively. A key consideration of viewpoints is that multiple viewpoints must present the same system consistently. We choose to describe a viewpoint in terms of an object-oriented model, because of the rich constructs provided by this form of design language.

The concepts associated with viewpoints are still evolving. For example, the Unified Modeling Language (UML) [13] uses intra-viewpoints, while the Open Distributed Processing (ODP) Standard [21] uses inter-viewpoints. However, neither UML nor ODP address consistency among viewpoints, a concept that is critical, if viewpoints are to be useful in design. Since viewpoint analysis and requirements analysis are similar, we expect to be able to use concepts and methods from the latter to assist with understanding and developing viewpoints.

We unify the object-oriented viewpoints in the second step in order to determine the corresponding object-oriented framework. Unification can be informally considered as finding the intersection of the viewpoints. The unification of viewpoints requires that all of them be expressed in a common vocabulary, and ideally that the concepts are mutually orthogonal. Because a common vocabulary is not possible across all applications, the vocabulary must be developed as a domain specific language in the context of an application domain. Current methods for producing frameworks provide little systematic guidance as they rely on the experience of the software developer. Elementary versions of the process unification have only been recently described [1], and this process requires substantial extensions and experience.

The resulting object-oriented framework or core of the system will be augmented in step 3 to produce the various applications that formed the original viewpoints. Informally, we can visualize and addition that is glued to the framework, as the difference between the framework and each viewpoint. The Abstract Design View and the accompanying views-a operation have appeared extensively in the literature [14, 4], and can be used as the model for the glue. Using this abstraction to model glue allows the software developer to visualize the assembly operation more clearly. Normally, the glue would be buried in a complex programming structure that would obscure its real purpose.

The views-a operation has well-defined properties or semantics [4]. We characterize the relationship between viewer and viewed objects based on a set of primitive semantic properties. For each of the primitive properties considered we provide an informal description. A rigorous definition of such properties, which can be seen as a formal model for gluing object-oriented components, is provided in [4, 9, 8]. The views-a properties include the ones shown in the following table:

	Views-a Property	Property Meaning
1.	Identity	viewed and viewer have different identities
2.	Cardinality	the cardinality of the relationship
3.	Creation/Deletion	dynamic aspects about the relationship
4.	Singularity/Multiplicity	single/multiple viewers
5.	Vertical Consistency	consistency between viewers
6.	Horizontal Consistency	consistency between the viewed and viewer
7.	Visibility	viewed objects do not know its viewers

In Figure 1 we illustrate how this relationship was formalized. It shows the diagram where two object class theories VR (the viewer) and VD (the viewed) and one relationship theory V (views-a) are interconnected by morphisms to derive the composite theory C. C is interpreted as the colimit of theories R, D, and V. Informally, the colimit is the mechanism that puts together (or combine) the two classes and the views-a relationship. In this

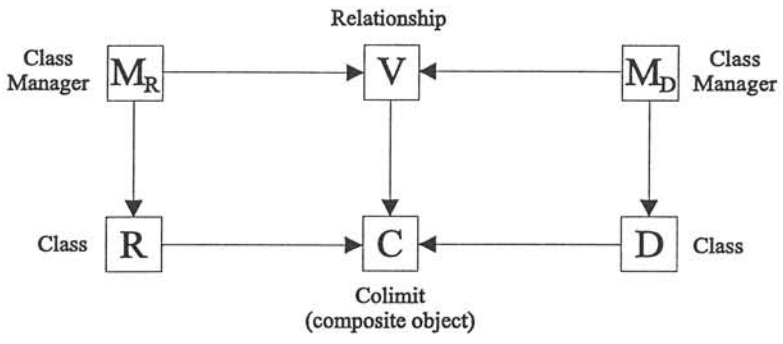


Figure 1: Diagram showing the colimit of object and relationship theories.

context, the formal model for the views-a operation can be seen as a model for the glue that puts components together. The set of properties which interpret the *views* relationship, relating viewer ($@VR$) and viewed ($@VD$) objects are defined in this formal setting [9].

The views-a operation can be implemented in terms of one of the standard object-oriented design patterns. In step 4 we choose the appropriate design pattern by comparing a collection of design patterns with the views-a properties required in a specific design. The result of this process is an object-oriented design model for the entire set of applications. We have informally demonstrated the mapping of the views-a operation into different design patterns, but a more structured approach is required before the technique is truly usable by a software designer.

Once the model is defined, we will combine object-oriented components with the framework model in an instantiation process to produce a concrete object-oriented application. However, available components are often not object-oriented, and so in step 5 we must specify the interface and functionality of these components in an object-oriented format for inclusion in the framework, and transform our object model into the style of commercial components that are available. The transformation approach allows the designer to use the rich modeling language of the object-oriented world for design, while mapping into the components currently available. We have some experience transforming object-oriented designs manually and using transformation systems such as DRACO [18, 19], but we need to extend our transformation approach in a more systematic manner.

Once the object model is described and mapped into available components, we will have also identified missing components. However, their design will be clearly defined by the process just described, and so implementation of these components can proceed. We have applied this process to some components in isolation, but need to verify the process within the context of complete system designs.

We believe that some of the steps in our proposed approach to component-based design can be used even when the software developer uses ad-hoc methods. We have found through experimentation, that using some of the steps to analyze a design in progress clarifies many of the design decisions, and often leads to an improved software system.

Our research group plans to examine specific application domains in order to define the steps in the design and implementation process more clearly. We will concentrate on network-centric applications such as the community network, and Web-based education systems, in order to define the domain-specific language, and to test the design concepts. Once we have a number of examples, we plan to generalize the lessons learned to a design approach for applications using component-based software.

4 Case Studies

In this section we briefly describe some case studies where we have applied our viewpoint-based approach to CBS. A first case study deals with the NACHOS file system. In this case [7, 6], there are two viewpoints (also called here subjects): one for concurrency and one for large extensible files (see Figure 3). As we have only one application (and not a domain) we did not have unification of viewpoints. The kernel of this application was the original NACHOS file system design and, as a black box framework, it would be extended to deal with the two viewpoints. The views-a operation was used to glue the two viewpoints and the black box framework together to create a complete application. This operation is represented by the arrows in Figure 3. The glue (or the views-a operations) was transformed into a design pattern (Figure 2). The views-a semantics was used to guide the selection of this appropriate design pattern. In this case, property 3 is the most important one. Horizontal and vertical consistencies (5,6) were not required. Notice that all views-a operations led to the same design pattern. The resulting object-oriented design was manually transformed into the "real" components. For details, see [7] and [6].

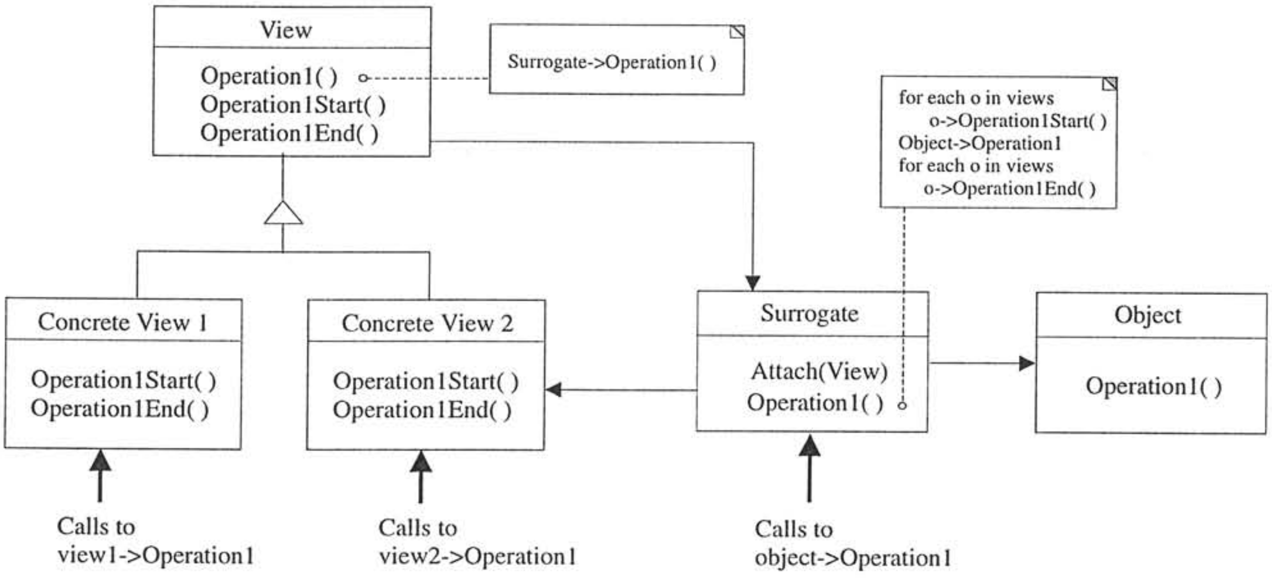


Figure 2: Design model for *views-a* operator

A second case study deals with a process life-cycle application. In this case [3], the viewpoints are given in Figure 4. The kernel of this application was obtained by informally unifying the viewpoints. In this case we have produced a domain-oriented design solution. The points at which the components can be glued to frameworks are presented in Figure 5. The views-a operation was used to glue the black box framework (kernel) and the components together to create a complete application. The views-a operation related to the views of a process program is presented in Figure 6. The glue (formally modeled by the views-a relationship) presented in Figure 6 was transformed into the design pattern shown in Figure 7. In this case, properties 5 and 6 (vertical and horizontal consistency) are the most important ones. Property 4 is also required. The views-a semantics (given by properties) was also used to guide the selection of all the other design patterns related to the other views-a relationships in Figure 5. The resulting object-oriented design was also manually transformed into the "real" components.

A third case study led to a domain-oriented CBS design solution for a Web-based education software system based on viewpoints [2]. The steps of the our approach were also followed and the the views-a operation, as a model for glue, was transformed, depending on its properties in a variety of design patterns.

5 Conclusions

The proposed approach is been refined as we experiment with other case studies. Unification, consistency, and transformation are issues we are working on in a formal level in order to provide a more systematic description of this approach.

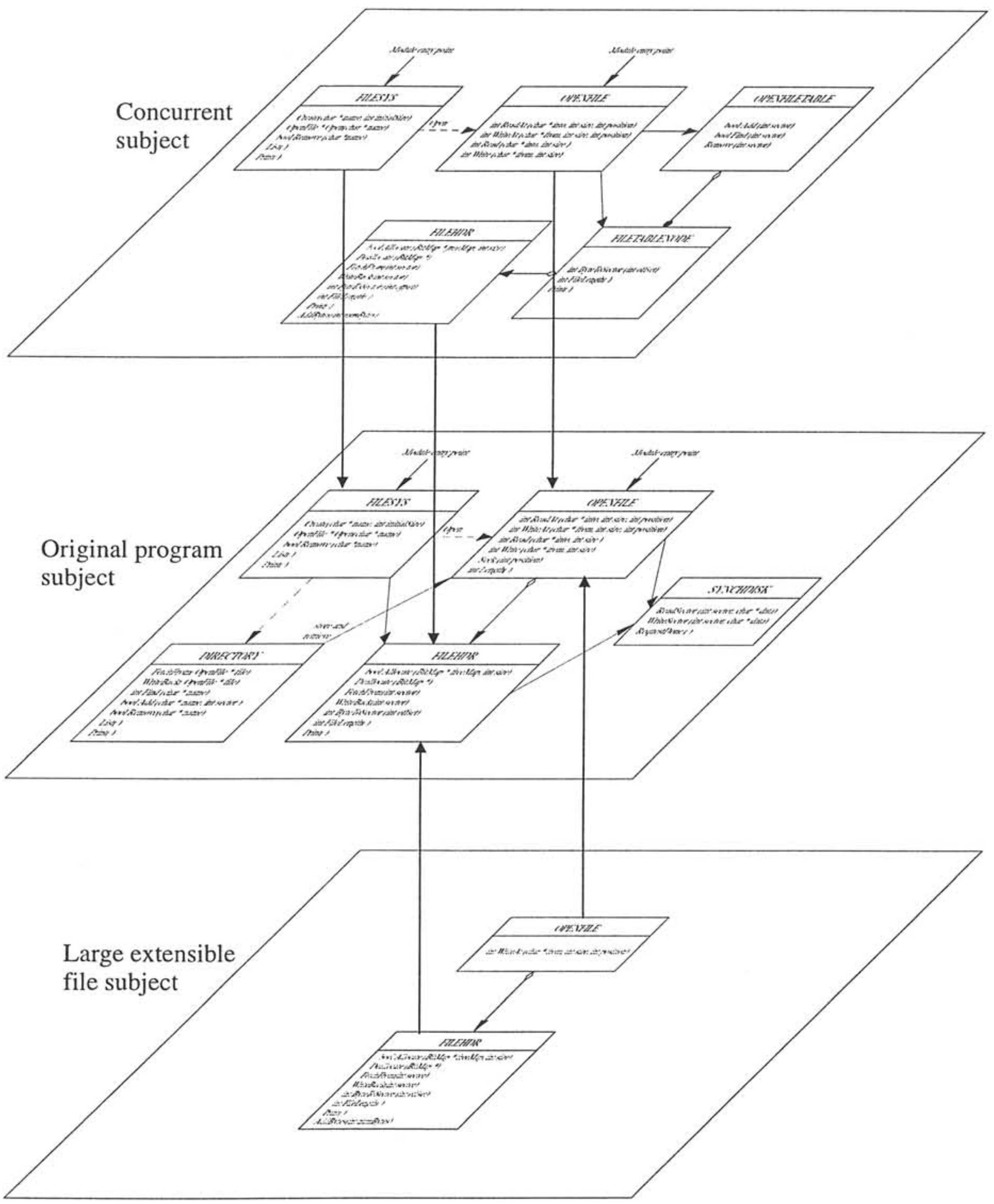


Figure 3: Viewpoint composition

References

[1] M. Ainsworth, S. Riddle, and P.J. Wallis. Formal validation of viewpoint specifications. *ACM Transactions on Software Engineering and Methodology 4(4)*, 1995.

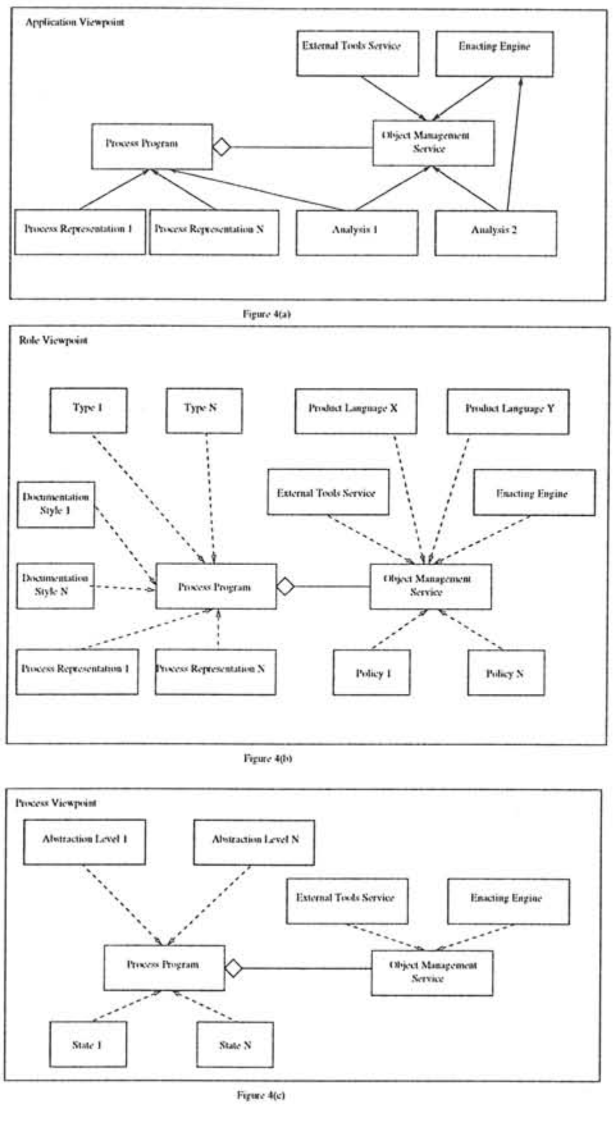


Figure 4: The viewpoints in the case study

[2] P.S.C. Alencar, D.D. Cowan, S. Crespo, M.F. Fontoura, and C.J.P. Lucena. A viewpoint-based design method: A case study in frameworks for web-based education. *CS-98, Computer Science Department, University of Waterloo*, 1998.

[3] P.S.C. Alencar, D.D. Cowan, M. Fontoura, and C.J.P. Lucena. A framework development approach based on viewpoints. *CS-97-, Computer Science Department, University of Waterloo; also submitted to ACM Book (two volumes) on Frameworks*, 1997.

[4] P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. A logical theory of interfaces and objects. *revised for IEEE Transactions on Software Engineering*, 1998.

[5] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and M.A.V. Nelson. An approach to hypermap-based applications. *Proceedings of the Second International Symposium on Environmental Software Systems (ISESS'97), Whistler, British Columbia*, pages 244–251, 1997.

