Volume 25 Number 4 November 2001 ISSN 0350-5596

Informatica

An International Journal of Computing and Informatics

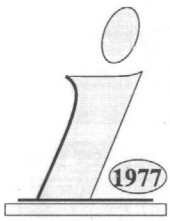
Special Issue:

Component Based Software Development

Guest Editors:

M.B. Juric, I. Rozman, D. Deugo

Informatica **25** (2001) Number 4, pp. 441–595



The Slovene Society Informatika, Ljubljana, Slovenia

Informatica

An International Journal of Computing and Informatics

Archive of abstracts may be accessed at USA: http://, Europe: http://ai.ijs.si/informatica, Asia: http://www.comp.nus.edu.sg/ liuh/Informatica/index.html.

Subscription Information Informatica (ISSN 0350-5596) is published four times a year in Spring, Summer, Autumn, and Winter (4 issues per year) by the Slovene Society Informatika, Vožarski pot 12, 1000 Ljubljana, Slovenia.
The subscription rate for 2001 (Volume 25) is
– USD 80 for institutions,
– USD 40 for individuals, and
– USD 20 for students
Claims for missing issues will be honored free of charge within six months after the publication date of the issue.

LaTeX Tech. Support: Borut Žnidar, Kranj, Slovenia.
Lectorship: Fergus F. Smith, AMIDAS d.o.o., Cankarjevo nabrežje 11, Ljubljana, Slovenia.
Printed by Biro M, d.o.o., Žibertova 1, 1000 Ljubljana, Slovenia.

Orders for subscription may be placed by telephone or fax using any major credit card. Please call Mr. R. Murn, Jožef Stefan Institute: Tel (+386) 1 4773 900, Fax (+386) 1 219 385, or send checks or VISA card number or use the bank account number 900–27620–5159/4 Nova Ljubljanska Banka d.d. Slovenia (LB 50101-678-51841 for domestic subscribers only).

Informatica is published in cooperation with the following societies (and contact persons):
Robotics Society of Slovenia (Jadran Lenarčič)
Slovene Society for Pattern Recognition (Franjo Pernuš)
Slovenian Artificial Intelligence Society; Cognitive Science Society (Matjaž Gams)
Slovenian Society of Mathematicians, Physicists and Astronomers (Bojan Mohar)
Automatic Control Society of Slovenia (Borut Zupančič)
Slovenian Association of Technical and Natural Sciences / Engineering Academy of Slovenia (Igor Grabec)

Informatica is surveyed by: AI and Robotic Abstracts, AI References, ACM Computing Surveys, ACM Digital Library, Applied Science & Techn. Index, COMPENDEX*PLUS, Computer ASAP, Computer Literature Index, Cur. Cont. & Comp. & Math. Sear., Current Mathematical Publications, Cybernetica Newsletter, DBLP Computer Science Bibliography, Engineering Index, INSPEC, Linguistics and Language Behaviour Abstracts, Mathematical Reviews, MathSci, Sociological Abstracts, Uncover, Zentralblatt für Mathematik

The issuing of the Informatica journal is financially supported by the Ministry for Science and Technology, Slovenska 50, 1000 Ljubljana, Slovenia.

Post tax payed at post 1102 Ljubljana. Slovenia taxe Percue.

Special Issue of the *Informatica* - An International Journal of Computing and Informatics

Dedicated to

Component based software development

Component based software development (CBSD) has become the predominant way of developing, packing, deploying and using software. CBSD influences all aspects of software development, which was reflected in a large number of submitted articles. Initially we received 77 contributions, which made the review process and the final selection very difficult.

For the special issue of the Journal Informatica, dedicated to Component Based Software Development we have selected fourteen quality papers, which cover different aspects of CBSD, including integration, modeling and design, patterns, agents, security, formal specifications, fault-tolerance, discussion of management processes for CBSD and a case study.

The first paper, "A Language and Framework for Supporting an Active Approach to Component-Based Software Integration" by Suzanne W. Dietrich, Susan D. Urban, Amy Sundermier, Yinghui Na, Ying Jin, and Sunitha Kambhampati, presents the IRules Component Definition Language and the environment, which acts as a mediator in the integration process by encapsulating the logic describing the interconnections between components using integration rules. It also presents the wrapper framework required for supporting the IRules active approach to component-based software integration.

The second paper, "DEPA (Design Pattern Application) – A Component-based Model for Applying Design Patterns in Software Development", by Katrina Ji and Sean Chen, discusses the lack of a formal model in applying design patterns. The authors present the DEPA model that allows a systematic way of applying design patterns in software development projects, particularly to those projects with resource constraints.

The third paper, "An Approach for Modeling Components with Customization for Distributed Software", by X. Xie and S. M. Shatz, discusses an approach for blending Petri net concepts and object-oriented features to develop a specification approach for distributed component software systems. A key result is the definition of a "plug-in" structure that can be used to create "subclass" object models, which correspond to customized components.

The fourth paper, "A Uniform Component Modeling Space" by Duane Hybertson, presents a component modeling space as a context for supporting component-based software development and accumulating component-related knowledge. It provides a uniform structure for modeling components and modeling systems in which the components may be integrated. The uniform structure can serve as the basis for an organized repository of knowledge of components and systems in which they can be used.

The fifth paper, "An Agent-Based Component Platform for Dynamically Adaptable Distributed Environments" by Rainer Weinreich and Reinhold Plösch, argues that increased flexibility can be achieved by using agent technology and agent platforms as powerful component environments. The authors present an adaptable component platform which incorporates mobile agent platforms and describe how important issues of component deployment, configuration and security are supported by the environment.

The sixth paper, "MobiDoc: A Mobile Agent-based Framework for Compound Documents" by Ichiro Satoh, presents a mobile-agent-based framework for building mobile compound documents, called MobiDoc, where the compound document can be dynamically composed of mobile agent-based components and can migrate itself over a network as a whole, with all its embedded agents.

The seventh paper, "BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems", by Sabine Moisan, Annie Ressouche, and Jean-Paul Rigault, answers the software engineering needs of the design of knowledge-based system engines in that it presents a framework composed of reusable and adaptable software components.

The eighth paper, "A Security Assurance Framework for Component Based Software Development", by Ashwin Kumar M. V. N., Arun K. Singh, and Ramesh Babu S., presents a framework to assure security of components. The framework uses Aspect Oriented Programming paradigm to capture security characteristics of the components and weaves the corresponding security checks into them. It also introduces a novel verification mechanism to ensure that the COTS components are developed as per security contract.

The ninth paper, "The ABCs of Specification: AsmL, Behavior, and Components" by Mike Barnett and Wolfram Schulte, shows how to use AsmL, an executable specification language, to provide behavioral interfaces for components. This allows clients to fully understand the meaning of an implementation without access to the source code.

The tenth paper, "Towards Rigorous and Effective Functional Contract for Components" by F. J. Galan Morillo, V. Diaz and J. M. Canete Valdeon, proposes a

form of type specification based on constructive terms. The form of specification is essential to consider abstract data types as rigorous and effective contracts between specifiers and programmers. The paper tries to establish a method, which is not only well founded but also effective.

The eleventh paper, "Approach to Component Based Synthesis of Fault Tolerant Software" by Behrooz Parhami, present a methodology which unifies previously proposed hybrid N-version programming and acceptance testing schemes, which are established methods for obtaining highly reliable results from imperfect software. The author presents a more general view which leads to higher reliability and/or greater cost-effectiveness compared to the previously envisaged hybrid schemes.

The twelfth paper, "Evolution of Fault-Prone Components in Legacy Systems: A Case Study", by Magnus C. Ohlsson, presents a model for classification of software components according to the number of times they required corrective maintenance over successive releases. The case study includes five system releases and 80 software components. Overall, the model was successful in identifying the most problematic components and provided information about the evolution of the system.

The thirteenth paper, "The Need for Speed: A Practitioner's View of Rapid Application Development in eBusiness", by Patricia Carando, reflects a practitioner's view on the present state of component-based software development in eBusiness. The focus is on component types which significantly increase the speed of development and on those types, while promising, did not realize their potential.

The fourteenth paper, "Management Process for Supporting the Component Development", by Haeng-Kon Kim, and Roger Y. Lee, discusses guidelines for supporting the development of the CBSD process. The authors focus on setting standards for components and address the impact that CBSD has on managing component development.

We would like to thank once again to all the authors who submitted papers for this special issue. Special thanks go to the reviewers for their excellent, but hard work. We hope that those who read the special issue will enjoy our selection.