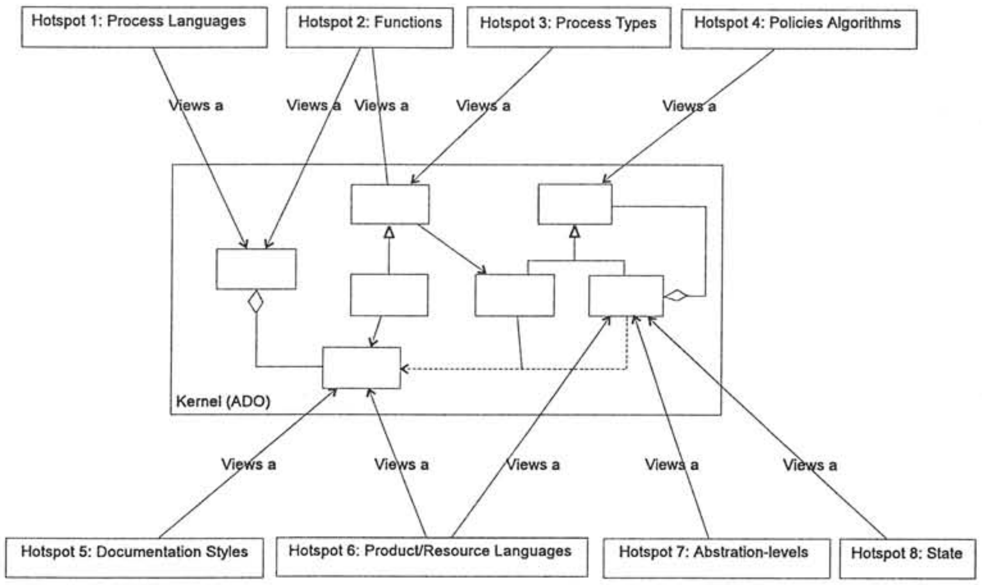

Hotspot 1: Process Languages
Hotspot 2: Functions
Hotspot 3: Process Types
Hotspot 4: Policies Algorithms
Views a
Kernel (ADO)
Hotspot 5: Documentation Styles
Hotspot 6: Product/Resource Languages
Hotspot 7: Abstration-levels
Hotspot 8: State

Figure 5: The application-specific framework structure

[6] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and T. Nelson. A viewpoint-based approach to software system evolution. *Proceedings of the Workshop on Software Process and Evolution - ICSE'97*, 1997.

[7] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and T. Nelson. Viewpoints as an evolutionary approach to software system maintenance. *CS-97-10, Computer Science Department, University of Waterloo, Proceedings of the International Conference on Software Maintenance, Bari, Italy*, 1997.

[8] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and L.C.M. Nova. A views relationship in object-oriented design. *CS-97, Computer Science Department, University of Waterloo*, 1997.

[9] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and L.C.M. Nova. A formal theory for the views relationship. *CS-98, Computer Science Department, University of Waterloo*, 1998.

[10] Paul G. Basset. *Framing Software Reuse.* Prentice Hall, 1997.

[11] A. W. Brown and K. C. Wallnau. Engineering of component-based systems. In *Component-Based Software Engineering*, Selected Papers from the SEI, pages 7–15. IEEE Computer Society, 1996.

[12] Alan Brown. Preface: Foundations for component-based systems software engineering. *Component-Based Software Engineering*, 1996.

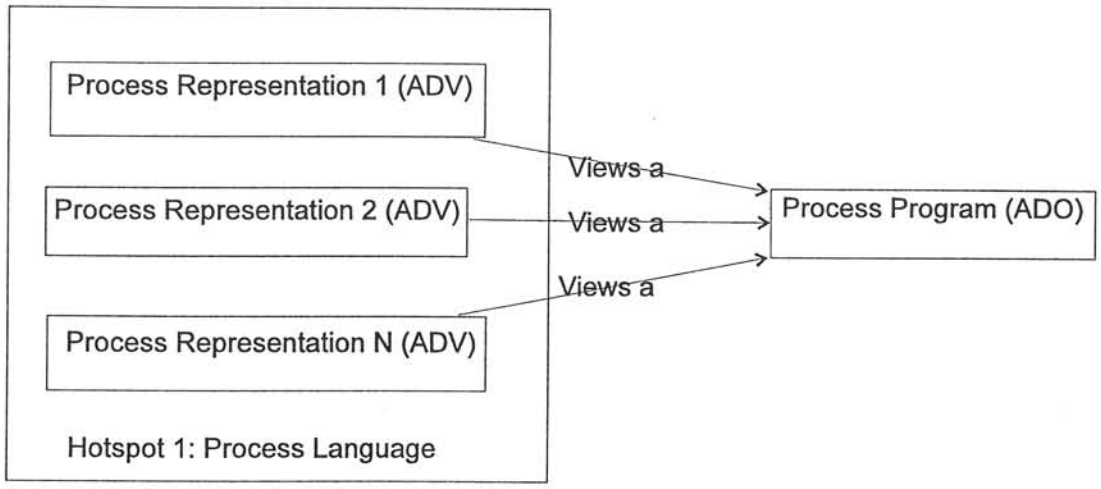
[13] Rational Software Corporation. *Unified Modeling Language version 1.1.* Rational, 1998.

[14] Donald Cowan and Carlos Lucena. Abstract data views: An interface specification concept to enhance design. *IEEE Transactions on Software Engineering*, 21(3), March 1995.

[15] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *The Proceedings of the Seventh International Conference on Software Engineering*, April 1995.

[16] D. Kiely. Are components the future of software. *IEEE Computer, Vol. 31, No. 2*, pages 10–11, February 1998.

[17] Gary T. Leavens, Oscar Nierstrasz, and Murali Sitaraman. 1997 workshop on foundations of component-based systems. *ACM SIGSOFT Software Engineering Notes*, pages 38–41, January 1998.



Process Representation 1 (ADV)
Process Representation 2 (ADV)
Process Representation N (ADV)
Hotspot 1: Process Language
Views a
Views a
Views a
Process Program (ADO)

Figure 6: The views of a process program

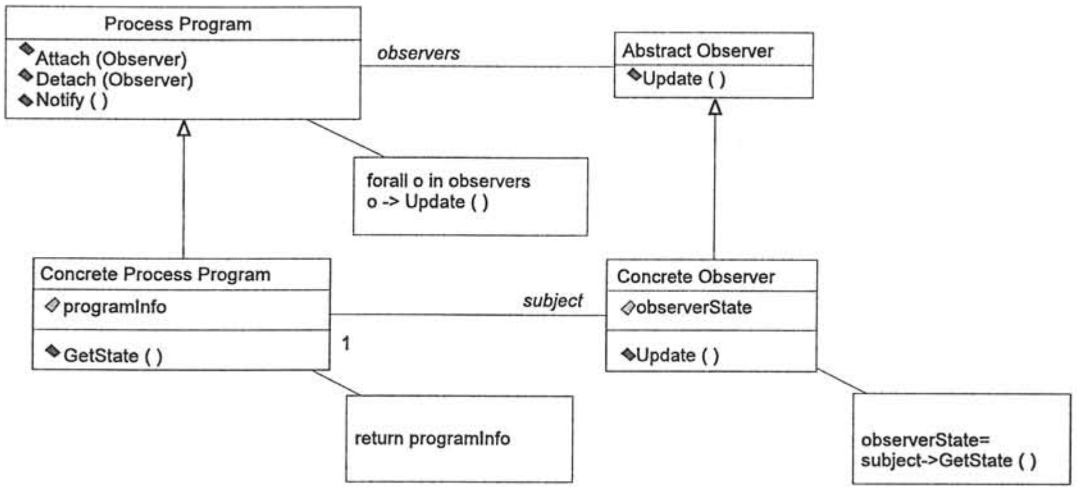


Figure 7: The observer pattern to create the process program views

[18] J.C.S.P. Leite, Marcelo Sant'anna, and Felipe Gouveia de Freitas. Draco-puc: a technology assembly for domain oriented software development. *The Proceedings of the Third IEEE International Conference on Software Reuse*, pages 1–7, 1994.

[19] J. Neighbors. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering 10(5)*, pages 564–574, 1994.

[20] Oscar Nierstrasz and Dennis Tsichritzis. *Object-Oriented Software Composition*. Prentice Hall, 1995.

[21] Open Distributed Processing. *Reference Model - http://www.iso.ch:8000/RM-ODP*. ISO, 1998.

[22] Enoch Y. Wang, Heather Richter, and Betty H. C. Cheng. Formalizing and integrating the dynamic model within omt. *Proceedings of the 19th ICSE*, 1997.

Features of the SYNTHESIS component-based information systems development method

Leonid Kalinichenko, Dmitry Briukhov
Institute for Problems of Informatics
Russian Academy of Sciences
e-mail: leonidk@synth.ipi.ac.ru

Franz Kapsner
Siemens AG
Munich, Germany
e-mail: Franz.Kapsner@oen.siemens.de

1 Introduction

We consider component-based development process to be positioned on the middleware platform (like OMG CORBA) supporting heterogeneous interoperable information resource environment (HIRE). As a component, an information resource in HIRE is a replaceable unit of development which encapsulates design decisions and which can be composed with other components as part of larger units. Components include not only implementations, but also more abstract development artifacts such as type specifications, application context specifications, generic and customizable units.

The idea of interoperability through the common software bus greatly improves an information systems development infrastructure. Evolution of the systems through integration of independently evolving heterogeneous subcomponents becomes much easier: distribution and technical heterogeneity of components do not impose any problem. Construction of a system as a collection of interoperable components simplifies its re-engineering and incremental migration to the new application and technological requirements.

Broad industrial scale of the interoperation technologies creates a strong demand for component-based information systems development based on the provided middleware. New information systems development methods are required to take advantage of the new capabilities.

It is clear that correct compositions of software and data components should be semantically interoperable in the context of a specific application (application contexts of the components should be coherent and their composition should be consistent within the context of the intended application). To close the gap that exists between the technical (CORBA) and semantical ability to interoperate, in the SYNTHESIS project [1] we introduced a notion of *semantic interoperation reasoning* (SIR) [9] as a collection of basic models and methods supporting the required level of semantic interoperation and finally leading to component-based information systems development.

Current industrial trends in component-based development (e.g., DCOM ActiveX and SunSoft JavaBeans solutions) can be characterized as overemphasizing low level specifications (actually, conventional programming techniques) hidden by visualization tools. Due to the lack of semantics, identification of components relevant to the requirements and their composition becomes pure programmer's intuitive decision. This may work in local libraries with rather limited numbers of pre-existing components. For HIRE having broad scale such approach is not applicable. Even more important is that intuitive using of incomplete specifications leads to unsafeness of component-based development (e.g., due to unsafe reuse of pre-existing modules [7]).

Standardization consortiums (like OMG) follow these unscaleable and unsafe trends applying ad hoc solutions. UML is just one example of recent ad hoc solutions by OMG [5]. Positive side of UML is that Object Analysis and Design Methods are consolidating around UML diagrammatic representation with the serious moves to component-based software reuse solutions [6]. Enhancing of the existing synergy between CORBA and Java by defining a CORBA component model compatible with JavaBeans [4] will not improve

[1]The SYNTHESIS project developed at the Institute for Problems of Informatics of the Russian Academy of Sciences (IPI RAS) is partially supported by the INTAS grant 94-1817, the Russian Foundation for Basic Research grant 97-07-90369 and by the cooperating research between the Siemens Corporate R & D and IPI RAS

the situation. Trading function for component-based development (a third party object that enables developers to find suitable components and servers in HIRE) should be based not on an IDL specifications of components (like in [16]) but on sufficiently complete specifications.

Developing of more well grounded strategy for the componentware infrastructure for survival in more durable perspective is required. Sound understanding of the essence of component-based development and reusability problem leading to proper component modeling, compositions and design in HIRE is required. We shall briefly refer to the decisions of the SYNTHESIS project that is focused on that at the Russian Academy of Sciences [9, 11, 12, 13, 14].

2 SYNTHESIS project approach

Investigations of the issues listed showed that the respective decisions are appropriate:

1. The required level of completeness of specifications for the component-based development.

   The problem of components is that they do not have sufficiently clean semantic specifications to rely on for their reuse. We distinguish between application semantics and object model semantics. The latter we represent in frame of a "canonical" semi-formal object model [10] used for uniform representations of various object and data model specifications in one paradigm. Application semantics of specifications we consider separately in frame of the ontological approach providing for specification of conceptual (terminological) context of the application domain.

   Component specifications should define contextual information of an application in form of ontologies as well as provide complete structural and behavioral information for the information system entities including complete specifications of type invariants, operations and workflows.

2. The required canonical modeling facilities to uniformly specify heterogeneous pre-existing components with the required completeness.

   Canonical modeling facilities should provide for complete abstract data type specifications - of an object and non-object kinds (including complete specification of operations and workflow types), subtyping (with rigorous semantics) and classification (metatype based) hierarchies, support of the spectrum covering unstructured, semi-structured and structured data, treating type attributes as type definitions in their turn, a complete type specification algebra producing new type specifications from the existing ones, object calculus support with formulae leading to a possibility of creating collections of values of new types produced by a type specification algebra.

   To give the canonical model exact meaning, we construct a mapping of this object model into the formal one (we choose for that Abstract Machine Notation (AMN) of the B-Technology [1]).

3. The appropriate models to specify the information system requirements and result of analysis, relationship of these models to the canonical one.

   Modeling facilities for all phases of the component-based development (including ontological specifications) are those of the canonical component specification model.

4. The required modeling facilities for the component-based design (with reuse).

   The design is based on the canonical model facilities having rich possibilities for type specification compositions.

5. How standard object models (like IDL, ODL, CDL) should be correlated to the canonical model.

   There should exist an equivalent mapping of IDL, ODL and other standard specification languages into the canonic specification model.

6. What are suitable representational facilities for the models above (graphical, diagrammatic, etc.).

   Any standard representation facilities may be suitable (e.g., a UML subset with some extensions may be quite good): but the approach is different from conventional, it is the canonical model that has primary importance providing proper interpretation and meaning for graphical representation.

7. How to identify component *capabilities* that can provide a support of a specific application (specification of requirements).

   Identification of component capabilities is based on the ontological relevance of component constituents to the constituents of the analysis model and on an ability of such component constituents to rigorously *refine* the respective constituents of the analysis model.

8. How to correctly decompose specifications of pre-existing components into smaller fragments (capabilities) potentially relevant for specific requirements.

   The ability to decompose is based on properties of the type specification algebra of the canonical specification model.

9. How to correctly compose relevant component capabilities to get a *refinement* of a respective part of a specification of requirements.

   The ability to compose is based on properties of the type specification algebra of the canonical specification model and on proper methods leading to rigorous formation of refinements of analysis model specifications.

   The approach to type compositions is based on an idea of a *common reduct* (a reduct is a subspecification of a type) for types T_1, T_2 that is such reduct R_{T_1} of T_1 that there exists a reduct R_{T_2} of T_2 such that R_{T_2} is a *refinement* of R_{T_1}. For reusability decision the *most common reduct* of two types (the analysis model type and the component model type) is the main target. Thus the largest component fragments potentially reusable for the respective reducts of the analysis model types can be identified. Type algebra is also based on this idea. More on that can be found in [13, 15].

10. What is the minimal set of required formal facilities and methods to justify different steps of the component-based design process.

    The required formal facilities and respective design procedures should provide for justification of: (i) consistency of specifications of requirements and of component specifications, (ii) correctness of abstraction of original component specifications by the canonical ones, (iii) identification of truly relevant component capabilities for a given specification of requirements, (iv) correctness of capabilities compositions, (v) validity of refinement of a specification of requirements (or its part) by a composition of pre-existing capabilities.

11. How the required formal facilities should be related to the canonical and requirements modeling facilities above.

    There should exist a mapping of the canonical modeling facilties to the formal ones.

The canonical modeling facilities possessing the features above were defined, principles of mapping of the canonical specifications to a formal notation were elaborated, methods of component-based design relying on the canonical model, its type algebra (making possible proper type specifications decomposition and compositions) and mapping to the formal notation have been developed, the process of the component-based design using graphical representation facilities for the developers has been developed, methods for abstracting of canonical component specifications from the original heterogeneous ones have been defined.

The component-base design process has been partially prototyped as the SYNTHESIS design tool. The prototype supports ontology integration, searching for relevant components, identifying reusable fragments of specifications of pre-existing components, resolving structural conflicts between specifications of requirements and pre-existing components and constructing compositions of the fragments to get a refinement of respective part of the specifications of requirements. GUI of the prototype includes graphical representation of application and components using the UML notation, graphical representation of fragments composition tree, special text forms to reach completeness of the specification of compositional types.

3 SYNTHESIS method prototype architecture

Figure 1 shows a general structure of the SYNTHESIS prototype supporting the design and implementation phases of the component-based development in HIRE.

We choosed the ParadigmPlus [8] to support the Requirement Planning and Analysis phases (the SYNTHESIS tool is independent on this choice). The UML notation augmented with SYNTHESIS specific features, such as ontological specifications and predicative specifications of operations is used. Actually any standard representation facilities can be suitable (as the UML subset with some extensions) but the approach is different from conventional one, it is the canonical model that has primary importance providing proper interpretation and meaning for graphical representation.

Using the CDIF representation [3] taken out of the ParadigmPlus, the specifications of the analysis phase are loaded into the metaobject repository supported by the PSE ObjectStore.

For formal modeling the B Abstract Machine Notation is used [1] that together with B-Toolkit [2] provides for type specification consistency check, adequacy of component specifications [12], establishment of a refinement condition, generating and proving respective proof obligations.

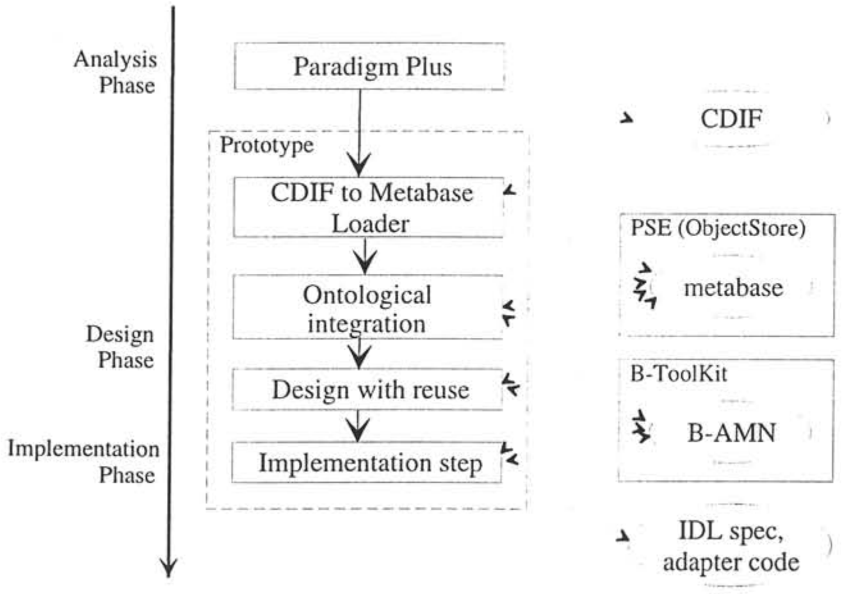


Figure 1: SYNTHESIS Prototype Structure

SYNTHESIS prototype is implemented using Java 1.1 under Windows NT 4.0. The tool is oriented on work in a CORBA-like interoperable environment: the tool generates an IDL specifications and CORBA-based adapters code.

4 Conclusion

Component-based information systems development is a complicated technology that requires serious investigations and systematic approach. Ad hoc solutions on separate constituents of the technology do not lead to a suitable practice. The constituents of the technology are so highly interdependent that only a systematic approach for their elaboration can give proper results.

The SYNTHESIS project shows that the systematic approach for the component-based information system development is feasible. Many issues remain still to be investigated. Among them is an issue of making the methodology easy to use for an average developer.

References

[1] Abrial J.-R. *The B Book: assigning programs to meaning,* Cambridge University Press, 1996

[2] Abrial J.-R. B-Technology. Technical overview. BP International Ltd., 1992, 73 p.

[3] EIA Interim Standard: CDIF-Framework for Modeling and Extensibility. EIA, 1991.

[4] CORBA Component Imperatives. ORBOS/97-05-25. IBM Corporation, Netscape Communications Corporation, Oracle Corporation, Sunsoft, Inc.

[5] M.Fowler UML Distilled, Addison-Wesley, 1997

[6] I.Jacobson, M.Griss, P. Jonsson, *Software Reuse*, ACM Press, 1997.

[7] J.-M. Jezequel, B. Meyer, Design by Contract: The Lessons of Ariane, http://www.tools.com/doc/manuals/technology/contract/ariane/index.html

[8] Paradigm Plus Reference Manual. Protosoft, 1997

[9] Kalinichenko L.A. Emerging semantic-based interoperable information system technology. In *Proceedings of the International Conference Computers as our better partners*, Tokyo, March 1994. World Scientific.

[10] Kalinichenko L.A. SYNTHESIS: the language for desription, design and programming of the heterogeneous interoperable information resource environment. Institute for Problems of Informatics, Russian Academy of Sciences, Moscow, 1995, 103 p.

[11] Berg K. and Kalinichenko L.A. Modeling facilities for the component-based software development method. In *Proceedings of the Third International Workshop ADBIS'96*, Moscow, September 1996.

[12] Kalinichenko L.A. Method for data models integration in the common paradigm. In *Proceedings of the First East European Workshop 'Advances in Databases and Information Systems'*, St. Petersburg, September 1997.

[13] Kalinichenko L.A. Workflow Reuse and Semantic Interoperation Issues. In *Advances in workflow management systems and interoperability.* A.Dogac, L.Kalinichenko, M.T. Ozsu, A.Sheth (Eds.). NATO Advanced Study Institute, Istanbul, August 1997

[14] Kalinichenko L.A. Component-based Development Infrastructure: A Systematic Approach *OMG-DARPA-MCC Workshop on "Compositional Software Architecture"*, Monterey CA, January 6 - 8, 1998

[15] Kalinichenko L.A. Composition of type specifications exhibiting the interactive behaviour. In *Proceedings of EDBT'98 Workshop on Workflow Management Systems,* March 1998, Valencia

[16] *ODP Trading Function - Part 1: Specification,* ISO/IEC IS 13235-1, ITU/T Draft Rec X950 - 1, 1997.

CBiSE Environments

Nikolay Mehandjiev, Melvyn Roberts
School of Management, University of Hull, HULL, UK

Abstract

To be effective, component-based development would need to use better ways to hide the complexity of the application systems from the developers. The abstraction levels offered by component technology such as CORBA are not adequate. This has motivated the use of application frameworks, which provide a higher level of abstraction. However, using application components that are too abstract would make it difficult to change the application system. In this paper we argue that some abstract concepts should not be a part of the underlying application framework. In order that we can still benefit from using them, we can implement them within the development environment instead. Based on this premise, we formulate four requirements for successful component-based development environments, and investigate how such environments can be built.

1 Introduction

Component-Based Information Systems Engineering aims to simplify the process of software development to a stage where applications will be combined in a LEGO(tm)-like fashion out of pre-defined and tested components [4]. We have certainly heard this idea before, we have heard it about subroutines, about abstract data types, about objects. What is new about components?

We believe there are two novel aspects: the focus on bridging different platforms to create heterogeneous distributed applications, and the realisation that re-using a single item is more difficult and less rewarding than re-using a set of items structured in an application framework.

Existing CBiSE work [4, 10] has focused on these two issues, which form the bottom two layers of Figure 1, the layer of technical infrastructure, and the layer of application frameworks. These two layers abstract to certain extent the complexities of contemporary information systems, hiding from developers issues of distributed cooperation protocols, for example.

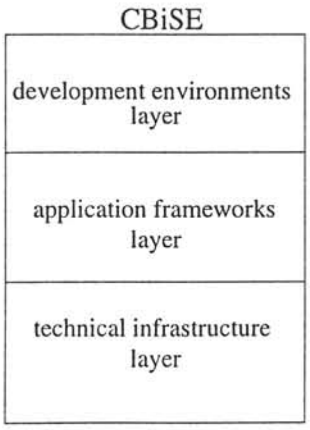

CBiSE
development environments layer
application frameworks layer
technical infrastructure layer

Figure 1: Layers of concern in CBiSE

Is this enough to take us to the new promised land, where we will assemble software out of LEGO blocks?

In this paper we argue that two factors are standing in the way. Firstly, the complexity of application software is too great even at the level of application frameworks layer. Secondly, the abstractions provided by the bottom two layers are too rigid to cater for the current rate of business requirements change.

These two factors motivate our interest in the third layer, the layer of development environments. We discuss the implications of these two factors for the requirements of CBiSE environments, and look at whether we can construct such environments with the help of the existing techniques and architectures.

2 Evolving domain-oriented abstractions in CBiSE

Let us look at the level of abstractions offered by the bottom two layers in Figure 1. The first of them, the technical infrastructure layer, hides the technical issues of distributed processing in an application both at design level and at implementation level. This layer will be based on standards like CORBA. The abstraction provided by this layer allows us to think of an application system as a "swarm" of active domain-oriented components. The application frameworks then put some structure on this "swarm" and build up further levels of abstraction. These further abstraction levels aid the developers of a system in their reasoning about the system [10, 9]. In complex domains, an application framework can provide different abstraction structures for different aspects of the application such as behaviour and data.

Application frameworks, however, seldom cater for the differences in perspectives that are held by the different participants in the system development process. Designers, for example, are likely to have different perspective on the system than analysts, analysts from customers and users, there would even be differences between users at different organisational levels [11].

One way to handle the multiplicity of perspectives is to over-elaborate the application framework by building into it different perspectives and abstractions. This might work for a short period of time, but it is not a viable strategy in the long run because of the rate of change in contemporary organisations. It is not only responsibilities and processes that change, the paradigms we use to think about organisations change as well. In the past, for example, activities were assigned to workers through their job description. The "job description" concept has recently been replaced by the concept of a "role". A "role" is more flexible, since one worker can play different roles at different times. Many organisations are now introducing the even more flexible concept of task teams, where responsibilities are dynamically allocated on the basis of goal-driven work plans.

Consequently, if a "job description" abstraction was implemented within an application framework a few years ago, it will by now be obsolete two times over. An application system built using this framework will "support" the organisation's team-focused work allocation in quite an appalling manner. It will also be quite difficult to modify this application because the problem is in the underlying application framework.

Applications frameworks, therefore, should not contain over-elaborated and highly abstract concepts which have a short life expectancy and can become obsolete in near future. These abstractions should be defined and maintained within the development environments, which should be made easily changeable to cater for changing abstractions and new perspectives to the application system.

3 Requirements towards CBiSE environments

If we develop further the ideas from the previous section, an effective CBiSE environment should fulfil the following main requirements:

(i) It should provide a range of descriptions to cater for the different aspects of the information system, for the different stages of system development, and for the multiple perspectives of the participants in the development process. We can consider each description to be provided by a separate tool. The multiplicity of integrated tools forms the CBiSE environment.

(ii) Modifying a description which has become out of date, or replacing an entire description with a new one, should be easy. This might happen while the application is deployed and functional, so that such changes should not disturb the component-based application and should not require it to be re-structured. This can be achieved by a "plug and play" mode of integration between the different tools, and between each tool and the application system.

(iii) The "plug and play" mode of integration between different tools should be combined with automatic consistency maintenance and tight integration mechanism between different tools. The integration mechanism should also provide facilities, where some descriptions will automatically reflect the dynamic state of the running application. This will allow for monitoring and control of the functioning application system.

(iv) Each tool may define and maintain abstract concepts, which are helpful for developing and understanding the system, but which have been deemed too volatile to go in the application framework. Other tools may need to also refer to these concepts, therefore facilities for sharing these concepts between tools are needed.

4 Techniques and architectures for building CBiSE environments

There are many existing integration techniques and architectures. In this section we will look into how they can be used to build CBiSE environments, bearing in mind the requirements from the previous section.

At the highest level of abstraction, existing integration approaches can be divided into two main topologies: "network" and "star". A node in each topology will correspond to a tool or to the application system in our model of CBiSE environments.

In the "network" topology, consistency and integration between any two nodes are implemented by the two nodes interacting directly, most often by exchanging messages. A clear example of this approach is the work on Viewpoints [1]. This network approach is quite suitable for environments where the working software document may be distributed, and where consistency rules may have to be relaxed to "tolerate" some inconsistencies between parts of the distributed document. Both these requirements are often true during early software lifecycle stages. However, the complexity of the "node to node" connection network can grow quite quickly, and this would result in a system where it would be difficult to trace the effects of any change at any one node of the system. Consequently, in its general form, this approach would not satisfy most of the requirements we have formulated for CBiSE environments.

The "star" topology can be seen as a special case of a "network" with one "central" node. This node holds all information to be shared in the system. All other nodes only communicate through this "central" node, thus forming a "star" pattern. This approach is often found in development environments which address later lifecycle stages. In these environments the central node is either a detailed software design document or the software itself. Because this topology is based on shared information which acts as a "model" of the MVC paradigm [2], it can not handle well the requirement for relaxed consistency rules.

Some architectures combine these two approaches. Such is the reference model proposed by the National Institute of Standards and Technology and European Computer Manufacturers Association (NIST/ECMA) reference model [8]. The model, shown on Figure 2, is nicknamed "the toaster model", because different tools can be plugged into a framework, which provides common presentation and data integration services, and protocols for message exchange between the tools. The model combines a "star" data integration approach through a common data module, and a "network" control integration approach by node-to-node message protocol between the tools. Example of an environment that fits this model is the NATURE requirements capture environment [3].

Environments that are built using this model are potentially suitable for engineering component-based information systems. The model provides facilities for flexible integration of multiple tools, thus satisfying the first and the second requirement from Section 3. Complicated "network" message exchange patterns

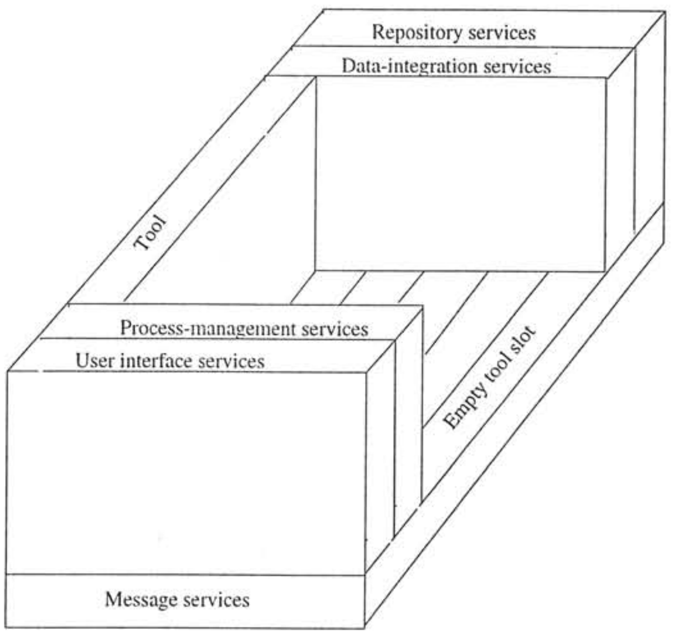


Figure 2: The NIST reference model

between different tools can cause potential problems when replacing a tool. These problems can be addressed by either standardising the tool message interfaces, or by providing brokerage facilities within the message exchange services.

Satisfying the third requirement, however, will require changes in the model, because the model is oriented towards manipulating passive data documents, rather than towards controlling and monitoring active application systems. The repository and data integration services will have to be modified into services for managing an application composed of active components.

It is also not clear how the model can satisfy the fourth requirement of tools sharing abstractions which are not a part of the application system. The answer depends on the particular tool integration mechanism being used. Meyers [7] defines five tool integration mechanisms, of which two are particularly relevant to this problem: the 'view-oriented database' mechanism and the 'canonical representation' mechanism.

In the 'view-oriented database' mechanism, each tool defines its own view of a central database, which, in our case, will contain the active application system. Although the data in each view can be organised in a substantially different manner from the central data, tools do not keep any of the information locally, they compute the relevant values through the view mechanism from the common repository [3]. This mechanism is widely used by environments aimed at initial lifecycle stages. It lacks flexibility when adding a new type of abstraction or a tool, because this nearly always requires enhancements to the database schema [7]. When the central repository is the application system itself, changes in its schema are often not possible. For example, if we work with a debugger for applications written in C++, it would be impossible to change the way the debugging information is stored in the application so that we can introduce a new debugging view.

For this reason, environments used for late stages of the software lifecycle use the second approach, the "canonical representation" mechanism. In this mechanism, every tool or description interfaces to a central canonical data structure, which contains the software document or application. For example JBuilder, Inprise's Java Integrated Development Environment, uses three standard formats for each of its software projects: a project description file, HTML files and Java files. The environment claims 100% compatibility with any standard Java file, which means that Java applications developed with a different environment can be modified in JBuilder.

Each tool builds its own interpretation of the shared data, stores it when necessary, and presents this interpretation to the user. JBuilder offers two views for the HTML files, a rendered view and a source code view. It also offers two views for Java files: a source code view and an interface design view. In JBuilder the conversion between a representation offered by a tool and the software document is relatively straightforward, for example it is comparatively easy to render the interface design view out of the Java source file.

'Canonical interface' systems such as JBuilder offer good flexibility when adding new descriptions or tools, since adding a new tool only requires knowledge of the canonical interface to the application repository. The problem with this approach arises from the fact that different tools can not share abstractions that are not in the central application repository (the fourth requirement), without loosing their "plug and play" flexibility (the second requirement).

In the next section we show one possible approach to addressing this issue. The approach combines the benefits of the 'view-oriented database' mechanism and the 'canonical interface' mechanism to satisfy our four requirements towards CBiSE environments.

5 The architecture of pluggable visual descriptions

The architecture of pluggable visual descriptions is described in detail elsewhere [5]. This section will give only a brief overview of the architecture and will offer some considerations as to how it might satisfy the four requirements to CBiSE environments. The main components of the architecture are shown on Figure 3.