Guest Editors:

Matjaz B. Juric
Ivan Rozman
Dwight Deugo

Reviewers (in alphabetical order) for the special issue of the Informatica, dedicated to Component Based Software Development:

Ivan J. Araujo, Michel Barbeau, Simon Beloglavec, Sondes Bennasri, Bostjan Brumen, Troy Bull, Particia Carando, Robert Cattral, Jean-Pierre Corriveau, Babak Esfandiari, Darrell Ferguson, David Flater, Grant Gayed, József Györkös, Marten Haglind, Abdelwahab Hamou-Lhadj, Marjan Hericko, Doug Howe, Hannu Jaakkola, Marko Juvancic, Fabio Kon, Andrej Krajnc, Ivan Lah, Timothy C. Lethbridge, James Moody, Oscar Nierstrasz, Franz Oppacher, Vojislav D. Radonjic, James Edward Ries, Colette Rolland, Kimmo Salmenjoki, P. G. Sarang, Ichiro Satoh, Carine Souveyet, Vladimir Tosic, Romana Vajde Horvat, Eugene Wallingford, Michael Weiss, Tatjana Welzer, Lee White, Ales Zivkovic

A language and framework for supporting an active approach to component-based software integration

Suzanne W. Dietrich, Susan D. Urban, Amy Sundermier, Yinghui Na, Ying Jin, Sunitha Kambhampati
Department of Computer Science & Engineering
Arizona State University, Tempe, AZ 85287-5406, U.S.A.
dietrich@asu.edu, s.urban@asu.edu

Keywords: component-based integration, active rules, events

Received: June 25, 2001

The IRules project at Arizona State University applies active rule technology to the integration of distributed, black-box software components. The goal of IRules is to provide an environment in which an application is developed through the integration of software components using active rules that are known as integration rules. Using the IRules Component Definition Language (CDL), the application integrator first describes a purchased, black-box component within the IRules environment to allow access to the properties and methods defined by the purchased component. In addition, CDL allows for the definition of named extents, stored and derived attributes, externalized relationships and events to enhance the features of the purchased components to support application development. After defining the desired interface for the component, the application integrator then develops the application using active integration rules that define the interaction of the components in response to events. This paper presents the Component Definition Language and its resulting framework that supports the IRules active approach to component-based software integration.

1 Introduction

Electronic commerce (e-commerce) and other Web-based applications are inherently distributed in nature since a client is not expected to be co-located with the data for the application. However, many Web-based applications are primarily three-tiered architectures with a presentation layer using a browser to interface with the user, a middle tier generally built with an object-oriented or component-based interfacing technology, and a database as the persistence layer. Commercial component standards such as Enterprise JavaBeans (EJB) from Sun Microsystems or COM+ from Microsoft are typically adopted for the middle tier, allowing the developers of an application to focus on creating code to represent business needs while relying upon a commercial component container to supply infrastructure services. The current state of middle-tier component software primarily addresses the requirements of three-tier architectures, simplifying the development of Web applications by providing an application-programming layer between the presentation and storage layers that decreases development time. However, e-commerce applications require access to data and software from many different sources. As a result, the simplifying assumption of one underlying database in the persistence layer as in most three-tiered architectures is too restrictive for some distributed applications.

There is a conflict between the goal of a commercial component container vendor to make three-tier web application development simpler and faster versus the longer-term goal of software engineering to make component-based application development a reality. Component-based software engineering research encourages the idea of building applications from purchased components. Using purchased components directly to build an application, however, is difficult to accomplish if the application developer must work with the limitations of an interface defined by the vendor of the component. This paper addresses some of the challenges inherent in application development using purchased components. We refer to these components as "black-box" components, since we assume the component must be used without modification to source code.

Our research focuses on adapting database technologies to the area of software component integration. As described in (Silberschatz & Zdonik 1997), certain forms of database functionality need to "break out of the box" to better serve the needs of applications that depend on distributed sources of information. Active rule processing technology (Widom & Ceri 1996) is an example of a database component that can provide useful services to advanced applications if the appropriate technology exists for the use of rules in distributed environments. Traditionally, active rules have been used to transform passive, centralized database systems into reactive systems that respond to database and external events through the use of rule processing features. Active rules are typically formatted as Event-Condition-Action (ECA) rules. When an event occurs, if an optional condition holds, then a specified action is performed.

The Integration Rules (IRules) project at Arizona State University (http://www.eas.asu.edu/~irules) is

investigating the middle-tier, rule processing technology necessary for the use of active rules in the integration of distributed, black-box software components (Urban et. al. 2001a, Urban et. al. 2001b). The intended use of this rule processing technology is for the specification of event-based processing logic in the development of component-based applications for distributed environments, where the granularity of the components can range from low-level database objects to an entire software system.

The IRules approach builds upon the use of the Enterprise JavaBeans (EJB) software component model specification from Sun Microsystems (J2EE 2001). The EJB component model promotes the vision of separating component services from the business logic of the components. Assuming that all databases and software sources of the application environment are encapsulated using EJBs, the application integrator uses the IRules Component Definition Language (CDL) to extend the definition of a purchased software component to declare named extents, additional attributes, externalized relationships and component events. Since there is no inherent support for direct object references between distributed components, *externalized relationships* (Rumbaugh 1987) play an important role in associating the purchased components that are being integrated in the distributed IRules environment.

Once components are defined in the IRules environment using CDL, application integrators can create distributed applications using the IRules Integration Rule Language together with application transactions. Integration rules provide a re-active capability to the environment so that as distributed components and external sources generate event notifications, integration rules invoke methods on components or perform higher-level application transactions. The purpose of this paper is to provide a description of the IRules Component Definition Language and the framework of component metadata and wrappers that are generated to support the IRules active rule architecture approach to the integration of purchased software components.

In the following sections, we outline the details of the IRules approach to component integration. Section 2 first provides an overview of related work. In Section 3, we provide an overview of the IRules approach, introducing an investment example that will be used to illustrate IRules concepts throughout the rest of the paper. Section 4 introduces the IRules Component Definition Language, illustrating the definition of named extents, additional attributes, externalized relationships and component-generated events. Section 5 elaborates on the static component metadata that is generated as a result of the compilation of CDL. Section 6 provides the details on how the IRules environment supports the enhancements to the purchased components using wrappers. The paper concludes in Section 7 with a summary of our work and a discussion of future research directions.

2 Related Work

Recent work on component integration has focused on the architecture of software interconnection based on the underlying component model. Software architectures such as COM+ (Microsoft 2000), CORBA (OMG 1998), and Enterprise JavaBeans facilitate the integration by supporting the development of systems from independently developed components.

One approach to the interoperability of components is event-based. In (Barrett et. al. 1996), the Event-Based Integration (EBI) framework was proposed as a high-level, general, and flexible reference model for event-based software integration. This approach outlines architectural concepts for interconnection through events. In (Ma & Bacon 1998), the CORBA-Based Event Architecture (COBEA) is a general event-driven architecture for building distributed active systems. COBEA extends the CORBA Event Service by supporting the publish-register-notify model and provides filtering, fault-tolerance, and access control services. COBEA is a general event-driven architecture for distributed active systems, rather than an implemented system.

There have been some initial results on the use of ECA rules for integrating distributed components. In (Pissinou & Vanapipat 1996) and (Pissinou et. al. 1997), an ECA rule approach is used in component interoperation. Distributed applications are modeled as Distributed Active Objects by adding wrappers on top of the components that do not have triggers so that ECA rules can be used in distributed environments. The ECA Object Service is based on the CORBA specification. Objects communicate by method invocations and service requests. The rule object is an independent CORBA object that is isolated from the application objects. The work in (Pissinou et. al. 1997) describes an architecture for executing rules in a distributed environment. In (Chakravarthy & Le 1998) ECA rules are proposed to solve distributed interoperation of components that have an OMG IDL interface. The project focuses on the specification, detection and management of composite events (Le & Chakravarthy 1998). The system uses the CORBA event service and implements conditions and actions by method calls. In (Bultzingsloewen et. al. 1996, Koschel & Lockemann 1998), the CORBA-Based Distributed Information System named C^2offein was developed to use ECA rules for distributed component interoperation. Wrappers are used for read access to the underlying data source and primitive event detection. The CORBA push model is used for event detection. C^2offein provides a concrete architecture and implementation of how to use ECA rules to integrate heterogeneous information sources.

Our own past work in the area of active database systems has influenced the research presented in this paper. In particular, our work with the ADOOD RANCH (Dietrich et. al. 1992) project resulted in a declarative language for

the integration of active, deductive, and object-oriented language concepts (Urban et. al. 1997), together with a framework for capturing the metadata of such an environment (Abdellatif et. al. 1999) and an execution model that supports the incremental examination of the database state during rule processing (Abdellatif 1999). Our approach to the use of derived attributes in the IRules Component Definition Language, as well as the structure of integration rules, extends our results from the ADOOD RANCH project to distributed domains. The work in (Ayyaswamy 1999) represents our initial investigation of a CORBA architecture for distributed ECA rule processing for the purpose of maintaining constraints in a loosely-coupled, federated database environment. More recently, we have performed a comparison of CORBA (OMG 1998), Java (J2EE 2001), and Jini (Arnold 2000) technologies for evaluating different architectural options for the execution of integration rules (Saxena 2000, Urban et. al. 2001c).

The IRules project differs from the above research projects in several aspects. First, IRules is based on the Enterprise JavaBeans component model. Second, IRules builds its own distributed environment. The compilation of the IRules Component Definition Language automatically generates the code for the wrappers of the black-box components rather than hard-coding the wrappers. Third, the IRules project is also investigating transaction management, conflict resolution, and failure handling issues in its distributed rule processing environment.

The externalized relationships of the IRules environment share some similarities with the CORBA Relationship Service (OMG 2000). The Relationship Service supports the definition and creation of relationships between distributed CORBA objects. As with IRules externalized relationships, the related objects do not have to be aware of the relationship. One obvious difference is that the Relationship Service is for relating CORBA objects while the IRules externalized relationships are designed for black-box components that adhere to the EJB component model.

3 IRules Overview

This section provides a high-level overview of the IRules project to establish the basis for a more detailed presentation of the IRules Component Definition Language and its supporting metadata and wrapper framework in the following sections. The IRules project adopted the Enterprise JavaBeans (EJB) server-side component model for the Java programming language (J2EE 2001). Due to space limitations, we assume prior knowledge of EJBs.