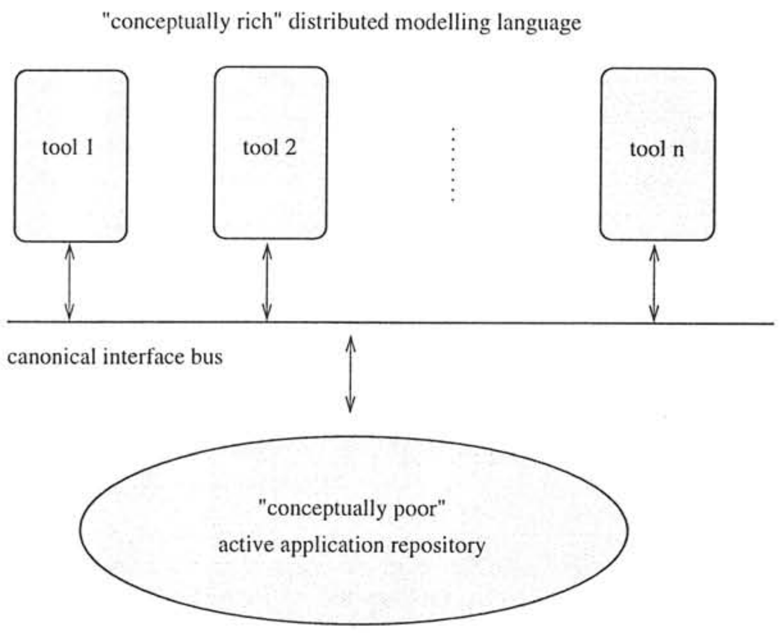

"conceptually rich" distributed modelling language
tool 1
tool 2
tool n
canonical interface bus
"conceptually poor"
active application repository

Figure 3: The architecture of pluggable visual descriptions

The application system in this architecture is composed of active actor-like components, held in a central repository. The development environment consists of tools which provide different visual descriptions. The tools are linked to the repository through a canonical interface, which forms a "plug and play" bus.

The central repository is based on a "conceptually poor" application framework, because it contains only a minimal number of relatively stable concepts. These concepts are application-domain oriented and are likely to be at a fairly low level of abstraction. These concepts are the components out of which the application system is constructed. The commands for controlling these components form the canonical interface between the application repository and the tools.

Because the resultant application structure is likely to be too complex for some participants in the development process such as users or analysts, the tools can define and maintain their own abstractions to use on their descriptions. These non-repository abstractions are shared between the tools by a mechanism in which the repository acts as a broker between the tools. This results in a distributed "conceptually rich" language implemented between the tools. Changes in this language do not require re-structuring of the application in the central repository nor changes in the central application framework.

The architecture allows for a number of different tools to be used to visualise and control an application system (requirement (i)). Because of the canonical "plug and play" interface to the application system, changes in a tool do not affect the operation of the system (requirement (ii)), although such changes may affect other tools. A high degree of integration is achieved by the tools acting as "views" to the active application system. Special mechanisms exist which broadcast all changes in the state of the application system. Tools which "care" can thus update their representations, and provide monitoring of the application functioning (requirement (iii)). Tools can share abstractions which are not in the application system through the "broker" mechanism (requirement (iv)).

The architecture was used to build ECHOES, a prototype environment where workflow applications can be constructed and modified by non-programmers. In ECHOES, a workflow application will be defined as a set of active components. The components will be instances of the comparatively stable concepts 'activity', 'clerk', 'message' and so on, which form a "conceptually poor" application framework. Volatile concepts such as 'job description' are not a part of the framework, and the fact that a clerk can perform a certain activity is specified by direct association between the two instances in the application.

Volatile concepts such as 'job description' can be quite useful when users or analysts work with the workflow application. For example, the user can specify that a clerk has a certain job description, and this will automatically make the clerk responsible for a set of activities. Later, when a new activity is introduced into the system we don't need to enumerate all clerks that can do it, we can just make it a part of a job description. To avoid hard-coding 'job description' within the application framework, we have chosen to define the concept within one of the visual description tools, the tool which handles the organisational structure. This gives us the advantage of being able to change the user interface language by replacing the 'job description' concept with a 'role' or a 'task team' when this becomes necessary. This will involve only changing the tool and not the underlying workflow application system, thus fulfilling our fourth requirement.

The system is designed in such a way that other tools can still access a particular job description, and can show it on their views to the workflow application. As far as the other tools are concerned, they would retrieve the concept through their normal mechanisms for interacting with the application. The application will in this case act as a broker and will re-direct the request for the required information to the particular tool which implements 'job description'. This mechanism implements a 'conceptually rich' language distributed between the tools forming the development environment.

The ECHOES prototype was implemented using Smalltalk VisualWorks and the generic drawing editor HotDraw. The dependency mechanism in Smalltalk was used to implement the canonical interface bus and the basic interaction mechanisms.

ECHOES was used to conduct several evaluation studies. Part of the studies included different types of changes in the tools of the environment. It was possible to modify tools in ECHOES without disturbing the workflow application. Another part of the evaluation studies included observing two non-programmers performing enhancements of two workflow applications. This part was also successful, and the subjects proved capable of modifying their workflow applications. They had no problems working with the multiplicity of visual descriptions provided in the ECHOES prototype. Details of the evaluation studies are reported elsewhere [5].

6 Limitations of the proposed approach

6.1 Limits of the independence between different tools

Having a canonical language along the "pluggability" interface bus should in theory ensure that any tools could be plugged into the system with ease, as long as they conform to this language.

However, just mixing tools is not the most effective approach to creating a CBiSE environment. The tools should be designed to work together. This means they should offer consistent interaction mechanisms such as 'dragging a symbol from a palette to a diagram represents a request for creating the relevant concept', and consistent representational conventions such as 'symbols stacked behind each other are related'. In addition, if two or more tools are to benefit from sharing a non-framework concept they should be designed to work together.

6.2 Limits of meta-enhanceability for a given application framework

The variety of concepts and paradigms used within the "conceptually rich" environment language is constrained by the semantics of the underlying "conceptually poor" application framework, since any concepts from the former have to be mapped to the latter. Therefore the level of abstraction in the framework should be carefully selected to enable a large variety of concepts to be represented by the tools of the environment. Indeed, using too high level of abstraction in the framework may introduce dependencies and rules that place unnecessary constraints on the tools. On the other hand, using too low level of abstraction will require more effort in implementing different tools.

It is also possible that the application framework will have to be up-dated after a comparatively long period of tool changes. Changes in the concepts and descriptions provided by tools are likely to cause deterioration of the integration support offered by the old application framework, and at a certain point the framework will have to be up-dated so that the integration support reaches the optimal level. An expanded version of this argument is contained elsewhere [6].

Testing the limits of changes in the concepts and descriptions provided by the tools, and the dependency between tools and underlying application framework represents an important issue, which should be a subject of further research effort.

7 Conclusion

Components encapsulate complexity and bring promises for rapid development of applications by assembling re-usable blocks. However, building effective environments to support this rapid development process is far from trivial, and requires re-conceptualising the existing environment architectures. There are many research questions to be answered in this area, which has until now remained on the periphery of CBiSE-related research.

References

[1] A. Finkelstein, S. Easterbrook, J. Kramer, and B. Nuseibeh. Requirements engineering through Viewpoints. Technical report, Imperial College, Department of Computing, 180 Queen's Gate, London SW7 2BZ, 1992.

[2] Adele Goldberg and David Robson. *Smalltalk-80: The language.* ParcPlace Systems Inc., 1989.

[3] M. Jarke, H. Nissen, and K. Pohl. Tool integration in evolving information system environments. NATURE report series NATURE-94-1, Lehrstuhl für Informatik V, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Germany, 1994. Available by email to natrep@informatik.rwth-aachen.de, or by ftp or WWW to ftp://ftp.informatik.rwth-aachen.de/pub/packages/NATURE.

[4] Don Kiely. Are components the future of software? *IEEE Computer*, 31(2):10–11, February 1998.

[5] Nikolay Mehandjiev. *User Enhanceability for Information Systems through Visual Languages.* PhD thesis, Department of Computer Science, University of Hull, HULL HU6 7RX, England, 1997.

[6] Scott Meyers. Difficulties in integrating multiview development systems. *IEEE Software*, pages 49–57, January 1991.

[7] Scott Douglas Meyers. *Representing Software Systems in Multiple-View Development Environments.* PhD thesis, Brown University, Department of Computer Science, Providence, Rhode Island 02912, USA, May 1993. Available as technical report CS-93-18.

[8] National Institute of Standards and Technology, Gaithersberg. *Reference Model for Frameworks of Software Engineering Environments*, draft version 1.5 edition, 1991.

[9] R. Prins, A. Blokdijk, and N.E. van Oosterom. Family traits in business objects and their applications. *IBM Systems Journal*, 36(1):12–31, 1997.

[10] William Tepfenhart and J. Cusick. A unified object topology. *IEEE Software*, pages 31–35, January 1997.

[11] L. Young and N. Mehandjiev. The 'knowledge curtain': Beyond the communication gap. In E. Dubois, A. L. Opdahl, and K. Pohl, editors, *Proceedings of REFSQ*98: Fourth International Workshop on Requirements Engineering: Foundation for Software Quality, June 8-9, 1998, Pisa, Italy*, 1998.

Layered Components [*]

Luc Claes

Computer Science Institute
University of Namur - Belgium
lclaes@info.fundp.ac.be

Abstract. *We describe a generic software environment intended to assist the construction of applications by the composition of reusable components. Two classes of components (and their related systems) are clearly distinguished: operational components (or components in the 'classical', technological sense) and descriptive components, encapsulating meta-data about, amongst others, their operational counterparts. Components are viewed as actors in a communicating entities system. Communications capabilities and needs are abstracted as a set of layers, each layer denoting a delimited conventions set and satisfying or dictating functional requirements.*
A case study describes how we are using - or intend to use - our models in real life applications. The selected case consists in the interconnection of two, otherwise incompatible, CASE tools.

1. Introduction

Every time our 'universal standardization' dreams are going away, we have to face that frustrating reality: software construction by components composition involves many heterogeneity-related issues: components technologies incompatibilities, frozen legacy applications, multiplicity of the interconnections and coding norms, not to mention the impediments raised by the various development 'culture gaps'. But maybe are we lucky in our misfortune: investigating the methods that could allow us to cope with that heterogeneity, could lead us to discover novel components design and/or selection rules favoring a better reusability.

In the next few pages, we roughly describe a software environment, allowing components to be assembled in a generic way by applying interoperability rules. Those rules assume a layered decomposition of inter-components communications. Furthermore, we evaluate the hypothesis suggesting that layered decompositions, of frequent use in data communications models (such as the ISO Open Systems Interconnection reference model [7]), are also applicable, at the cost of concepts generalizations, to the software components connectivity modeling.

Every communication implies the explicit or - more frequently - implicit sharing of a number of conventions between the communicating partners. By modeling that elementary principle and its associated rules, we expect to bring some useful components composition guidelines to the fore.

The software architecture described in this paper is currently tried out with the interconnection of various CASE tools. We will investigate our tentative experimentation plan.

2. Conceptual Background

2.1. Descriptive vs. Operational Components.

If it is quite usual, and natural, to conceptually isolate the active part of a component from its descriptive part - its 'meta-data' - we suggest here a more drastic approach, by granting the 'component' status to the meta-information as well.

[*] This work is founded by the European Community under ESPRIT Reactive Long Term Research 21.903 CREWS (Cooperative Requirements Engineering with Scenarios)

Components are divided in two classes: descriptive components (residing in a descriptive system) and operational components (residing in an operational system). Both systems are having their own ontology while interacting closely.

- *Operational* components are the actual actors, carrying out, by their conjugate actions, an expected 'run time' functionality. They are components in today's classical (technological) sense [12].

- *Descriptive* components encapsulate specifications. Those specifications are restricted to the communication-oriented framework described in the next paragraph. A descriptive component characterizes a *communicating entity*. A communicating entity is either an operational component or another individual intervening in the communication process while remaining somewhat 'out of control' in respect to our environment (Amongst others: operating systems services, legacy or otherwise existing applications, remote objects, hardware devices, operating system services and, more generally, the communication environment.)

The descriptive/operational separation allows our models to capture information about all relevant communication actors. Meta-data being made accessible as individual components as well, could be stored locally or remotely in components catalogs or repositories. A componentized software configuration needs to be built and validated at the 'descriptive' level before being instantiated and activated at the operational level.

2.2. (Descriptive) Components Layering.

Descriptive components are organized as a set of layers, reflecting the communication structure of a communicating entity.

Each layer has a set of properties, of which we selected:

- The **Protocol**. In a much broader sense than in data transmissions, the protocol is defined here as a *delimited set of conventions*. It could be a programming interface, a classical communication protocol, a well-defined set of assumptions made about the environment, etc. A shared protocol is a necessary condition for a communication to take place. The protocols are only identified, localized in a taxonomy allowing specialization / generalization relationships. The model does not capture the precise protocol semantics.
- **Polarity**. Does the communicating partners, at a given layer, have an identical status (= neutral polarity) or are they 'asymmetrical', like, for example, in client / server or master / slave relationships? Arbitrarily, the 'originating' actor is given a positive polarity, while the 'answering' one has a negative polarity.
- **Multiplicity** expressed as an interval [m, M]. That property synthesizes two sub-properties: (i) does this layer act as a requester ($m > 0$) or as a provider ($m = 0$ or undefined) of the service identified by the protocol (ii) does this layer support multicasting ($M > 1$ or undefined), by being able to serve multiple requesters simultaneously.
- Beyond those intra-layer properties, an inter-layer relationship expresses dependencies of the kind 'Layer x **uses** the services provided by Layer y' defining a directed acyclic graph, of which the classical 'protocol stack' is a particular case.

The properties we have described participate in the elaboration of a rules system, constituted of a set of functional requirements that each component layer will in turn satisfy or require. Ideally, all conditions necessary to achieve the components interoperations should be expressed, even the most implicit one, starting from the lower level aspects (operating system services, media access, processors, virtual machines...) up to the communication semantic aspects.

3. Case Study

3.1. Problem Statement.

Let us introduce an experimentation plan where the actors are various CASE tools, and of which the central element is a graphical/textual editor allowing users to express specifications using the formal requirements engineering language *Albert-II* [2]. Our objective is to set up interconnections between that software application and an array of somewhat related tools, like specification animators, theorem provers, traceability tools, etc. Concretely, we are currently implementing a 'bridge' between the *Albert* Editor and the requirements engineering environment *Pro-Art* [10]. Pro-Art is used in order to capture trace information throughout the Albert-II specification process.

We consider successive transitions starting from a straightforward static components assembling and ending with a more ambitious, almost automatic and optimized solution, following a 4-steps schema.

Let us first examine our problem given information by sketching the initial descriptive components. It is worth to mention that many simplifications have been introduced: for example, the user-interface is not modeled, protocols are too generic and most low-level services are oversimplified.

In order to describe the various layers, we use the graphical syntax of Figure 1.

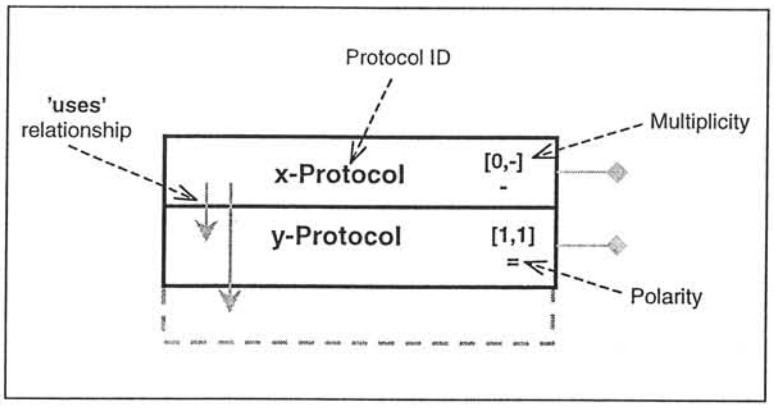


Figure 1 Graphical Syntax

We are considering the central piece of software (the *Albert* editor) only with respect to its inter-applications connectivity. That connectivity was implemented by setting up a generic communication mechanism, offering a controlled access to all Albert editor's relevant/useful methods, properties and events. We expect that no restrictive assumption be made in this interface definition concerning the potential software applications that could potentially use it.

As the editor did already represent a respectable amount of code, we were implicitly led to a technological solution considering that investment (C++, MFCs, Win32,...). We were also almost inevitably guided to implement a COM connectivity (Component Object Model: the foundation of Microsoft's OLE and ActiveX [1]), materialized by COM-compliant interfaces (custom, 'early bound' interfaces and automation, 'late bound' interfaces). However, obviously, our models are independent from the components technology.

Our - now 'open' - editor could be represented, as the set of communication layers of Figure 2:

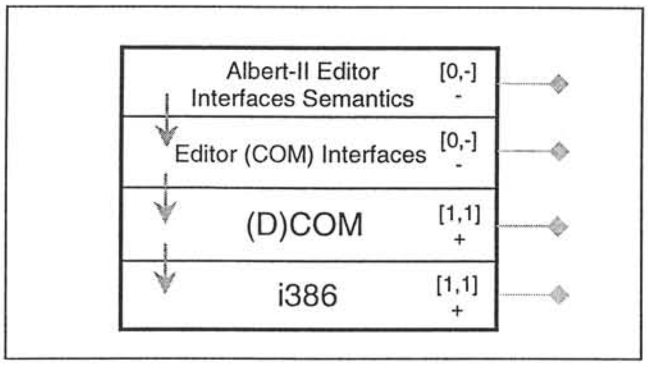


Figure 2 - Albert-II Editor Descriptive Component

The different communication levels, each dictating or satisfying a necessary interoperability condition, could be roughly described as follows (Table 1), starting from the most 'semantic' and ending with the most 'physical' level.

Layer	Description	Specified/described by
Albert-II Interface Semantics	Albert-II language semantics (subset)	Intuitive and formal semantics ; metamodel
Editor COM Interfaces	Albert Editor custom COM interfaces	IDL specification; Interface dynamic behavior formal or intuitive specification.
DCOM	Distributed Component Object Model operating environment	Microsoft documents
i386	Processor: data format, instructions set	Intel documents

Table 1 - Albert-II Editor Layers Description

To illustrate the properties values:

- A layer implements the *Editor (COM) Interfaces 'protocol'* with a negative polarity, meaning it acts as a 'server' for potential 'clients': it *offers* a COM interfaces implementation. The [0,-] indicates that no communication partner is required, but that an indefinite number of such partners could be 'served'.
- Another layer indicates that the software requires the presence of a DCOM environment. Here, it acts as a 'client' (positive polarity) and the service is required ([1,1] multiplicity).

The *'PRO-ART'* software we intend to connect to the Albert-II language editor is described, from our communication point of view, as follows (Figure 3). Beyond their actual application-level semantics, Pro-Art services are exposed through a (proprietary) object-model protocol, using a classical TCP/IP transport.

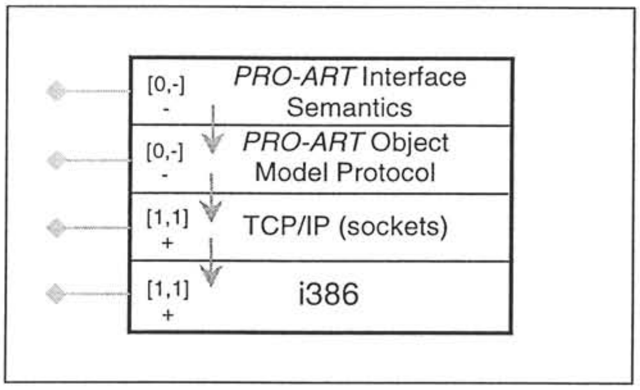

[0,-]
-
PRO-ART Interface Semantics
[0,-]
-
PRO-ART Object Model Protocol
[1,1]
+
TCP/IP (sockets)
[1,1]
+
i386

Figure 3 *PRO-ART* Descriptive Component

3.2. Introducing Bridgelets: Protocol(s) Converter Components.

Interoperability between the PRO-ART software and the Albert editor is obtained by creating one or more (here: 2) intermediate components implementing the necessary communication adaptations, up to the higher levels of 'incompatibility' (here: semantic).

The first component - the PRO-ART/COM *bridgelet* (Figure 4) - is designed in order to offer a reusable front-end to PRO-ART, masking the complex proprietary object model protocol and hiding the TCP/IP low-level housekeeping as well. Note that this bridgelet has no semantic-level knowledge. It acts only as a 'syntactical' bridge, carrying out straightforward translations.

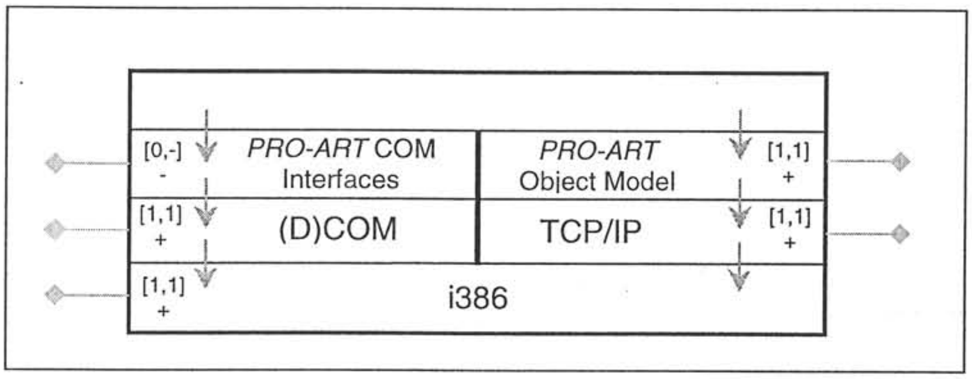


Figure 4 - PRO-ART / COM 'bridgelet' (Descriptive) Component

A second component - the *Albert* editor / *PRO-ART* bridgelet (Figure 5) - implements the actual high-level bridging. Noteworthy, the low-level software technology is here somewhat different: Java is used as implementation language, instead of a machine-dependant C++ compiled solution.

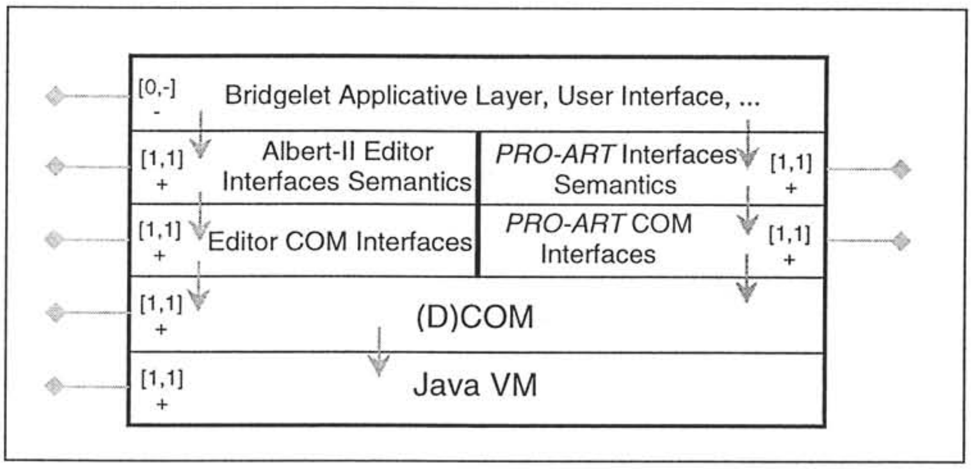

[0,-]
-
Bridgelet Applicative Layer, User Interface, ...
[1,1]
+
Albert-II Editor Interfaces Semantics
PRO-ART Interfaces Semantics
[1,1]
+
[1,1]
+
Editor COM Interfaces
PRO-ART COM Interfaces
[1,1]
+
[1,1]
+
(D)COM
[1,1]
+
Java VM

Figure 5 - *Albert-II* / *PRO-ART* 'Bridgelet' Descriptive Component

The overall configuration, satisfying all functional requirements could be described as follows (Figure 6).

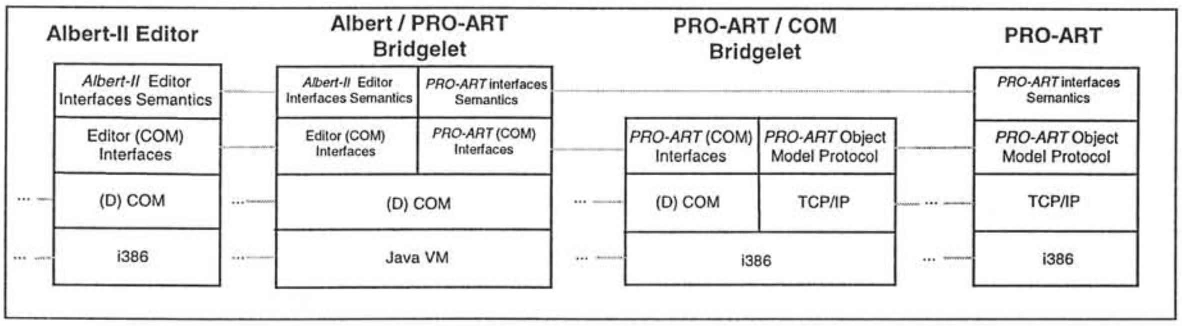

Albert-II Editor
Albert-II Editor Interfaces Semantics
Editor (COM) Interfaces
(D) COM
i386
Albert / PRO-ART Bridgelet
Albert-II Editor Interfaces Semantics
PRO-ART interfaces Semantics
Editor (COM) Interfaces
PRO-ART (COM) Interfaces
(D) COM
Java VM
PRO-ART / COM Bridgelet
PRO-ART (COM) Interfaces
PRO-ART Object Model Protocol
(D) COM
TCP/IP
i386
PRO-ART
PRO-ART interfaces Semantics
PRO-ART Object Model Protocol
TCP/IP
i386

Figure 6 - Overall (Descriptive) Configuration

3.3. Experimentation plan

Let us describe a 4-phases experimentation plan, reflecting a bottom-up process.

Starting from a purely operational architecture, we gradually enrich the descriptive system. A components composition framework is progressively constructed.

3.3.1. Phase 1. Specific, 'Manual' Components Composition.

During phase 1, existing applications and operational components are assembled statically at design time. The Albert editor is modified in order to act as a container for 'well-known' components. Those components are instantiated and interconnected by specific code.

3.3.2. Phase 2. Generic, 'Manual' Components Composition.

Phase 2 targets to establish a solid conceptual foundation for the next phases. The graphical, intuitive syntax of Figure 1 is extended and formalized:

- The descriptive and operational systems ontologies are currently modeled using the *Telos* notation [8]. Telos is a conceptual modeling language in the family of entity-relationship models, supporting unlimited classification levels (meta-classes) and treating attributes as full-fledged objects.
- The models inferential aspects and integrity constraints are tentatively expressed using *Object-Z*, an object-oriented extension to the specification language Z [3].

A simplified components composition framework is set up, allowing components configurations to be constructed and validated at the descriptive level. Simple configurations activation, with adequate operational components instantiations is supported.

3.3.3. Phase 3. Generic, Assisted Components Composition.

A 'Phase-3' framework is able to generate components configurations (semi-) automatically, on the basis of (i) requirements expressed by the user as a given set of descriptive components (ii) a formalized knowledge about the operating environment and available resources (that is, catalogs of locally or remotely available components). Criteria taken into account at this stage are only targeting basic components interoperability: communication necessary conditions, functional requirements have to be satisfied.

Regarding our case study, this means that, given an initial, incomplete configuration containing two descriptive components ('Albert Editor' and 'PRO-ART'), the framework has to infer a complete, valid configuration by fetching components satisfying all constraint, (for example the 'Albert/PRO-ART' and 'PRO-ART/COM' bridgelets, but other solutions could be found) in components catalogs.

An example of such configuration automation, somewhat restricted, is described by [4].

3.3.4. Phase 4. Generic, Assisted, Optimized Components Composition.

The 'Phase-3' framework could generate, for a given requirements set, several valid configurations (i.e. satisfying all constraints). During this fourth phase, we intend to integrate the ability to consider some non-functional requirements, in order to select not only a 'valid' configuration, but also the 'best' one. Interesting criterions would be, amongst others, connections cost, performance, security, etc.

4. Conclusions

We described a 'components world', with a partial map like (Figure 7).

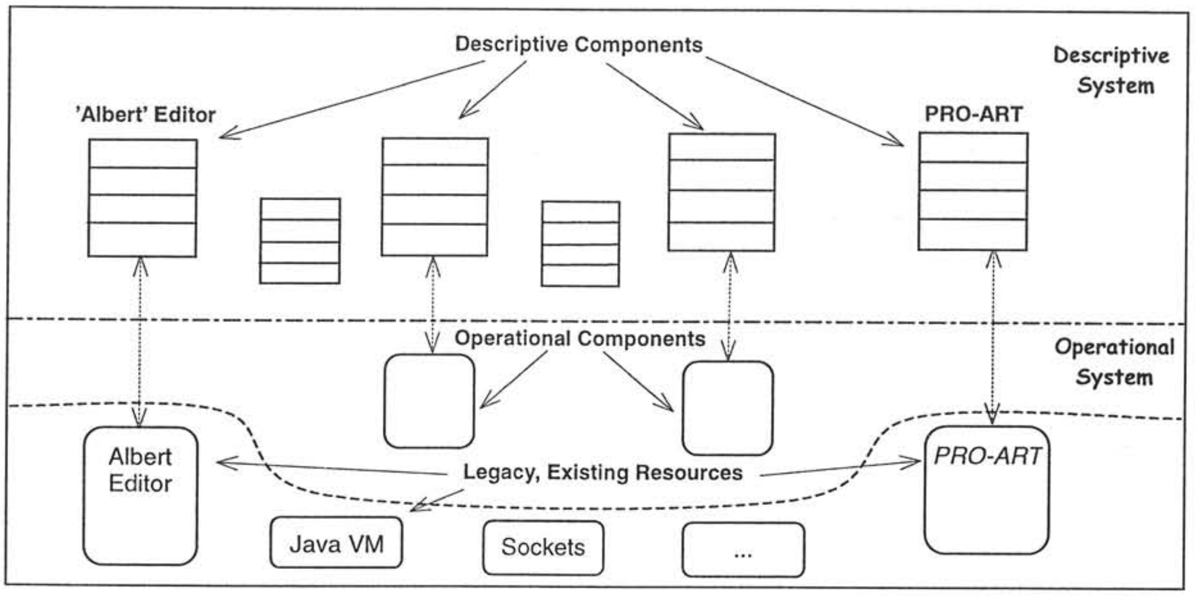


Figure 7

However, our map contains many white areas, of which we could mention:

- In order to break down the components specification into layers, we need an accurate set of criteria, like 'substitutability'. Nowadays, that decomposition lacks precision, particularly at the higher, semantic levels.
- Some fundamental dynamic aspects are not taken into account: initialization, activation, finalization, components sharing among multiple configurations, etc.
- Inter-layer relationships deserve a more precise modeling. By simply indicating that 'Layer A **uses** Layer B', the information loss is often unacceptable.

Acknowledgements This work is founded by the European Community under ESPRIT Reactive Long Term Research 21.903 CREWS (Cooperative Requirements Engineering with Scenarios). The author would like to thank Eric Dubois and Patrick Heymans for their comments on an earlier version of this paper.

References

[1] Kraig Brockschmidt, *Inside OLE*, 2nd edition, Microsoft Press, 1995.

[2] Philippe Du Bois, *The Albert-II Language: On the Design and the Use of a Formal Specification Language for Requirements Analysis*, PhD Thesis, University of Namur (Belgium), September 1995.

[3] Roger Duke, Paul King, Gordon Rose, Graeme Smith, *"The Object-Z Specification Language Version 1"*, Software Verification Research Centre Technical Report 91-1, University of Queensland, May 1991.

[4] David Hovel, *'Using Prolog in Windows NT Network Configuration'*, presented at *Practical Applications in Prolog'95*, Paris, April 1995.

[5] Michael R. Genesereth, Narinder P. Singh, Mustafa A. Syed, *"A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation"*, 3rd CIKM Workshop on Intelligent Information Agents, Gaithersburg, Maryland, December 1994.

[6] Sandra Heiler, Renée J. Miller, Vincent Ventrone, "*Using Metadata to Address Problems of Semantic Interoperability in Large Objects Systems*", Proceedings of the First IEEE Metadata Conference, Silver Spring, April 1996.

[7] International Organization for Standardization, *"OSI Basic Reference Model"*, ISO 7498-1, 1992. (Also known as ITU-T X200)

[8] John Mylopoulos, Alex Borgida, Matthias Jarke, M. Koubarakis, *"Telos - a language for representing knowledge about information systems"*, in ACM Transactions on Information Systems, 8, 4, 1990, pp. 325-362.

[9] Xavier Pintado, *"Gluons and the Cooperation between Software Components"*, in O.Nierstrasz & D. Tsichritzis *Object-Oriented Software Composition*, Prentice-Hall, 1995.

[10] Klaus Pohl, *Process-Centered Requirements Engineering*, Research Studies Press Ltd., 1996.

[11] Mary Shaw & David Garlan, *Software Architecture - Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[12] John Udell, "Componentware", *Byte*, May 1994, pp. 46-56.

The execution model: a component-based architecture to generate software components from conceptual models

Jaime Gómez [†], Emilio Insfrán [‡], Vicente Pelechano [‡], Oscar Pastor [‡]

[†] Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante
C/ San Vicente s/n
03690 San Vicente del Raspeig. Alicante (Spain)
{jaime@dlsi.ua.es}

[‡] Departament de Sistemes Informàtics i Computaciò
Universitat Politècnica de València
Camí de Vera s/n
46071 Valencia (Spain)
{einsfran, pele, opastor@dsic.upv.es}

Abstract

A basic problem of software development is how to derive executable software components from requirements, and how this process could be systematized. Current object-oriented CASE tools support various graphical notations for modeling an application from different perspectives. However, the level of built-in automation is relatively low as far as how to produce a final software product.

In this paper, we present a component-based architecture based on a formal object-oriented model which gives the pattern for obtaining software components. This software components set up the basis for a software prototype that is functionally equivalent to the conceptual model in an automated and reusable way.