To illustrate the IRules approach to the integration of EJB components, this section presents an Investment application that is used throughout the remainder of this paper. This application is depicted in Figure 1, illustrating four different containers with purchased software components. The Portfolio container maintains current information in the form of entity beans about client portfolios, including information about current and past stock holdings and the orders under which stocks were bought and sold. The Portfolio container also provides a session bean with application logic to conduct buying and selling of stocks. The Pending Order container provides entity beans for storing pending orders that are waiting for execution when a particular market condition is met. The Stocks container represents locally managed information about stocks and their current prices as entity beans. This information exists independently of the portfolios that own them. We are assuming that the information in the Stock container is updated based on current stock prices from external sources. The Stock container can also generate events to signal changes in value depending upon buy/sell transactions in the stock market. Finally, the User container uses entity beans to store billing information about portfolio accounts and the users that are associated with accounts. The User container also provides a session bean with procedures for billing users for stock buy and sell transactions.

There are implied relationships between the four containers in Figure 1. Portfolios contain specific stocks. Pending orders are related to a portfolio and represent buy and sell transactions on stocks. Portfolios are owned by a specific account, and accounts are billed for buy and sell transactions. In general, the application programmer must know of the specific relationships and write procedural code to achieve the integration. In a typical Web-based, three-tier architecture, this approach may be satisfactory. Advanced distributed applications, however, may require the interconnection of components in containers provided by multiple companies from distributed locations. These types of applications can benefit from an environment that provides greater support in understanding and establishing relationships between distributed components.

In addition to illustrating the four independent containers of our sample application, Figure 1 also illustrates the the IRules approach to component interconnection. The lines between containers represent externalised relationships of the desired object model, defining relationships between components on different servers. For example, we wish to represent the fact that a Portfolio may have orders waiting for execution, where the order information is stored in the Pending Order component. The Pending Order component is also defined to act upon a specific type of stock. The application integrator uses the IRules Definition Language to define a distributed application.

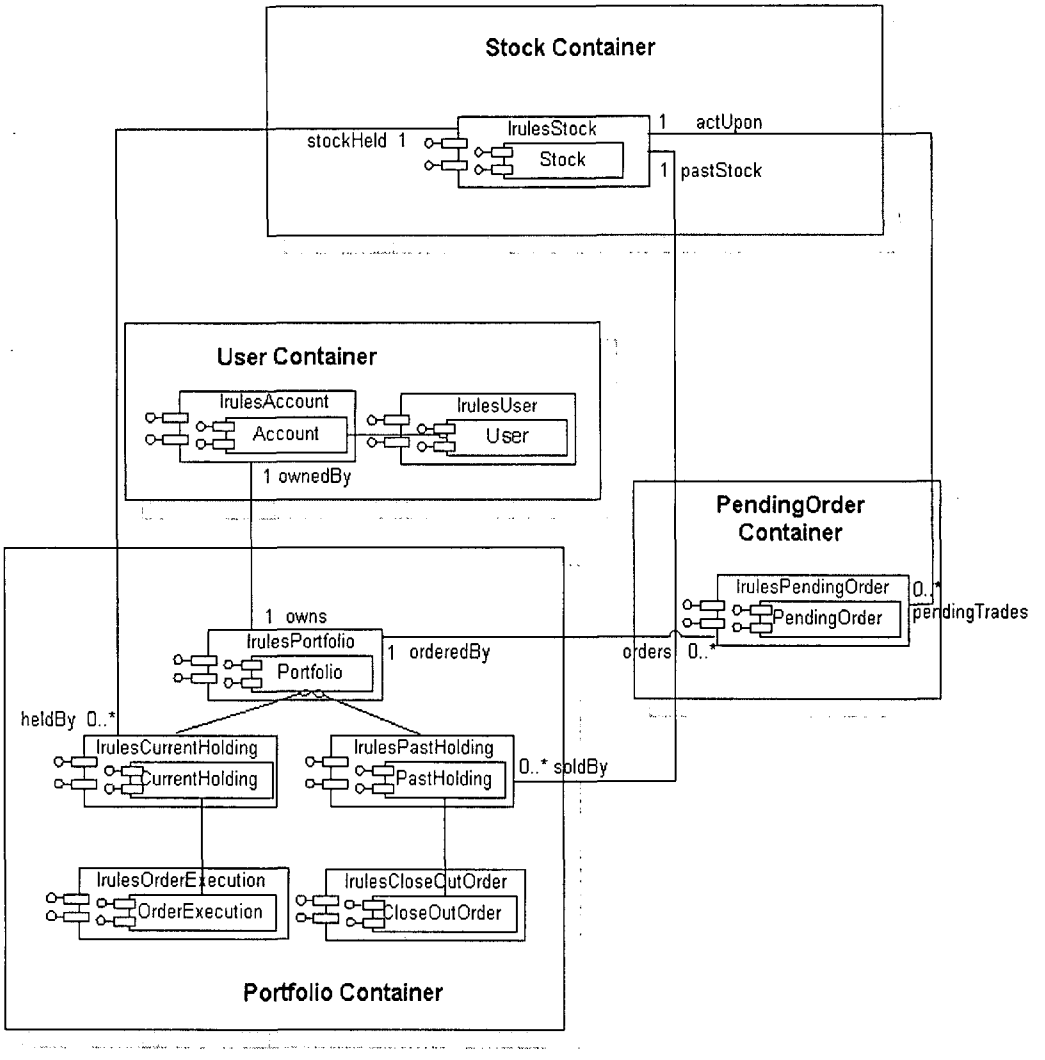


Figure 1: Investment Example

The IRules Definition Language consists of four sublanguages: (1) the Component Definition Language (CDL), (2) the Event Definition Language (EDL), (3) the Integration Rule Language (IRL), and (4) the IRules Scripting Language (ISL). Using CDL, the application integrator first describes a purchased, black-box component within the IRules environment to allow access to the properties and methods defined by the purchased component. In addition, CDL allows for the definition of a named extent, stored and derived attributes, externalized relationships and events to enhance the features of the purchased components to support application development. EDL provides a language for the definition of external and system-level events. After defining the desired interface for the components and events, the application integrator then develops the application using active integration rules via the IRL, which defines the interaction of the components in response to events. ISL provides the application integrator with a more complete approach to transaction development over the object model of the distributed application. Currently, we are investigating JACL (DeJong & Laird 1997) and its extensions as the foundation of the ISL.

Figure 2 illustrates a high-level architectural view of the IRules processing environment. In the IRules architecture, the object manager uses component metadata and the abstract IRules wrapper interface to provide the rule processor with the appropriate references to remote interfaces as needed to process rules and transactions. Thus the object manager encapsulates the choice of the EJB component model from the other system components in the IRules framework and architecture. The metadata manager stores information about the IRules object model of the application, resulting from the compilation of the IRules Definition Language. The compilation of CDL also results in the generation of wrappers for the purchased components, which provide required information for supporting the IRules environment. The object manager and the metadata manager are used by the transaction and rule

processor to execute the application logic of the environment. Transactions specified in the scripting language can execute methods on entity and session beans, where the object manager is first consulted to locate the component required for the execution of the method. The execution of such methods can send event notifications via the IRules wrappers to the event handler, denoting the before and after points in the execution of such methods. The event handler communicates with the transaction and rule processor to trigger integration rules. The execution of integration rules triggers additional application transactions, beginning a new cycle in the execution of methods on EJB components. A more complete description of the execution environment for IRules can be found in a companion paper (Urban et. al. 2002). The following sections elaborate on the CDL and the metadata and wrapper framework required to support a rule-based approach to software component integration.

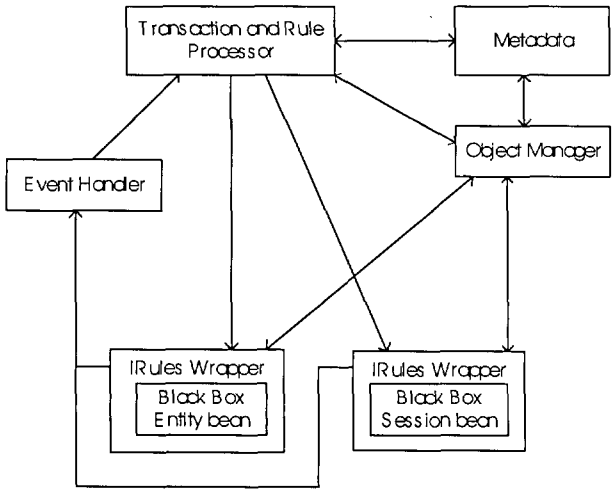


Figure 2: IRules Architectural Overview

4 Component Definition Language

The Component Definition Language provides the application integrator with a tool to describe purchased, black-box software components to the IRules environment. Recall that IRules assumes the EJB component model and components can be either entity beans or session beans. The application integrator defines only additional properties using CDL. In other words, the behavior of the black-box EJB is not redefined in CDL, since it is available using reflection. Figure 3 presents the CDL for the Investment Example presented in the previous section. Specific examples from this figure will be used to describe CDL in more detail.

The syntax of CDL is loosely based on the syntax of the Object Definition Language (ODL) of the Object Data Management Group (ODMG) standard (Cattell et. al. 2000). ODL defines the classes that an object-oriented database (OODB) manages in its persistent store. For each class, the database designer may define an extent, which is a named collection of the objects of that type in the database, and the properties and behavior associated with that class. The term *property* refers to *attributes*, describing characteristics of the object, or *relationships*, defining associations between objects. The relationship between two objects is automatically maintained by the database system. When one side of the relationship is updated, the database system is responsible for maintaining the inverse relationship.