1. Introduction

Nowadays OO methodologies like OMT[1], OOSE[2] or Booch[3] are widely used in industrial software production environments. Industry attempts to provide unified notations such as the UML[4] proposal which was developed to standarize the set of notations used by the most well-known existing methods. Even if the attempt is commendable, this approach has the implicit danger of providing users with an excessive set of models that have overlapping semantics without a methodological approach. Following this approach we have CASE tools such as Rational ROSE[5] or Paradigm Plus[6] which include code generation from the analysis models. However if we go into depth with this proposed code generation feature, we find that it is not at all clear how to produce a final software product which is functionally equivalent to the system description collected in the conceptual model. This is a common weak point of these approaches. Far from what is required, what we have after completing the conceptual model is nothing more than a template for the declaration of classes where no method is implemented and where no related architectural issues are taken into account.

In order to provide an operational solution to the related problem, we think that the idea of clearly separating the conceptual model level, centered in *what* the system is, and the execution model, intended to give an implementation in terms of *how* the system is to be implemented, constitutes a good starting point.

In this paper we present a component-based architecture based on a formal object-oriented model which gives the pattern for obtaining software components. This software components set up the basis for a software prototype that is functionally equivalent to the conceptual model in an automated and reusable way.

The starting point is the OO-Method proposal. OO-Method is an OO methodology that allow us to collect the relevant system properties. The main feature of OO-Method is that developers' efforts are focused on the conceptual modeling step, where analysts capture system requirements. Once we have an appropiate system description, a formal OO specification is automatically obtained. This specification provides a well-structured framework that enables the building of an automatic code generation tool from a component-based perspective.

This document is organized as follows: section 2 present a short description of the main OO-Method features describing the diagrams used to capture the system properties in order to produce what we call a conceptual model. Subsequently, we will show which is the underlying semantic to represent this model in any target software development environment according to an abstract execution model. Section 3 describes the component-based architecture to produce the software components that allow us to link the conceptual model with the abstract execution model. To illustrate this step, we use Java Beans[7] as a concrete component-based model. Finally section 4 presents the conclusions.

2. The OO-Method proposal

The OO-Method was created on the formal basis of OASIS, an OO formal specification language for Information Systems[8]. Basically, we can distinguish two components: the conceptual model and the execution model.

2.1. Conceptual Model

Conceptual modeling in OO-Method collects the Information System relevant properties using three complementary models:

- **Object Model**: a graphical model where system classes including attributes, services and relationships (aggregation and inheritance) are defined. Additionally, agent relationships are introduced to specify who can activate each class service (client/server relationship).

- **Dynamic Model**: another graphical model to specify valid object life cycles and interobjectual interaction. We use two kinds of diagrams:

    - *State Transition Diagrams* to describe correct behaviour by establishing valid object life cycles for every class. By valid life, we mean a right sequence of states that characterizes the correct behaviour of the objects.

    - *Object Interaction Diagram:* represents interobjectual interactions. In this diagram we define two basic interactions: triggers, which are object services that are activated in an automated way when a condition is satisfied, and *global interactions*, which are transactions involving services of different objects.

- **Functional Model**: is used to capture semantics attached to any change of an object state as a consequence of an event occurrence. We specify *declaratively* how every event changes the object state depending on the involved event arguments (if any) and the object's current state. We give a clear and simple strategy for dealing with the introduction of the necessary information. This is a contribution of this method that allows us to generate a complete OASIS specification in an automated way. More detailed information can be found in [9].

From these three models, a corresponding formal and OO OASIS specification is obtained using a well-defined translation strategy. The resultant OASIS specification acts as a complete high-level system repository.

2.2. Execution Model

Once all the relevant system information has been specified, we use an execution model to accurately state the implementation-dependent features associated to the selected object society machine representation.

In order to easily implement and animate the specified system, we predefine a way in which users interact with system objects. We introduce a new way of interaction, close to what we could label as an OO virtual reality, in the sense that an active object immerses in the object society as a member and interacts with the other society objects. To achieve this behaviour the system has to:

1. **identify the user** (an access control): logging the user into the system and providing an **object system view** determining the set of object attributes and services that it can see or activate.

2. **allow service activation**: finally, after the user is connected and has a clear object system view, the users can activate any available service in their worldview. Among these services, we will have system observations (object queries) or events or transactions served by other objects.

   Any service activation has two steps: build the message and execute it (if possible).

   In order to build the message the user has to provide information to:

1. **identify the object server**: The server object existence is an implicit condition for executing any service, unless we are dealing with a new event[1].

2. **introduce event arguments**: asking for the arguments of the event being activated (if necessary).

Once the message is sent, the service execution is characterized by the ocurrence of the following sequence of actions in the server object:

1. **check state transition**: verification in the object State Transition Diagram (STD) that a valid *transition* exists for the selected service in the current object state.

2. **precondition satisfaction**: the precondition associated to the service must hold

   If 1 or 2 don't hold, an exception will arise and the message is ignored.

3. **valuation fulfillment**: the induced event modifications (specified in the Functional Model) take place in the involved object state.

4. **integrity constraint checking in the new state**: to assure that the service execution leads the object to a valid state, the integrity constraints (static and dynamic) are verified in the final state. If the constraint does not hold, an exception will arise and the previous change of state is ignored.

5. **trigger relationships test**: after a valid change of state, the set of condition-action rules that represents the internal system activity is verified. If any of them hold, the specified service will be triggered.

The previous steps guide the implementation of any program to assure the functional equivalence between the object system specification collected in the conceptual model and its reification in a programming environment.

3. Component-based Architecture for the execution model

The component-based architecture for execution model in OO-Method methodology is created using the familiar model-view-controller (MVC) paradigm[10]. In other words, we have a specific non-visual component which encapsulates the details of the data model, GUI visual components handle the presentation of information, and non-visual control components encapsulate the application logic according to the OASIS specification.

Our proposal is to define a logical three-tiered architecture that is MVC-compliant. We can obtain all the information we need to automatically create the software components in these tiers. We will carry this process extracting from OO-Method conceptual model diagrams each piece of information that is relevant for a specific component as it is shown in figure 1.

Java Beans component architecture provides the context to implement the major types of services for software components proposed in figure 1. To show this, we have to explain how is the process to implement these components in each tier using Java Beans.

[1] Formally, a new event is a service of a metaobject that represents the class. The metaobject acts as object factory for creating individual class instances. This metaobject (one for each class) has the class population attribute as a main property, the next oid and the quoted new event.

*Complex Classes → (Inheritance, Aggregation) **Services → (Events, Transactions)

		Components ↓	CONCEPTUAL MODEL			
			Object Model	Dynamic Model		Funtional Model
				STD	OID	
EXECUTION MODEL	Persistent Tier (Model)	Persistent Mediator	Simple Classes Complex Classes*			
	Interface Tier (View)	Access Control	Simple Classes Complex Classes* Agents			
		System View	Simple Classes Complex Classes* Services**		Interactions	
		Service Activation	Event Arguments			
	Application Tier (Controller)	Domain Classes	Atributes Services** Constraints	Preconditions State Transitions	Triggers	Valuations

Figure 1: linking Conceptual Model with the Execution Model

3.1. Persistence Tier

This tier maps the model. It is composed by non-visual Java Bean component called *Persistent_Mediator*, a generally reusable object-oriented interface for conventional relational databases. This component implements the methods for saving, deleting and retrieving system domain objects that are stored in a RDBMS server as is explained in [11]. The *Persistent_Mediator* class has the following general structure:

```
class Persistent_Mediator extends java.lang.Object{
       void delete ();
       void save ();
       void retrieve ();
}
```

JDBC classes are used for proper interaction with the involved RDBMS servers in the implementation of these methods. Particularly, each class in the OO-Method object model diagram is mapped to a table in the relational database. The resulting table consists of a column for storing a unique object's ID and a column for each atributte declared in the class definition. We mapped simple types directly to the equivalent data type provided in the relational database and replaced object references with unique object IDs.

To retrieve objects from the database, the *Persistent_Mediator* component generates the appropriate SQL query and returns the data as a list of objects. Likewise, when storing objects in the database, the component creates an SQL update or insert statement.

3.2. Interface Tier

This second tier maps the view. It is composed by three GUI Java Beans components. These components handle the presentation of information which help to implement the interaction between the user and the application, following the underlying semantic of our object model. The figure 2 shows the basic Java Bean specification to these components.

		Access_Control (java.awt.Container)	System_View (java.awt.Frame)	Service_Activation (java.awt.Dialog)
BeanInfo	Properties	Oid Password Classname	Class_list Services_list	Argument_list

	Events	SetOid()
		GetOid()
		SetPassword()
		GetPassword()
		SetClassname()
		GetClassname()
			SetClass_list()
			GetClass_list()
			SetServices_list()
			GetServices_list()
				SetArgument_list()
				GetArgument_list()

Figure 2: basic Java Bean specification for interface tier components

The BeanInfo class is used to describes these Beans. This class allows that the properties and events of a generic Bean can be determined through the introspection mechanism.

The `Access_Control` component is defined as container Bean that allow users to be identified in the system as was explained in section 2.2. The BeanInfo class defines `oid`, `password` and `classname` properties that provide the information required to this process. This class also defines the events to manipulate the `setter/getter` methods asociates to the properties defined.

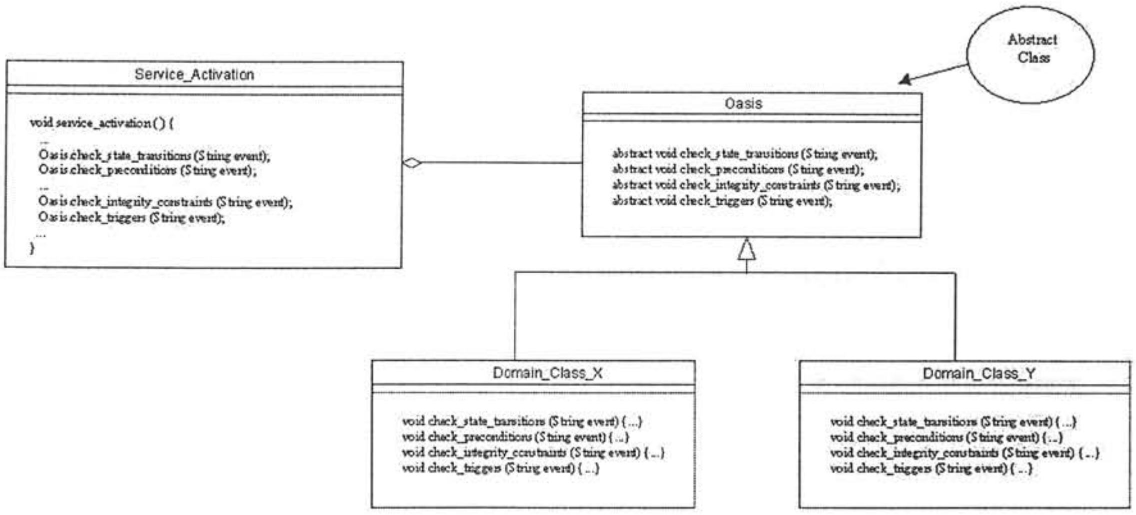
The `System_View` component is defined as a customizable Bean from the base class `java.awt.Frame`. In this case, the BeanInfo class defines the `class_list` and `services_list` properties that are needed to create an specific system view. Particularly, the `class_list` will constitute the menu bar (classes the user is allowed to interact with), which will contain as a menu items the information provided by `services_list` property (services that the user can activate for each class). The events defined in the BeanInfo class allow to manipulate the content of these properties.

Finally, the `Service_Activation` component is defined as a customizable Bean from the base class `java.awt.Dialog`. The BeanInfo class defines the `argument_list` property. This property contains the relevant service arguments for a specific service activation. The events to manipulate this property are described in the figure 2.

3.3. Application Tier

The last tier maps the controller. This tier is composed by non-visual control components (one for each domain class in the conceptual model). These components encapsulate the application logic which follow our underlying semantic of execution model structure. In concrete, the domain classes can model the behavior of the system using the state pattern which was described in [12]. According to this generic pattern, we can implement the state-dependent behavior when a *service activation request* is called by an agent class.

To illustrate this process, figure 3 contains a class diagram using UML for representing classes and relations, and Java for describing method definitions. The *Service_Activation* class describes objects that are used to activate the services offered by domain classes. The Oasis class is an abstract class that really acts as a kind of placeholder. Because it is abstract, it is not possible to create class instances. Instead, when a abstract class method is referenced, the Oasis class simply delegates a request to one of the objects create from a subclass. Then, each subclass defines the appropiate behavior implementing the abstract methods for a specific server class.



```
void service_activation() {
  ...
  Oasis.check_state_transitions (String event);
  Oasis.check_preconditions (String event);
  ...
  Oasis.check_integrity_constraints (String event);
  Oasis.check_triggers (String event);
  ...
}
```

```
abstract void check_state_transitions (String event);
abstract void check_preconditions (String event);
abstract void check_integrity_constraints (String event);
abstract void check_triggers (String event);
```

```
void check_state_transitions (String event) {...}
void check_preconditions (String event) {...}
void check_integrity_constraints (String event) {...}
void check_triggers (String event) {...}
```

```
void check_state_transitions (String event) {...}
void check_preconditions (String event) {...}
void check_integrity_constraints (String event) {...}
void check_triggers (String event) {...}
```

Figure 3: UML class diagram to model the state-dependent behavior in domain classes

To properly implement this pattern in our context, the OASIS class must be defined as a **Java interface**. This interface specifies the necesary services to support the execution model structure as was explained in the section 2.1. These methods must be implemented in each class that belong to the system domain. The following paragraph shows the OASIS interface structure:

```
import java.awt.*;
interface Oasis {
      void check_precondition (string event)
      void check_state_transition (string event)
      void check_integrity_constraint (string event)
      void check_trigger (string event)
}
```

Moreover, in order to ensure that the implementation of the domain classes has persistence facilities, each class is declared as an extension of *Persistent_Mediator* class. The following paragraph shows a generic class domain declaration:

```
import java.awt.*;
public class X extends Persistent_Mediator implements OASIS {...}
```

Every *service activation request* will call to the server class methods that implements the effect of the event on the object state. In concrete, checking the state transition correctness and method preconditions. If this checking process succeeds, the object change of state is carried out according to the funcional model specification. We finish the *service activation execution* by verifying the integrity constraints and the trigger condition satisfaction in the new state. Finally, the update of the object change of state become valid through the services inherited from the *Persistent_Mediator* component in the selected persistent object system.

4. Conclusions

Over the past few years, constructing applications by assembling reusable software components has emerged as highly productive and widely accepted way to develop custom applications. We have presented a component-based architecture that allow developers to generate software components that can be dynamically combined together to create a software prototype from the conceptual model step. To achieve this goal, we use well-defined OO-Method methodological framework, which properly connect OO conceptual modeling and OO software development environments using a component-based model. The most relevant contributions of this paper are the following:

- an operational implementation where a concrete execution model is obtained from a process of *conceptual model translation* using a component-based architecture.
- a set of precise object-oriented components that can be reusable and where the use of a formal specification language as a high-level data dictionary is a basic characteristic.

5. References

[1] Rumbaugh J. et al. W. *Object Oriented Modeling and Design*. Englewood Cliffs, Nj. Prentice-Hall. 1991.

[2] Jacobson I. et al. G. *OO Software Engineering , a Use Case Driven Approach*. Reading, Massachusetts. Addison -Wesley.

[3] Booch,G. *OO Analysis and Design with Applications*. Addison-Wesley, 1994.

[4] Booch G., Rumbaugh J., Jacobson I. *UML. v1*. Rational Software Co., 1997.

[5] Rational Software Corporation. Rational Rose User's Manual, 1995.

[6] Platinum Technology, Inc., Paradigm Plus: Round-Trip Engineering for JAVA, White Paper from Platinum Web Site: http://www.platinum.com/. 1997.

[7] Sun MicroSystems. *Java Beans: A component Architecture for Java*, White Paper from Sun MicroSystems Web Site: http://java.sun.com/. 1996.

[8] Pastor O.,Hayes F. and Bear S. *OASIS: An object-oriented specification language*. In P. Loucopoulos, editor, Proceedings of the CAiSE'92 conference, pp. 348-363, Berlin, Springer, LNCS 593 (1992).

[9] Pastor O. et al. *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods*. In Antoni Olivé and Joan Antoni Pastor editors, Proceedings of CAiSE'97 conference, pp. 145-158 ,Berlin, Springer-Verlag, LNCS 1250. June 1997.

[10] A. Goldberg and D. Robson *Smalltalk-80, The Language and It Implementation*. Addisson-Wesley, Reading, Mass., 1983.

[11] Argawal S., Jensen R., and Keller A. M. *Architecting object applications for high performance with relational databases.* In OOPSLA Workshop on Object Database Behaviour, Benchmarks, and Performance, Austin, 1995

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides *Design Patterns: Elements of reusable Object-Oriented Software.* Professional Computing Series. Adisson-Wesley, Reading, MA, October 1994.

Configuration Management in Component-Based Development Projects

Gunnar Arneberg, Knut Sagli and Dag I.K. Sjøberg
Department of Informatics, University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
email: {gunnara, knutsag, dagsj}@ifi.uio.no

Abstract

In component-based development, the unit of design and implementation is the component, *while the unit of operation in software configuration management(SCM) tools is the* file. *A case-study carried out in a major Norwegian component-based development project identified inconsistency problems and extra workload on the programmers due to the mismatch between the lower abstraction level of files in SCM and the higher abstraction level in other development activities. To help solve this mismatch problem, we propose a model for SCM that raises the abstraction level from files to components. This paper also describes a prototype tool that, by adding a layer to an existing SCM tool, uses the component as the unit for SCM. An evaluation of the model and the prototype is given in the context of the studied development project.*

1 Introduction

Component-based development has recently gained increased popularity in the industry. The growing interest in component technology is also reflected in the research literature [1, 6]. Component models such as Microsoft's COM and Javasoft's JavaBeans define components as the fundamental application building blocks. Components are built from collections of files and several components may share files.

Keeping track of changes to software and combining versions of software components to constitute valid applications are referred to as *software configuration management* (SCM) [14, 2, 4]. Although the idea of performing SCM on higher level units is well described in the literature [10, 13, 4], the authors have been unable to find work that addresses SCM in component-based development projects in particular. Through close contacts with several companies that exploit component technology, we have experienced that they still use conventional file-based tools to support the SCM processes. An identified problem is the mismatch between the lower abstraction level of files in SCM (defining versions and configurations, consistency checking and teamwork coordination) and the higher abstraction level in other development activities (design, implementation and documentation of application components). Creating and maintaining the necessary mapping between these two levels are both cumbersome and error-prone, and may be very complicated for large applications.

We have studied the problem of SCM over a period of three months in Nauticus, a component-based development project with over 250 components developed over a four year period by an average of 18 developers, at the foundation Det Norske Veritas (DNV) [3]. Nauticus will support DNV's work on maritime classification, certification and verification in their about 300 offices in 100 countries.

In the Nauticus development environment, we defined a model for SCM where the component is the unit for version control, thus overcoming the abstraction level mismatch. We also built a prototype tool based on this model that provides updated component specific information, such as version history, dependencies and changes. We have anecdotal evidence for our claim that such a tool will help developers to better understand a system and improve cooperation and coordination among them.

The remainder of this paper is structured as follows. Section 2 describes the SCM in the Nauticus project. Section 3 presents our model for SCM in component-based development environments.

Section 4 describes the implementation of a prototype tool that supports this model. Section 5 covers the evaluation of the model and the tool. Section 6 describes related work. Section 7 concludes and sketches future work.

2 Configuration Management in Nauticus

The Nauticus applications are constructed with COM-components as building blocks, using C and C++ as the main implementation languages, and Visual Basic and Java as complementary languages. The SCM tool currently being used in the project is Microsoft's Visual SourceSafe [11], a check-in/check-out tool operating on a versioned file repository. In addition, internally developed scripts are used for build and change management.

A COM-component is built from a set of files. Some of these files may be shared with other components. The project experienced inconsistency problems because the developers used different versions of shared files in their development on the local workstations. The building of all components with one set of common files then failed. To help solve this problem, it was decided to build the entire system daily. The developers now use the source code of the latest build as basis for development.

Although the shared file inconsistency no longer is a major problem, another form of SCM related inconsistency exists, as illustrated in Figure 1. In that example, three co-relating components, A, B and C, form an application. A uses B and C, while B uses C. Since only one version of a component can exist in one configuration, the inconsistency occurs when A demands C in one version and B demands C in another version. With few components, this is reasonably straightforward to handle, but with over 250 components, as in Nauticus, detecting these version inconsistencies becomes very difficult. The version inconsistency is a particularly complex problem because the abstraction level mismatch makes it difficult to get status information about the components from the current SCM system. Developers reported that they used a significant amount of time to solve such inconsistency every day.

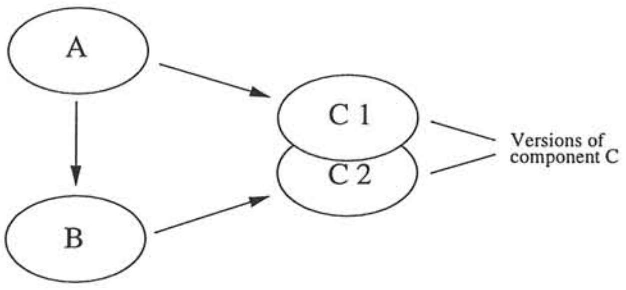


Figure 1: Version inconsistency example

3 A Model for SCM in Component-Based Environments

To help eliminate the problems addressed above, we introduce a model where the unit handled is the component. Section 3.1 describes the SCM activities that we expect will be performed in component-based environments. Section 3.2 describes a model for the repository of a corresponding SCM tool.

3.1 Activities

Based on our experience from the Nauticus project, we identify a set of SCM activities that are carried out in a component-based development environment, where the unit in SCM support still is the file. A subset of these activities can be automated in an SCM tool, thus raising the level of all the SCM activities to the component level. The activities are:

1. Identify components as units for SCM
2. Identify the interfaces and the dependencies to the components that implement them

3. Identify the dependencies from components to the files that implement them
4. Identify the compile- and run-time dependencies among components
5. Identify the development tools used to create components
6. Identify new versions of components and interfaces
7. Select components to create configurations
8. Handle change requests (e.g. requests for error correction and new functionality) and associated change actions
9. Check-in/check-out of components and configurations
10. System build (compilation and linking)

To perform the activities 1–4 and 6 in file-based SCM, the source files have to be analyzed. By automating these activities, the developer no longer has to deal with files in SCM. This automation will simplify the performance of activities 5 and 7–9, since the developers that perform them are able to operate on higher abstraction level units. System build, activity 10, is important in SCM and may benefit from this model, since all units of a system and dependencies among them are identified. A tool that automates or supports several activities of the model, is described in section 4.

3.2 Data Model

A data model for an SCM repository is expressed as a class diagram in UML notion in Figure 2. It shows the classes and references that are used to describe a system's structure, support change management and keep track of development tools.

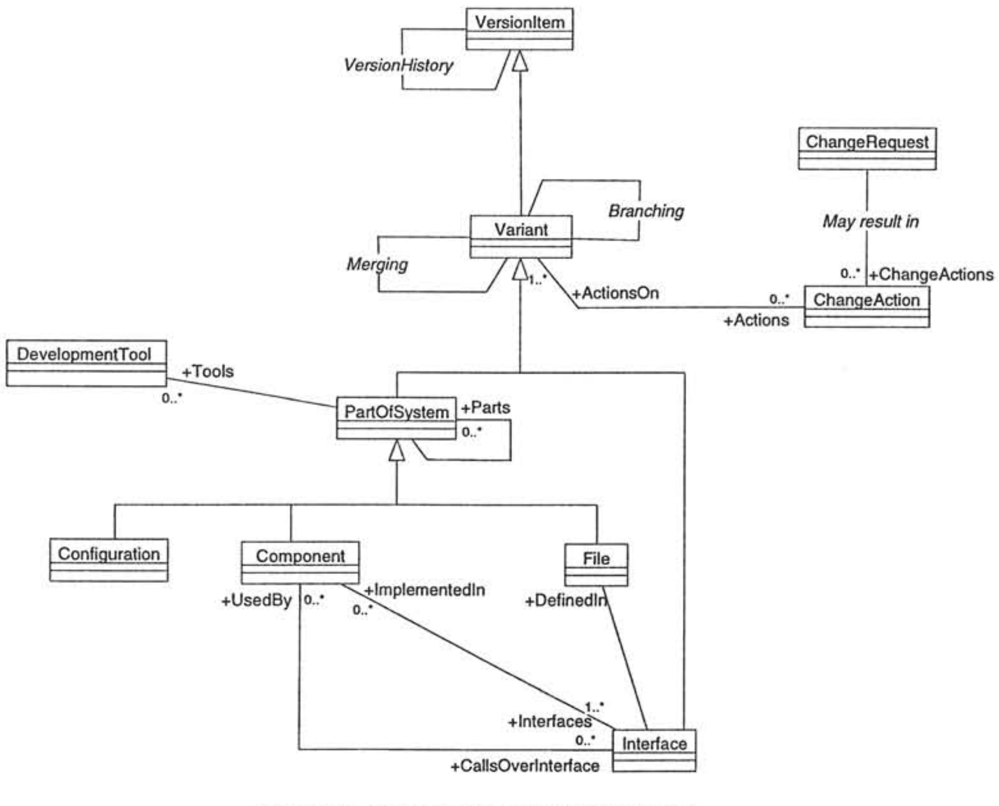


Figure 2: Class diagram in UML notation

The structuring and building units of a system are the classes Configuration, Component, File and Interface, which can form version and variant graphs (history) by inheriting the classes VersionItem and Variant.

A system's hierarchical structure is made of instances of the classes Configuration, Component and File using the reference parts inherited from the class PartOfSystem. The class Component holds the set of interfaces it implements and the set of files and components to which it has dependencies. The class Configuration is used to hold sets of structuring units that form configurations. A configuration may consist of other configurations, components and files.

A component may implement one or more interfaces (reference Interfaces) and one interface may be implemented in one or more components (reference ImplementedIn). The dependencies between components given by the call from one component to another over an interface are held by the references CallsOverInterface and UsedBy.

The class ChangeRequest holds all the incoming change requests. Change requests may result in one or more actions (class ChangeAction) that are to be carried out on one or more units in the system. The references ActionsOn and Actions hold the relation between change actions and system units.

To be able to recreate a specific version of a runnable unit in a system, it is important to have knowledge of which development tools used to develop the unit. The class DevelopmentTool holds the tool and the reference Tools holds the relation to the system unit.

4 Supporting Tool

This section describes the prototype of a supporting tool based on the model described above. The tool consists of three parts (Figure 3):

- Repositories
- Application logic for identification, versioning and change management
- User interface

The tool's repository is implemented with the class diagram in Figure 2 as schema. Instances of the class File hold references to the appropriate version of source files, which are stored in the project's versioned file repository.

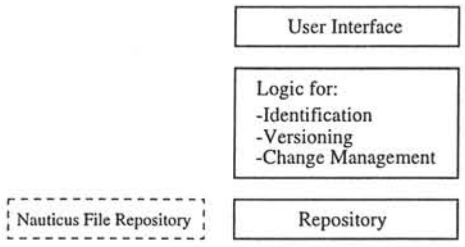
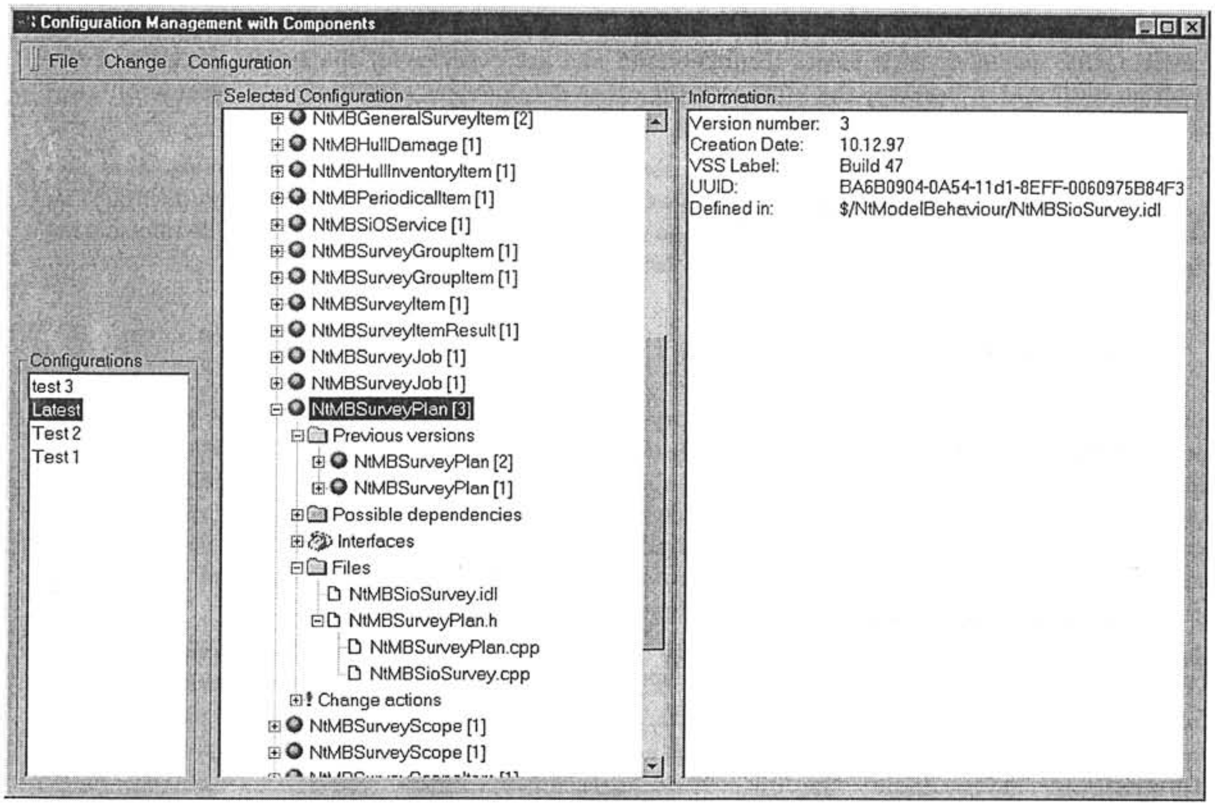


Figure 3: Implementation of supporting tool

The IDL-files are parsed to identify components, interfaces and dependencies from components to implemented interfaces (activity 1 and 2 from 3.1). The components' dependencies to source-files are found by parsing C/C++ files for *#include*-statements, starting with the header-file with the same name as the component (activity 3). Dependencies between components are identified by parsing a component's C/C++ files for the use of interfaces implemented by other components. All components that implement the identified interfaces have a dependency to the parsed component (activity 4). Identification of the development tools used to create a system unit is not implemented (activity 5). All identified units and their dependencies are stored in the repository. Only COM-components implemented in C/C++ are identified by the tool.



Version number:	3
Creation Date:	10.12.97
VSS Label:	Build 47
UUID:	BA6B0904-0A54-11d1-8EFF-0060975B84F3
Defined in:	$/NtModelBehaviour/NtMBSioSurvey.idl

Figure 4: The SCM window of the prototype

When a system is updated, for example after a build, the identification process is repeated. The result is compared with the contents of the repository to identify new versions. The repository is updated to include the new versions (activity 6).

The *graphical user interface* (GUI) presents configurations, components, interfaces and files (Figure 4). The configuration selected in the configuration list-box is displayed as a tree-view of components. By browsing a component, information about the components history, interfaces, dependencies to other components, *#include*-graph and change actions can be viewed.

The GUI offers menu selections to initiate and update the repository. Configurations are created by entering the configuration's name and selecting the appropriate versions of components in the tree-view (activity 7).

The change requests and change actions are viewed, created and edited in a separate GUI for change management (activity 8). By manually navigating through the tree-view, one can detect the components that depend on a component being the subject of a change.

Check-in/check-out routines (activity 9), tools for system building (activity 10) and functionality for branching and merging are not implemented in the tool.

The tool is implemented in Java with Microsoft J++ 1.1 and Microsoft SDK 2.0, with Microsoft Visual Source-Safe as repository for the source files.

5 Evaluation

To identify problems of SCM in the Nauticus project, we interviewed developers and SCM coordinators, and observed the SCM process and a series of meetings in a quality improvement team. The file repository used in the Nauticus project was analyzed by the tool over a two week period. Accumulated data was used in demonstrations and discussions with the development team.

The given feedback was that the component developers very much appreciate the updated component specific information provided by the tool, information which is not accessible or difficult to access (person dependent) with the file-based SCM tool used in Nauticus. The developers emphasised that this information would make it easier to understand and get overview of the system because of the abstraction level match between the SCM and the other development activities, and because of fewer units to deal with than in file-based SCM.

Developers in the project consider information about interfaces, such as which components that call over an interface and how the interfaces evolve, as important as information about components. The developers would like to have interface information presented in an own view in a tool. The information needed to construct this view is already present in the repository.

6 Related Work

Our tool uses MS Visual SourceSafe as repository because the Nauticus project used it to store files. More comprehensive SCM systems, such as ClearCase [9] could be used with our model. Our tool could exploit ClearCase's features, such as post-check-in triggers to start identification of components and interfaces, hyper-links and configuration-records to hold dependencies, and ClearMake to identify dependencies among files. However, the functionality of ClearCase alone does not fully cover the need of an SCM tool based on our model. A separate tool, like ours, is necessary to identify and store information about components, interfaces and dependencies among them.

Jordan and Van De Vanter [7, 8] describe the use of an object oriented database for persistent storage and configuration management of system units. The database is used to replace files with the storage of units such as packages (e.g. Java Packages), classes and methods. Their work focuses on the storage of units, while our work focuses on the relations among units and presentation of component specific information.

Lin and Reiss [10] describe a model for SCM in terms of fine-grained modules, such as classes and functions in C/C++. The POEM prototype based on their model records and presents modules and dependencies among modules, similar to our work. The main difference to our work is that we focus on the industrial standard notion of a component, which is a more coarse-grained module.

Our tool identifies dependencies within and among components. Dependency graphs are also required in build management. Work on identifying dependency graphs of components at a higher level than files has been carried out in the context of build management. Sjøberg et al. [12] describe a tool for incremental linking and building where components may be any typed collection of code, typically modelled as modules in Pascal and Modula-2, and packages in Java.

Fosså and Sloman [5] describe a work on interactive configuration management on software components in distributed applications. Although the component abstraction level is the same as ours, their work focuses on interactive run-time reconfiguration of distributed applications.