The IRules Component Definition Language provides the application integrator with the ability to define named extents, attributes, relationships and events to be associated with a black-box component deployed in the IRules environment. The applicability of these definitions depends on the type of the EJB, such as entity beans or session beans.

4.1 Entity Beans

Each entity bean is described to the IRules environment by a component declaration, giving the name of the black-box component (*ComponentName*). Entity beans are identified by the implements EntityBean clause of the component definition. Entity beans defined within the IRules environment may include a named extent, additional attributes, externalized relationships and events. In the abstract syntax shown below, italicized identifiers represent names that are filled in based on the specification of the application:

```
component ComponentName implements EntityBean
(extent ExtentName)
{ attribute     AttributeType      AttributeName
                { OptionalAttributeDefinition };
  relationship RelationshipType  RelationshipName
                inverse InverseRelationshipName;
  event         IRulesEventName(EventParameters)
                { method EventDefinition };                }
```

The *ExtentName* provides the name of an extent that the application integrator can use in the specification of the integration rules to iterate over objects of the type *ComponentName*. In the Investment CDL of Figure 3, each entity bean has a defined extent. By convention, the name of the extent is the plural of the name of the component. For example, the component Stock has an extent named stocks.

An attribute defined in CDL specifies default behavior to get and set the attribute's value: the method *getAttributeName* returns *AttributeType*, and the method *setAttributeName* takes an argument of *AttributeType* to which the attribute value is set. This attribute will be stored as part of the IRules wrapper for the component. In Figure 3, the Portfolio component has a stored attribute, named lastPortfolioValue of type float. The IRules wrapper automatically provides the accessor (getLastPortfolioValue and setLastPortfolioValue) methods for the stored attribute.

```
component Stock implements EntityBean
 (extent stocks)
 {relationship set <CurrentHolding> heldBy
    inverse CurrentHolding::stockHeld;
  relationship set <PendingOrder> pendingTrades
    inverse PendingOrder::actUpon;
  relationship set <PastHolding> soldBy
    inverse PastHolding::pastStock;
  event beforeSetPrice(NewPrice)
   {method before setPrice(NewPrice)}; };
component User implements EntityBean
 (extent users) {};
component Account implements EntityBean
 (extent accounts)
 {relationship Portfolio owns inverse Portfolio::ownedBy;};
component Portfolio implements EntityBean
 (extent portfolios)
 {attribute float lastPortfolioValue;
  attribute float portfolioValue {
    portfolioAI.calculatePortfolioValue(Portfolio self)};
  relationship Account ownedBy inverse Account::owns;
  relationship set<PendingOrder> orders
     inverse PendingOrder::orderedBy; };
```

```
component CurrentHolding implements EntityBean
 (extent currentHoldings)
 {relationship Stock stockHeld inverse Stock::heldBy;};
component PastHolding implements EntityBean
 (extent pastHoldings)
 {relationship Stock pastStock inverse Stock::soldBy;};
component PendingOrder implements EntityBean
 (extent pendingOrders)
 {relationship Stock actUpon inverse Stock::pendingTrades;
  relationship Portfolio orderedBy inverse Portfolio::orders;
  event afterCreatePendingOrder
    (pnId,portId,stockId,numOfShares,desPrice,action)
 {method after
  create(pnId,portId,stockId,numOfShares,desPrice,action)};};
component OrderExecution implements EntityBean
 (extent orderExecutions) {};
component CloseOutOrder implements EntityBean
 (extent closeOutOrders) {};
component PortfolioSession implements SessionBean
 {event afterSellStock(stockId,price,portId,numOfShares)
 {method after sellStock(stockId,price,portId,numOfShares)};};
component PortfolioAI implements SessionBean {};
```

Figure 3: Component Definition Language for the Investment Example

An attribute may also be a derived attribute, meaning that its value is computed using a predefined method. In the Investment example shown in Figure 3, the Portfolio component has a derived attribute portfolioValue of type float. When the portfolioValue attribute is referenced (using the getPortfolioValue method), its value will be computed using the calculatePortfolioValue method defined in the portfolioAI session bean. The suffix AI in this example stands for Application Integrator, since it is the responsibility of the application integrator to define the meaning of a derived attribute. At this point in time, IRules allows this logic to be coded as a method of a session bean. We have introduced the self syntax here to indicate that the method is called on the Portfolio object itself. We are planning to allow for additional parameters to the method call, which could include properties of the purchased components and the properties defined as part of the enhanced IRules environment.

We have briefly explored the use of the Enterprise JavaBeans Query Language (EJB QL) to declaratively specify the meaning of a derived attribute. The current specification of EJB QL within the EJB 2.0 specification has several limitations that discourage its use within IRules at this time. One limitation restricts values returned from a query to be either an existing object or part of an existing object. Another limitation restricts the traversal of relationships to only those deployed in the same container (and the same ejb-jar file). Since the goal of IRules is to provide externalized relationships across distributed components in multiple containers, these limitations are too restrictive. Therefore, we have provided the application integrator with a more general option to specify the required logic as a method of a session bean.

Externalized relationships play an important role in specifying the associations between the black-box components being integrated. In the Investment example, the association ownedBy in the Portfolio component relates a portfolio to its associated account. The inverse relationship owns in the Account component associates the account to its portfolio. Section 6 describes how these externalized relationships are maintained in the IRules wrapper for the black-box component.

The application integrator ultimately specifies event-based integration rules to glue the black-box components together. When an event occurs, if an optional condition holds, then a specified action is performed. At the component level, the application integrator defines IRules events that the integration rules monitor based on method calls to the underlying black-box component. The IRules environment supports the generation of an event before or after a method call. For example, in Figure 3, the component Stock defines beforeSetPrice as an event that the IRules environment monitors, which is raised before the call to the setPrice method in the underlying black-box Stock component. The event parameter newPrice is obtained from the newPrice parameter to the setPrice method call.

We also plan to have the IRules environment support the selective monitoring of internal events from black-box components that are compliant with the Java Message Service API (JMS), which is the event service adopted

for EJBs. The details for providing this support are currently being investigated.

4.2 Session Beans

Session beans must also be declared to the IRules environment by a component declaration using the implements SessionBean clause, allowing IRules access to the properties and methods defined by the purchased component. A session bean may also define events to be monitored at the IRules level.

component *ComponentName* implements SessionBean
{ event *IRulesEventName(EventParameters)*
{ method *EventDefinition* }; }

The *EventDefinition* is consistent with events defined for entity beans. The event is specifying the interception of a before/after method call to the underlying black-box session bean. In Figure 3, the CDL component definition for the session bean PortfolioSession defines the IRules event afterSellStock, which is raised after the method call to sellStock. We are also investigating a mechanism to support the selective monitoring of an internal event of a session bean by the IRules environment.

5 Component Metadata

Metadata is the data maintained by the system that describes the data in the system itself. For example, relational databases use metadata to represent the data of any application. In a similar manner, the IRules system uses metadata to represent both the components and the processing logic (application transactions and integration rules) of the application, allowing the system components of the architecture to be data-driven by the metadata describing the application. The IRules environment will store metadata as the result of compiling the IRules Definition Language. This section describes the metadata stored by the compilation of CDL. The current prototype of the IRules component metadata is written using serialized Java objects. We plan to investigate the use of the JavaSpaces service in Jini for the distributed metadata implementation.

Figure 4 gives a UML diagram illustrating the static component metadata generated as a result of the compilation of the IRules Component Definition Language. The metadata stored for an IRules Wrapper includes its name, the name of its black-box component and its JNDI name. JNDI is the abbreviation for the Java Naming and Directory Interface, which provides location and organization services in a distributed computing environment. The wrapper includes an association to the black-box component that it wraps. The JNDI name of the black-box component is obtained from the deployment descriptor for this purchased component.

An IRules Wrapper is itself an EJB of the same type as the EJB that it is wrapping. An IRules Wrapper that is an entity bean must store the name of its associated extent and properties, which are relationships and attributes. A relationship has a cardinality, such as single-valued or multi-valued. In this case, the IRulesRelationship metadata class shows a multivaluedFlag that is set to true for a multivalued relationship. A relationship also has an inverse, which is indicated by a recursive association in Figure 4 to the IRulesRelationship class, which gives the inverse relationship. An attribute has a type, and may be explicitly stored or derived. A derived attribute, as shown by the IRulesDerivedAttribute metadata class, records the name of the method and its session bean that is called to derive its value based on input parameters. The names of the attributes associated with the black-box component are also maintained in the metadata as BlackBoxAttribute since the IRules Wrapper acts as its proxy. Similarly, the IRulesMethod class represents the methods of the black-box components and the default accessor methods for the IRulesAttributes.

The events associated with a component are either method events or internal events, and are illustrated in Figure 4 by an association to an eventStub. The eventName provides an access path into the event metadata, which also includes external and system-level events that are defined using EDL.

6 Wrappers

One of the goals of the IRules project is to integrate commercial-off-the-shelf (COTS) components using containers produced by commercial vendors. The IRules wrappers play an important role in enhancing the interface of a purchased software component, providing the additional behavior required for interacting with the IRules environment. The wrappers provide a mechanism to act as a proxy to the original black-box component and to add the definition of IRules extents, attributes, externalized relationships, and events. This section describes how the IRules environment wraps both entity beans and session beans to become part of an IRules distributed application.

Figure 5 provides an overview of the IRules approach to wrapping black-box EJB components, assuming the naming conventions for EJBs. The EJB Layer in the diagram reinforces the description of the EJB component model. The EJBHome interface represents the life-cycle methods of the component. The EJBObject interface, also known as the remote interface, defines the signature of business methods for changing attribute values and carrying out business logic functions that are specific to the EJB component. The right-most column represents the implementation of the enterprise bean. The Wrapper Abstract Layer provides the behavior that is inherited by every IRules wrapper. The Wrapper Implementation Layer shows the IRules Wrapper for the BlackBox component. The lowest layer of the diagram is the Component Layer and identifies the BlackBox EJB.

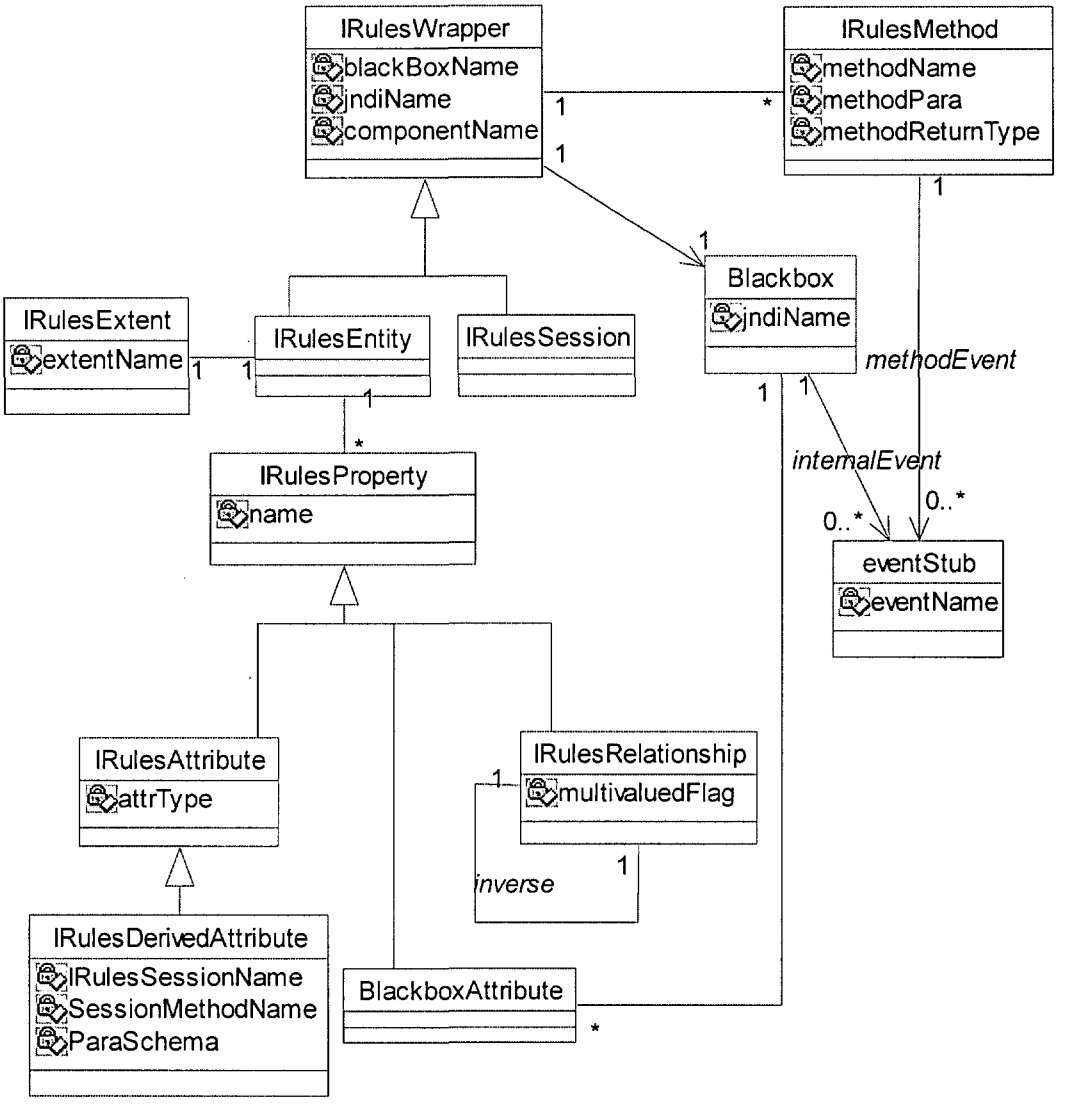


Figure 4 Static Component Metadata

To develop the functionality of the wrappers, we have implemented a prototype for the Investment application that provides proof of concept of the technology. Since our initial prototype, the IRules wrapper design has already been refined based on changes to the EJB specification. The EJB 2.0 specification introduced a new container-managed persistence (CMP) contract. For container-managed entity beans, the deployment descriptor indicates which fields and relationships of the bean are to be maintained by the container. The actual mapping of these container-managed fields (CMF) to a persistent store happens in a server-specific way and is not included in the deployment descriptor. In the case of the BEA Systems Weblogic Server (BEA 2001) that we are using for our implementation, an XML file specifies the object-to-relational mapping between the CMFs and container-managed relationships (CMRs) to the underlying relational database store. The only container-managed relationships supported are those between entity beans deployed at the same time (in the same ejb-jar file). Since this limitation on CMRs is too restrictive for the IRules environment, the IRules wrappers explicitly provide a mechanism to store and retrieve the externalized relationships from the underlying persistent storage.

6.1 Entity Beans

Since the IRules wrapper for a deployed black-box entity bean stores persistent data and needs to be shared between clients, the wrapper is also an entity bean. The IRules wrapper defines container-managed persistent fields for (1) the reference to the black-box entity bean it is wrapping, (2) stored attributes, and (3) externalized relationships. The wrapper also includes code that is generated from the compilation of CDL to provide accessor methods for attributes, manipulation methods for relationships, a proxy to method calls, and support for raising events to trigger rules.

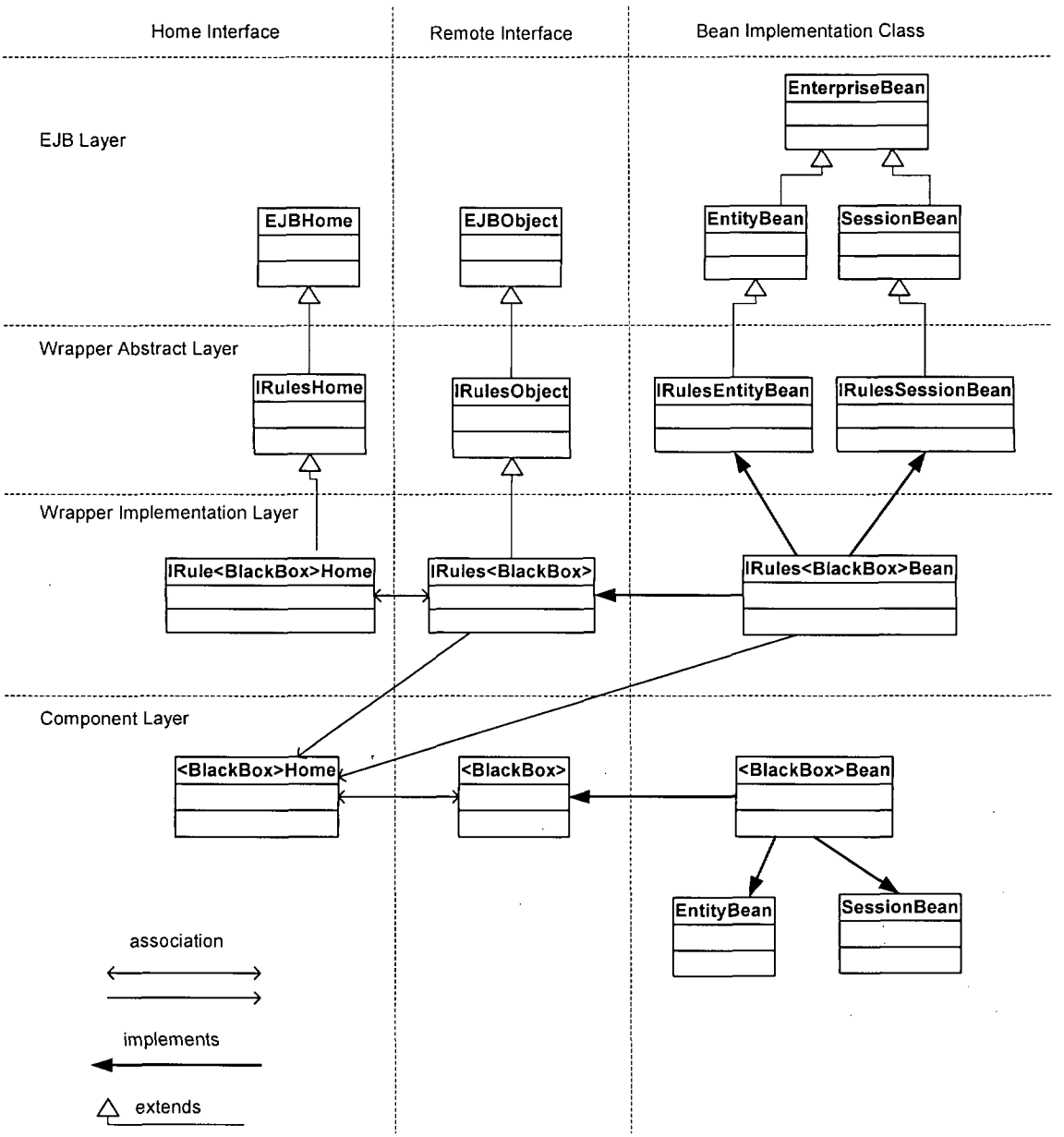


Figure 5: IRules Wrapper Overview

The code for an IRules wrapper is generated by the compilation of CDL. The instances of the wrapper are created using the create method in the Home interface of the wrapper. The parameters to the create method include the primary key of the black-box EJB that the wrapper is wrapping and any initial values for stored attributes and externalized relationships. Note that the primary key of the IRules wrapper is set to the same value (and type) as the primary key of its wrapped purchased component.