7 Conclusions and Future Work

We have carried out a case-study in an industrial component-based development environment where we have identified a mismatch between the abstraction level of files in SCM and the abstraction level of components in other development activities. To raise the abstraction level in SCM from files to components, we defined a model for SCM in component-based development, which introduces components and their interfaces as the units in focus.

A prototype tool was developed, that extends a file-based SCM tool with a layer to support such "higher level" SCM. In order to carry out the SCM activities, the supporting tool automatically retrieves and maintains information about components, interfaces, files, dependencies and changes. This information is an important part of the documentation of large component-based systems and may be exploited in the communication and collaboration among the software engineers.

There are plans to implement the remainder of the model (check-in/check-out routines, management of variants, support for other implementation languages and interface view) and to conduct a follow-up study on the usage of the prototype tool or its successor.

Third-party binary, reusable components are used in many development projects. Such components ought to be presented and managed equally to components with source code accessible by the SCM tool. Future work also includes expanding the model to include build management and configuration management of derived binary files.

We believe that our identification of the need for SCM of components and our initial model are useful input to SCM vendors. Building an extra layer on top of existing tools may also be a viable approach for commercial SCM products.

Acknowledgements

The generosity of Det Norske Veritas, in permitting access to the Nauticus project and the use of project resources, is very much appreciated. The authors would like to thank Kristian Rosenvold of Computas AS, Jon Skandsen of Genera AS and Erik Arisholm for the advise as the work proceeded and for useful comments on earlier versions of this paper. The authors would also like to thank Ray Welland and Frantisek Plášil for useful comments on earlier versions of this paper. Some of this work has been supported by the SPIQ-project, funded by the Research Council of Norway, grant 118206/221.

References

[1] A.W. Brown, editor. *Component-Based Software Engineering.* Selected papers from the Software Engineering Institute. IEEE Computer Society Press, 1996.

[2] R. Conradi, editor. *Software Configuration Management.* Number 1235 in Lecture Notes in Computer Science. Springer-Verlag, 1997. ICSE'97 SCM-97 Workshop.

[3] Det Norske Veritas web server. URL: http://www.dnv.com.

[4] P.H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, ESD-9-TR-7, Software Engineering Institute, Carnegie Mellon University, March 1991.

[5] H. Fosså and M. Sloman. Interactive configuration management for distributed object systems. In *Enterprise Distributed Object Computing Workshop (EDOC'97).* Goald Coast, Australia, October 1997.

[6] IEEE software, special issue on component-based software. To appear September/October 1998.

[7] M. Jordan and M. Van De Vanter. Software configuration management in an object oriented database. In *USENIX Conferance on Object-Oriented Technologies (COOTS)*, Monterey, CA, June 1995.

[8] M. Jordan and M. Van De Vanter. Modular system building with java packages. In *8th Conferance on Software Engineering Environments*, Cottbuss, Germany, April 1997.

[9] D.B. Leblang. *The CM Challenge: Configuration Management that Works*, pages 1–37. John Wiley & Sons, 1994.

[10] Y. Lin and S.P. Reiss. Configuration management in terms of modules. In J. Estublier, editor, *Software Configuration Management*, number 1005 in Lecture Notes in Computer Science, pages 101–117. Springer-Verlag, 1995.

[11] Microsoft Visual SourceSafe HomePage. URL: http://www.microsoft.com/ssafe.

[12] D.I.K. Sjøberg, R. Welland, M.P. Atkinson, P. Philbrow, and C. Waite. Exploiting persistence in build management. *Software—Practice and Experience*, 27(4):447–480, April 1997.

[13] I. Sommerville and G. Dean. PCL: A configuration language for modelling evolving system architectures. Research Report SE/8/1994, Lancaster University, Computing Science Dep., 1994.

[14] W.F. Tichy, editor. *Configuration Management.* Trends in Software 2. John Wiley & Sons, 1994.

ISO Information Resource Dictionary System (IRDS) standards and support for Component Based Information Systems Engineering.

Bernadette M Byrne, Wolverhampton University, U.K. email b.byrne@wlv.ac.uk
Paul Golder, Aston University, U.K. email p.a.golder@aston.ac.uk

Abstract

The ISO IRDS family of standards are standards for repositories. Component based Information Software Engineering (ISE) needs to be able to store and locate components when necessary for reuse. The storing and cataloguing of components needs to be done in a repository based environment. Components without a repository are analogous to a library without a filing or cataloguing system for books. The lender needs to know how to locate books, they need to be placed in a shelf position on return so that other lenders can borrow (reuse) the books. This paper describes the required functionality of a repository as outlined in an international standard and explores current issues in repository research: extensibility, data interchange standards and the interoperability of heterogeneous tools. We conclude by describing our own work in this area.

Introduction

IRDS stands for Information Resource Dictionary System. Work has been ongoing for several years on the ISO IRDS standards and there is continuing support for them. They are in fact repository standards, although the 'R' in IRDS stands for resource and not repository. Recent work has stated that the development of repository standards is a critical step towards a new industry of data integration CASE tools [3] and that repositories would benefit from more research by the database community. [1]

The link between repositories, CASE tools and component based tools.

From the mid 1980s, coinciding with the move towards the standardisation of data dictionary systems, there was a rise in the availability of, and take up of, Computer Aided Software/System Engineering (CASE) tools. Large integrated CASE (I-CASE) tools were often referred to as tools which covered all, or a major part of, the software development life cycle and were usually the product of one vendor (for example IEF from Texas Instruments and Oracle*CASE form Oracle Corporation). Component CASE tools were usually referred as a collection of tools from different vendors and these tools would help in particular stages of the SDLC. The I-CASE tools had their own central repositories, for example, the Encyclopaedia in IEF and the Oracle*CASE Dictionary in Oracle*CASE. If component CASE tools need to work in an integrated environment it will be necessary to access a common repository of information in that environment. The situation is much the same for component based ISE. Components, whether developed by CASE tools or not, cannot operate on their own but will need to link into a central repository in order to exchange information between other components, to have their meta data stored in the repository for reuse, to store information on different versions of components, to control unique naming of objects and to be able to answer impact analysis questions.

The ISO IRDS standard outlines the functionality required for a repository. The definition of an IRDS in the framework document is: "An IRD is a shareable repository for a definition of the information resources relevant to all or part of an enterprise" [4]

The ISO IRDS Standard

The ISO IRDS standard document describes a repository as having all the basic facilities of a DBMS as well as other facilities necessary for the Information Resource Management requirements of a repository. Figure 1 lists those facilities described in the standard as general database management facilities and figure 2 lists the extra facilities specific to information resource management. The ISO framework standard states that *"A large part of the task of information resource management is concerned with controlling the development, introduction and use of information processing systems".* [4] The facilities outlined for information resource management in the standard are intended to help with this task.

GENERAL DATABASE MANAGEMENT FACILITIES

Enforcement of Constraints	It should be possible to specify constraints relating to domain integrity, cardinality constraints.
Access Control	Access should be limited to suitably authorised users.
Audit Trail	It should be possible to record to an IRD or an IRD definition.
Limits and Defaults	It should be possible to specify limits and defaults for specified attributes.
Database Integrity	Integrity of the data should be supported in a single or multi-user environment (if a multi-user environment is supported).
Query and Reporting Facilities	Should be able to query the IRD or IRD definition.
Remote Data Access	Users do not necessarily have to be at the same real system as the IRDs.

Figure 1

FACILITIES SPECIFIC TO INFORMATION RESOURCE MANAGEMENT

Naming	A facility for the unique naming of objects
Status of Dictionary Content	The user should be able to distinguish between data which is no longer used (archived), data which is unstable and data which is stable.
Information System Life Cycle Management	The IRDS should be able to support the idea of phases in a Life Cycle. This is obviously useful for controlling the development and use of information systems.
Version Control	Users should be able to create and maintain versions of objects, or groups of objects and be able to use different versions of an object if required.
Partitioning	For convenience the IRD may be divided into partitions.
Copy Creation	Users should be able to make copies of an object or group of objects.
Impact Analysis	An IRDs should be able to answer queries concerned with impact analysis

Figure 2

Content of the Framework Standard

The framework standard is intended to be a general and broad document:
"The Framework identifies in general terms, the kinds of data together with the major processors and their associated interfaces and the broad nature of the services provided at each interface." [4]

THE IRDS functionality is described as four data levels as shown in figure three. The four levels are called:

a) IRD Definition Schema Level
b) IRD Definition Level
c) IRD Level
d) Application Level

The IRD Definition Schema Level

The IRD Definition Schema Level prescribes the types of objects used to define dictionaries. It prescribes the types of the objects which have instances or occurrences on the level below.

The IRD Definition Level

Instances on the IRD Definition Level have been defined on the IRD Definition Schema Level and likewise the IRD Definition Level prescribes the types of the objects which have instances on the level below. There may be several IRD definitions at this level. For example there may be IRD definitions under development and also archived IRD definitions. (The wording in the standard when describing this level may become a little confusing as there are several terms i.e. IRD Definition Schema Level, IRD Definition Level, IRD Definitions, IRD Definition Schema and IRD Schema which all have different meanings!)

IRD Level

Again instances on the IRD level have been defined as types on the level above and some of the content of the IRD level will define types at the application level. This is the 'data dictionary' level and is in fact data about data at the application level.

Application Level

Real world data is recorded at this level. Data with instances at this level will have its type recorded in an IRD on the level above.

As can be seen the levels use the concept of types and instances, for example, the IRD Definition Schema Level prescribes the types of object about which data may be recorded on the level underneath - the IRD Definition level. The IRD Definition level in turn prescribes the types of object which may be stored in the level underneath. The 'level pairs' referred to in the standard are the pairs that make up this type and instance pair.

Levels and Level Pairs

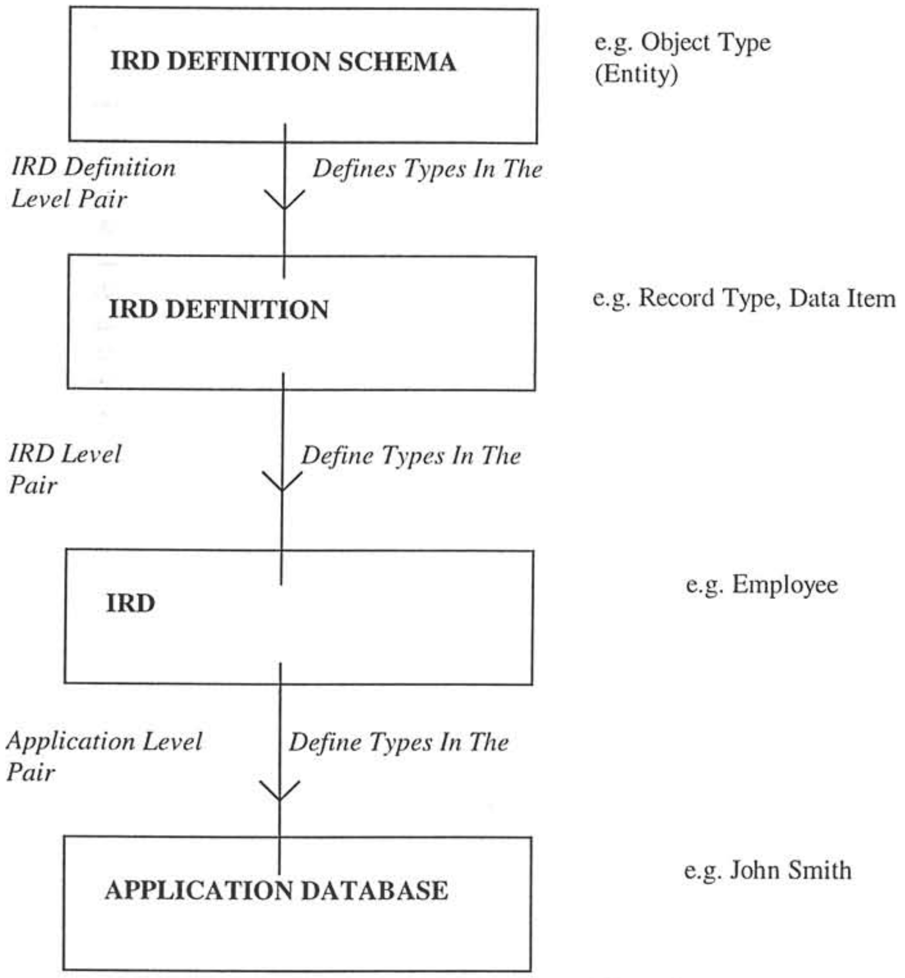


Figure 3

It is only the first two level pairs which are the subject of the standard and there is no functionality prescribed for the application level pair. Although the standard points out that there will be other International Standards (for example database languages or Programming languages) which do provide functionality which would normally be associated with the application level pair.

Family of Standards

The other standard in the IRDS family of standards is the Services Interface standard (ISO/IEC IRDS Services Interface).[5] This became a full standard in 1993. The purpose of the Services Interface Standard is to describe the services on the IRD Level and the IRD Definition Level.

The framework standard defines the context within which the Services Interface standard is defined. The services interface describes the facilities provided at the two top level pairs - The IRD definition level and the IRD level. The facilities are supported by data which is recorded in 27 IRD Definition tables. The tables are described formally using standard SQL.

Objectives of the Research

The objectives of our research are to examine the Information Resource Management facilities of a repository to identify how they perform in practice and to identify any technical and non-technical problems. It is intended to use as a starting point each of the facilities specific to Information Resource Management mentioned in the standard and to use these as a framework to explain in more detail what is intended in the standard description and to comment on how useful these have proved in a working environment, and, any problems encountered.

When examining the IRDS facility of 'Information System Life Cycle management' the research will also examine the ways and means of integrating heterogeneous systems with the main repository - this is something which will be necessary with component based development. There are two ways in which this component integration can take place; the repository can be extended in order to map on data from other components or, a standard data interchange format can be used to exchange information. One of the differences stated between a repository and a data dictionary system is that repositories should be able to be extended so that other tools can be mapped directly on to them. Currently work is being done on IRDS content modules for export/import. There are some data interchange format standards extant which could be used to exchange data between tools. However, some feel that this should be an interim measure [2] and that data should be integrated through the repository. The research will also examine the availability and use of an extensibility feature in current products and the feasibility of exchanging data using a standard data interchange format.

Therefore, the main research question in our own work is:

What are the technological issues related to the use of repository based tools and what are the perceived benefits of using these tools? The ISO IRDS standard is being used as a theoretical framework to examine the performance and operation of repositories within the existing technology in the industry. Case studies are being undertaken in organisations currently using repository based tools. Questions are being asked on the functionality of repositories as well as the wider issues of standard interchange formats between tools and repository extensibility. Qualitative research methods are being used. The data analysis part of the research is not yet complete and therefore it is not yet possible to discuss any results. However the research keeps to the forefront that the aim of software development and use is to produce quality software, on time and within budget and considers how the use of a shareable, extendible repository with standard component interchange formats can contribute towards this.

Further Issues in repository research

Extensibility

Users should be able to extend the repository and customise it to their own needs. With ISO IRDS this would involve extending the schema at the IRD Definition Level.

Case data interchange format standards

The interoperability of heterogeneous case tools. Common interchange standards would enable CASE tools to exchange data using a common format. However this could in effect cut out going through the repository and if CASE tools are going to work together in an integrated environment they need to communicate through a common repository data model.

Issues not addressed in the current ISO standards

Both the IRDS Framework Standard and the Services Interface Standard are firmly based on Relational Technology. Even though the standard claims that it makes no assumptions about an implementation environment, and assumes no specific run-time or compile-time interfaces, it is obvious from the way the structure and services are outlined that it is based on the Relational Model. The standard also states that a database services processor could be used to run the repository, although this is not a requirement. Therefore the data structures do suffer from the limitations of the Relational Model. Work is currently being carried out on Object-Oriented extensions to the model described in the Services Interface Standard, but as yet a revised standard is not available. Work is being done on a CORBA IDL binding for the IRDS Services Interface (ISO/IEC 10728).

The IRDS repository as described in the standard is a passive/storage repository not a runtime repository. Therefore it will suffer from the same problems as passive data dictionaries and will have to have strict procedures enforced by the organisation to ensure that the repository and the application data structures are kept in line. A run-time repository will have a huge performance overhead and the implementation of a distributed run-time repository will be a non-trivial task.

Although our work uses the IS0 standard as a theoretical framework it is hoped that the results of the case studies will provide general as well as specific contributions on the use of repository technology and its support for CASE tools and component based Information Systems Engineering.

REFERENCES

[1] Bernstein, Dayal, (1994) "An overview of Repository Technology" *Proceedings of the 20th VLDB Conference* Chile

[2] Rosen, B.K., Fontaines, I. (1991) "Guide to Schema and Schema Extensibility" National Institute of Standards and Technology (NIST) Special Publication 500-197

[3] Rosenthal, Seligman, (1994) "Data Integration in the large: The Challenge of Reuse" *Proceedings of the 20th VLDB Conference* Chile

STANDARDS DOCUMENTS

[4] [ISO/IEC10027] Information Resource Dictionary System (IRDS) framework. (1990)

[5] [ISO/IEC 10728] Information Resource Dictionary System (IRDS) Services Interface. (1993)