The wrapper establishes a reference to its black-box component storing its handle, which is a persistent network reference to an EJB object, as a CMF called cmpForHandle. In a container-managed entity bean, the data type for a CMF can be a Java primitive type or a Java serializable type. The container is responsible for transferring data between an entity instance and the underlying persistent storage, and provides a get and set method for each CMF. To store the EJB handle to the black-box component, the EJB handle is cast to the type java.lang.Object and used as a parameter to the setCmpForHandle method, which is supplied by the container. The following code snippet illustrates how the handle to a PendingOrder entity bean, given by the variable p, is stored in the IRules wrapper:

```
Handle handleToBB = p.getHandle();
Object objHandleToBB = (Object)handleToBB;
setCmpForHandle(objHandleToBB);
```

To retrieve the handle as the appropriate type, the wrapper implements a refToBlackBox method that calls the

getCmpForHandle method to retrieve the serialized handle and casts it to the type of the associated purchased component, in this case PendingOrder:

```
public PendingOrder refToBlackBox()
    {Object obj = getCmpForHandle();
    Handle handle = (Handle)obj;
    EJBObject ejbobj = handle.getEJBObject();
    PendingOrder blackBox = (PendingOrder)ejbobj;
    return blackBox;}
```

There is nothing added to the wrapper itself to support named extents. When the CDL is compiled, the extent name is entered into the static component metadata. The extent name is only used by the application integrator in the IRL application specification to iterate over the extent of a component. The underlying implementation of the IRL will use the findAll method provided in the home interface to realize the extent.

CDL allows the definition of both stored and derived attributes. Since a stored attribute persists as a CMF of the IRules wrapper, the container is responsible for the get and set methods. A derived attribute is not stored but virtual. Its value is derived using the get method in the wrapper that is generated as the result of compiling CDL. The get method calls the method of the session bean that derives the value of the attribute. Consider as an example the getPortfolioValue method in the wrapper for Portfolio that calls the calculatePortfolioValue method of the portfolioAI session bean. The getPortfolioValue method first gets the home interface of the wrapped portfolioAI session bean. This lookup functionality is abstracted in the method lookupAIHome() that uses JNDI to locate the home interface. The call to calculatePortfolioValue calculates the portfolio value with the corresponding black-box portfolio EJB as an input parameter.

```
public float getPortfolioValue()
    {IRulesPortfolioAIHome home = lookupAIHome();
    IRulesPortfolioAI ai = home.create();
    Portfolio self = refToBlackBox();
    float value = ai.calculatePortfolioValue(self);
    return value;}
```

Externalized relationships are also implemented as a CMF in the IRules wrapper, since the current restriction of CMRs in the EJB 2.0 specification limits relationships to entity beans deployed in the same ejb-jar file. Relationships are associations that can be single-valued or multivalued. A single-valued relationship is stored in a manner similar to the reference to the black-box component, storing the serialized handle to the related object. Multivalued relationships use a Vector to store the handles of the multiple related objects. Since a Vector is serializable, it is then stored in the CMF for the multivalued relationship. The wrapper provides the required translation between the CMF and the required types.

Consider as an example, the pendingTrades relationship defined in the Stock component that represents the set of pending orders for the stock. The following code snippet illustrates the functionality of the addPendingTrades method that adds a PendingOrder instance to this multivalued relationship. The getCmpForPendingTrades() method provided by the container returns the CMF for the relationship, which is called cmpForPendingTrades. The retrieved object is cast to a Vector and the handle to the IRules wrapper for the pendingOrder is added to the Vector before it is made persistent by the call to the container-provided method setCmpForPendingTrades.

```
public void addPendingTrades(IRulesPendingOrder ir)
    {Object obj = getCmpForPendingTrades();
    Vector relatedPendingOrder = (Vector)obj;
    Handle handle = ir.getHandle();
    relatedPendingOrder.addElement(handle);
    Object ref = (Object)relatedPendingOrder;
    setCmpForPendingTrades(ref);}
```

The IRules Definition Language allows the application integrator to refer to purchased components and their methods. Therefore, the underlying implementation of these languages must translate a call to a method on the purchased component to its IRules wrapper, allowing the hooks into the IRules environment. Thus, the IRules Wrapper acts as a proxy for calling a method on its associated black-box component. Whenever a method on a black-box EJB is called from within the IRules environment (from an action of an integration rule or an application transaction), the IRules environment passes the control of execution to the corresponding method of the IRules wrapper. Every method in the black-box EJB has a corresponding method with the same method name in its IRules wrapper. The arguments to the method in the wrapper include all the parameters to the corresponding method in the black-box component and in the same order. There are additional parameters to pass the transaction context.

The Component Definition Language allows for the definition of events that are raised before or after a method call on the underlying black-box component. The IRules Wrapper for the component is responsible for triggering these method events to the IRules environment. Consider the case where an event is raised after a method call. After completing all of the preliminary actions needed by the IRules environment, the IRules Wrapper delegates to the business method of the black-box component to execute the business logic. After executing the method of the black-box component, control returns to the IRules Wrapper, which is then responsible for triggering the after method event. The wrapper bundles all the necessary information including the transaction context into a common IRules event data structure and publishes the occurrence of the afterEventName to the IRules topic via the JMS messaging service. The IRules Event Handler notifies the rule processor when a new event is detected and rule processing is done. Further detailed information about the execution environment can be found in a companion paper (Urban et. al. 2002).

6.2 Session Beans

The IRules wrapper for a deployed black-box session bean is also a session bean. Similar to the wrapper of an entity bean, the IRules wrapper wraps all of the business methods in the underlying black-box session bean and acts as a proxy to method calls on the purchased component. The IRules wrapper also generates the method-based events that are defined in the CDL. Since session beans do not represent persistent data, the IRules session bean wrapper does not support the additional attributes or externalized relationships supported by the IRules entity bean wrapper.

The IRules session bean wrapper is a stateful session bean. The IRules Wrapper holds a reference to the underlying black-box stateful session bean to maintain the conversational state across method calls. For a black-box stateless session bean, the IRules Wrapper creates a new instance of the underlying black-box component before invoking any methods on the purchased component. In the initial prototype that has been implemented, all of the required information to access the underlying black-box session bean, like the JNDI name of the black-box component it is wrapping and the EJB server URL has been stored as part of the state information of the IRules wrapper. The IRules session bean wrapper is designed as a stateful session bean to retain this information for the entire lifecycle of the IRules wrapper bean. The current design could be modified to obtain the information from environment variables, thus opening up the possibility of having an IRules stateless session bean wrapper that wraps a stateless black-box session bean.

7 Summary and Future Directions

This paper presented the IRules Component Definition Language and the metadata and wrapper framework required for supporting the IRules active approach to component-based software integration. The IRules environment acts as a mediator (Gamma et. al. 1995) in the integration process by encapsulating the logic describing the interconnections between components using integration rules.

The implementation described in this paper and its companion paper (Urban et. al. 2002) on transaction and execution control is a prototype of the technology based on the Investment example. Work is underway to develop a general-purpose system that uses Jini as the basis of the distributed computing environment.

There are also language issues to be investigated and implemented. We are currently developing a compiler for CDL that uses JavaSpaces for the storage of metadata and automatically generates the EJB wrapper code for the components. Although we have an initial design for the IRL, we are in the process of investigating condition evaluation techniques for IRL rule conditions that involve distributed query processing.

Acknowledgements

We want to thank Rohini Patil for her assistance in the refinement and illustration of the component metadata diagram.

This work was partially supported by a grant from the National Science Foundation (IIS-9978217).

References

[1] T. Adbellatif. An Architecture for Active Database Systems Supporting Static and Dynamic Analysis of Active Rules Through Evolving Database States, Ph.D. Dissertation, ASU, Dept. of Computer Science and Engineering, Fall 1999, 375 p.

[2] T. Abdellatif, R. Chan, S. W. Dietrich, B. Siddabathuni, A. Sundermier, and S. D. Urban, "Meta-Data Components in Support of an Active Deductive Object-Oriented Database System," Proc. of the 3rd IEEE Meta-Data Conference, Bethesda, MD, On-line publication: http://computer.org/conferen/proceed/meta/1999/, paper #16, 1999.

[3] K. Arnold, The Jini™ Specifications: 2nd Ed., Addison-Wesley Publishers, NJ, 2000.

[4] K. Ayyaswamy, "The Design and Implementation of a CORBA based environment for Distributed Constraint Maintenance," M.S. Thesis, Dept. of Computer Science and Engineering, ASU, 1999.

[5] D. Barrett, L. Clarke, P. Tarr, and A. Wise, "An Event-Based Software Integration Framework," ACM Transactions on Software Engineering and Methodology, vol. 5, no. 4, October 1996.

[6] BEA Systems Weblogic Server, http://edocs.beasys.com/wls/docs61/index.html

[7] G. Bultzingsloewen, A. Koschel, and R. Kramer, "Active Information Delivery in a CORBA-based Distributed Information System," Proc. of the First International Conference on Cooperative Information Systems (CoopIS'96), Brussels, Belgium, June 1996.

[8] R. G. G. Cattell, ed., The Object Database Standard, ODMG 3.0, Morgan Kaufmann, 2000.

[9] S. Chakravarthy and R. Le, "ECA Rule Support for Distributed Heterogeneous Environments," Proc. of the International Conference in Data Engineering, Orlando, 1998.

[10] M. DeJong and C. Laird, "TCL+Java = A Match Made for Scripting," http://www.sunworld.com/sunworldonline/swol-11-jacl.html.

[11] S. W. Dietrich, S. D. Urban, J. V. Harrison, and A. P. Karadimce, "A DOOD Ranch at ASU: Integrating Active, Deductive, and Object Oriented Databases," Data Engineering Bulletin, Special Issue on Active Database Systems, vol. 15, no. 1-4, December, 1992, pp. 40-43 (see also http://www.eas.asu.edu/~adood)

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Design, Addison-Wesley, Reading, MA, 1995.

[13] Java™ 2 Platform Enterprise Edition 1.3, http://java.sun.com/j2ee/

[14] A. Koschel and P. Lockemann, "Distributed Events in Active Database Systems: Letting the Genie Out of the Bottle," Journal of Data and Knowledge Engineering (DKE), vol. 25, pp. 11-28, 1998.

[15] R. Le and S. Chakravarthy, "Support for Composite Events and Rules in Distributed Heterogeneous Environments," Technical Report, Computer and Information Science and Engineering Dept., University of Florida, January 1998.

[16] C. Ma and J. Bacon, "COBEA: A CORBA Based Event Architecture," 4th Conference on Object Oriented Technologies and Systems (COOTS), New Mexico, April 1998.

[17] Microsoft Corporation. COM+, http://www.microsoft.com/com/tech/complus.asp

[18] Object Management Group: The Common Object Request Broker, Architecture and Specification, Revision 2.3, December 1998.

[19] Object Management Group: Relationship Service Specification, version 1.0, April 2000.

[20] N. Pissinou and K. Vanapipat, "Active Database Rules in Distributed Database Systems: A Dynamic Approach to Solving Structural and Semantic Conflicts in Distributed Database Systems," Computer Systems Science and Engineering, vol. 1, pp. 35-44, 1996.

[21] N. Pissinou , K. Makki, and R. Krishnamurthy, "An ECA Object Service to Support Active Distributed Objects," Informatics and Computer Science, pp. 63-104, 1997.

[22] J. Rumbaugh, "Relations as semantic constructs in an object-oriented language," Proc. of OOPSLA, 1987, pp. 466-481.

[23] A. Saxena, An Evaluation of Distributed Architectures for the Integration of Black-Box Software Components, M.S. Thesis, ASU, Dept. of Computer Science and Engineering, Tempe, AZ, Fall 2000.

[24] A. Silberschatz and S. Zdonik, "Database Systems – Breaking Out of the Box," ACM SIGMOD Record, vol. 26, no. 3, September 1997.

[25] S. D. Urban, A. P. Karadimce, S. W. Dietrich, T. Ben Abdellatif, and R. Chan, "CDOL: A Comprehensive Declarative Object Language," Data and Knowledge Engineering, vol. 22, 1997, pp. 67-111.

[26] S. Urban, S. Dietrich, A. Saxena, A. Sundermier, "Interconnection of Distributed Components: An Overview of Current Middleware Solutions," Journal of Computer and Information Science in Engineering, vol. 1, no. 1, March 2001, pp. 23-31.

[27] S. D. Urban, S. W. Dietrich, Y. Na, Y, Jin, A. Sundermier, and A. Saxena, "The IRules Project: Using Active Rules for the Integration of Distributed Software Components," Proc. of the 9th IFIP Working Conference on Database Semantics: Semantic Issues in E-Commerce Systems, Hong Kong, April 2001, pp. 265-286.

[28] S. D. Urban, A. Saxena, S. W. Dietrich, and A. Sundermier, "An Evaluation of Distributed Computing Options for the Integration of Black-Box Software Components," Proc. of the 3rd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, June 2001, pp. 100-109.

[29] S. D. Urban, S. W. Dietrich, A. Sundermier, Y. Na, Y. Jin, and S. Kambhampati, "Distributed Software Component Integration: A Framework for a Rule-Based Approach," To appear in the Handbook of Electronic Commerce in Business and Society, P. Lowry, J. Cherrington, and R. Watson (editors), CRC Press, 2002.

[30] J. Widom and S. Ceri, Active Database Systems: Triggers and Rules for Advanced Database Processing, Morgan Kaufmann Publishers, San Francisco, 1996.

DEPA (Design Pattern Application) – a component-based model for applying design patterns in software development

Katrina Ji
SOFTITLER NET Inc., 6464 Sunset Blvd., Suite 970, Hollywood, CA 90028, USA
katrinaji@yahoo.com

Sean Chen
Department of Accounting and Legal Studies, College of Charleston, Charleston, SC 29424-0001, USA
Phone: 843 953 8068, Fax 843 953 5697
seanchen@cofc.edu

Keywords: design patterns (DP), software development (SD), component-based software development (CBSD)

Received: June 5, 2001

This article reports the DEPA (Design Pattern Application) model – a component-based model for applying design patterns (DP) in software development (SD). Prior research has suggested the usefulness of DP in large, complicated SD projects. However, there is still a lack of formal models with which a software engineer could apply DP to SD in a systematic way. The DEPA model is one such formal method that allows systematic applications of DP. We have also illustrated how the DEPA model works in a realistic setting. It shows that the model can be applied to various domains. Further research is suggested in order to develop other models of DP applications.

1 Introduction

Design patterns (DP) are blocks and chunks of codes that are used to describe core solutions of recurring problems at an abstract level, typically in large-scale software development (SD) projects. When DP are used in a specific application, oftentimes there are multiple DP that may be applicable. To an experienced software engineer this may not be a problem, because he can quickly look into the DP that are available, decide on the cause of action, and choose the one(s) that will fit the application the best. For a novice software engineer, however, such a *"scanning-evaluation-selection"* process of DP may be problematic primarily due to the following two reasons:

1. The selection problem - Novice software engineers may lack the necessary knowledge or experience in making a sound choice as to what DP are most appropriate in a particular application.

2. The communication problem – They may not be able to understand why is a particular DP chosen by other more experienced software engineers.

This research project is motivated by our desire to develop a model that will address the above two problems. The end result of the project is the development of a formal model that provides a systematic way for a novice software engineer to select DP in DP while at the same time to allow them to know the underlying principles involved in the DP selection process thus reducing communication gaps among software developers.

Keller et al. (1999) has proven that DP provide great helps in understanding the complexity of large software systems. By applying DP, the SD process does not have to start from scratch. It results in savings in development time, in facilitating a productive software life cycle, and in providing a better communication among software engineers involved in the SD processes.

According to Gamma et al. (1995), DP contain the following features that make them an excellent tool in SD:

1. They do not give solutions in a specific application environment, but provide detailed descriptions of where, when, why and how a DP should be used.

2. They give key elements such as participants, structure, and collaborations that contribute to the solution to a problem.

3. They include discussions of consequences and implementation details when using a particular DP. It also suggests factors to consider if other DP are to be applied.

2 Design Pattern Application

Even though it is generally agreed that DP is a useful tool in large-scale SD, to date there is a lack of formal models of how DP can be applied in a systematic manner. In our literature review, we only found a brief suggestion by Gamma et al. (1995) on how to select a DP and how to

use it. Gamma et al. suggested the following steps in selecting a DP:

1. Consider how design patterns solve design problems
2. Scan intent section
3. Study how patterns interrelate.
4. Study patterns of like purpose.
5. Examine a cause of redesign.
6. Consider what should be variable in your design

These steps are general guidance for correctly selecting a DP. However, the real task of selecting a DP relies heavily on developers' experiences, their understanding of DP in general, and their familiarity with the target system to be implemented

Gamma et al. (1995) also suggested the following guidelines to decide the appropriateness of a DP when that DP is selected:

1. Read the pattern once through Structure, Participants, and Collaborations sections.
2. Look at the Sample Code section to see a concrete example of the pattern in code
3. Choose names for pattern participants that are meaningful in the application context.
4. Define the classes.
5. Define application-specific names for operations in the pattern.
6. Implement the operations to carry out the responsibilities and collaborations in the pattern.

The two sets of guidelines provided by Gamma et al. are helpful to assist the identification and application of DP to some extent. Nonetheless, there are limitations about how we could make the best use of DP in SD projects. That is to say that the aforementioned rules are somewhat ad-hoc, when we take a close look at them. Just as Gamma et al. (1995) pointed out, these steps *"are just guidelines to get you started."* Software design, in general, will require more detailed guidelines than the ones by Gamma et al. in above. The DEPA model is a formal framework that we developed in order to resolve such gap in DP application. Specifically, the DEPA model provides a solution to the following two problems:

1. *The lack of communication in software development*: Unlike a hardware product or a construction project, an implemented software product still needs to be upgraded, maintained, added new features, or deleted old features. This makes it essential to document the software design . In a large software system with thousands or even millions of lines of source code, it is difficult to understand the system by reading the source code that lacks proper system documentation. Because DP tell rationales behind software designs, such as why and how the software was structured, proper documentation when applying DP in SD can ease the system understanding and maintenance in the future. Our proposed DEPA model provides a systematic way to document the application of DP in SD.

2. *Multiple applicable DP*: DP are descriptions of solutions to recurring problems at an abstract level. They explain, in natural languages, why a particular DP exists; where, when and how it is applied; and potential problems and trade-offs associated with each application. Therefore, each DP may be applied in different ways, depending on the situation they are applied. The following is an illustration of the Singleton Pattern. It shows that there are more than one way to implement the *singleton* pattern, a DP described in Gamma et al. To avoid confusion in DP application, a sound model should be able to ensure that, when the DP is used, it chooses only one optimal DP in the software project. For example, for the singleton pattern described in Gamma et al. (1995) (see Figure 1 in below), there could be at least the following two different ways to implement it.

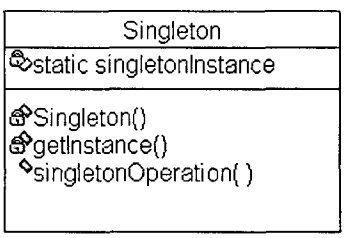


Figure 1: *Structure of a Singleton Design Pattern*

Method 1:

A sample code in C++ that applies the *Singleton* pattern to the MazeFactory class:

```
class MazeFactory{
  public:
      static MazeFactory* Instance();
      // existing interface goes here protected:
      MazeFactory ();
  private:
     static MazeFactory*  _instance;
};
```

The corresponding implementation is:

```
MazeFactory* MazeFactory::_instance = 0;
MazeFactory* MazeFactory::Instance() {
  if ( _instance == 0)
  {     _instance = new MazeFactory;
  }
        return _instance;
}
```

In addition to the above two situations, this pattern could be used to control the number of instances in an

application as well as to implement the pattern with other programming languages such as Java (Grand 1998).

Method 2:

The following is another way for sub-classing the Singleton class.

```
class Singleton {
public:
   static void Register(char* name, Singleton*);
   static Singleton* Instance();
protected:
   static Singleton* Lookup (const char* name);
private:
   static Singleton* _instance;
   static List<NameSingletonPair>* _registry;
};
```

The above example shows that, for different applications, the detailed implementation can vary significantly. Even for a simple Singleton DP, there are multiple ways to implement. A formal model in DP application needs to provide a way to choose a DP when multiple patterns may be applicable to a potential SD project. In our proposed DEPA model, we will address this issue.

Research groups have been studying and compiling common DP, and putting efforts in automating the use of DP. But there is only little progress in the methodology and the process of applying DP. Without such a methodology, how to convert a DP that is selected in the abstract level into software codes in the implementation level will still be difficult. Khriss et al. (1999) pointed out the need to develop a formal methodology that will direct software developers in a step-by-step procedure of how to apply DP. We believe that the DEPA model is an early attempt to fill the need of such a formal methodology.

A model directing the whole process of applying DP, which defines how they evolve from the abstract level to the implemented code level, makes the best use of design patterns. The documentation of the whole process will store the rationale behind the software design, which will make it easier for software maintenance once it is implemented.

3 The DEPA Model

3.1 Objectives

We proposes a model that details and defines the process between the abstract level and the implemented level. This model also describes a step-by-step guidance for applying DP. The DEPA model will provide the following three objectives:

1. To provide a step-by-step process of applying the solutions in DP in the *abstract* level to the programming language codes in the *implementation* level. All steps in the DEPA model should be easily followed by software engineers.

2. To provide a way of tracing the design process. Every step in the DEPA model should be documented to make it possible of tracing back to the original design specifications.

3. To establish a guidance for the future automation of the DEPA model that will be a tool with easy access and usage of DP in SD.

3.2 The Five Steps in DEPA

The DEPA model proposes the following five steps when applying DP in a specific SD: *generic and domain-specific*, *concrete*, *specific*, *integrated*, and *implemented* design patterns. The following sections define each of the five steps in the DEPA model.

3.2.1 Generic design patterns

Generic DP are descriptions of solutions in natural languages. They are generally published DP compilations (e.g. Gamma et al. 1995, Grand 1998) that are readily available to general public.

According to Gamma, elements of DP include intent or purpose, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses and related patterns. There are also other ways to describe DP. The purpose of the descriptions is to help understanding all aspects of a particular DP. Since the solutions given in DP are at the abstract level, they can not be used directly without considering the context in different applications. Only when a software developer understands a DP thoroughly and knows why it can be used in a specific context can he use that DP be effectively and correctly.

In the proposed DEPA model, the original DP with descriptions of solutions are converted into more detailed forms in order for them to be used in SD.

3.2.2 Domain-specific design patterns

Domain specific DP are core solutions to problems in a specific domain. They are similar to generic DP and are at the same level of abstraction as generic DP. The difference between the two is that generic DP are applicable in all areas, while domain specific DP may only be applicable to a certain area.

For example, the *Master-Slave* pattern, found in the development of mobile agents, is a domain specific design pattern (Lange et al. 1998). It defines a scheme whereby one agent, called *master*, can delegate a task to a *slave* agent. Figure 2 in below shows the structure of this pattern.

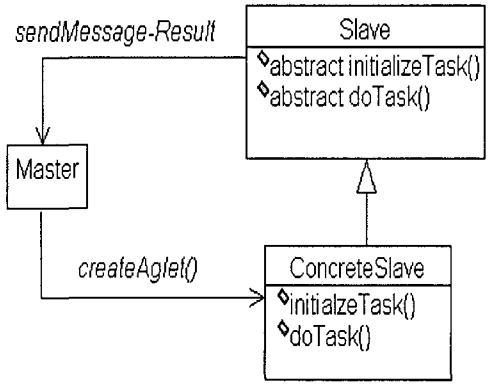


Figure 2: *Structure of master-slave pattern*

The description of the *Master-Slave* pattern consists of same elements as those of generic DP: i.e. *intent, applicability, participants, collaboration, consequences, implementation*, and *sample code*. But this pattern is only for applications in mobile agents, and it may not be applicable beyond this domain.

There are also other DP for different domain-specific applications. For example, patterns for network and communication (Schmidt 1996) and for the modeling of building simulators (Schutze et al. 1999).

The reason why we put a domain-specific DP at the same abstract level with a generic DP is that the former is also core solutions in a description of natural languages. Therefore, the process of applying a domain-specific DP is similar to that of a generic DP.

3.2.3 Concrete design patterns

Concrete DP eliminate the ambiguities in the generic/domain-specific DP level. In order to eliminate ambiguities involved in the selection DP from the generic/domain-specific DP level, the DEPA model weights the trade-offs, pitfalls, hints, and techniques associated with the selected DP to come up with concrete DP that will clearly show their intent.

The following is an example of several possible concrete DP derived from one generic DP – the *Observer* pattern:

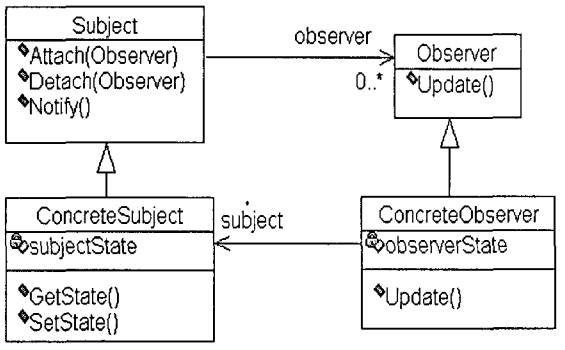


Figure 3: *The Structure of Observer Pattern*

The *Observer* pattern (Gamma et al. 1995) in Figure 3 defines a one-to-many dependency between objects, so that when a *subject* changes its state, all its *dependants* are notified and updated automatically. The *Observers*, in turn, will update their states after receiving the notification. The benefit of this pattern is to keep abstract coupling between the *Subjects* and the *Observers*, and to avoid being tightly-coupled. The Subject keeps a list of Observers, but do not know further details of any particular Observer.

There are several ways to convert this original Observer DP to a concrete DP:

1. An *Observer* may depend on several *Subjects*. In such case, an update interface should include the *Observer*. When the *Observer* updates its state, it will know which *Subject* has changed its state.

2. The update method may be triggered by a *Subject* or by a *Client*. The difference is that the *Client* may wait until a series of states being changed before notifying the *Observer*.

3. When the *Subject* broadcasts additional information about changes, the *Subject* may send *Observers* detailed information about the changes.[1] Otherwise the *Subject* may send only notifications, and *Observers* ask for details explicitly thereafter.[2]

4. The *Observers* may register in a *Subject* as only interested in specific events. In such case, the *Attach* and *Update* methods must have a parameter indicating the *Observers'* interests.

The above example illustrates that, when applying a generic/domain-specific DP, it is necessary to generate concrete DP will eliminate ambiguities involved in the generic/domain-specific DP. The concrete DP will not be represented by descriptions in natural languages as those in generic/domain-specific DP.

3.2.4 Specific design patterns

After the concrete DP eliminated ambiguities in the generic design states, the next step is to consider the specific requirements and situations for specific applications. In this step, Specific DP, it includes not only key solutions from concrete DP, but also detailed designs for specific applications.

The solutions in the concrete DP contain only *key* elements including classes, methods, and relationships among classes that have contributed to solving the problem. They are not directly related to the current

[1] Such process is called the *push* mode. See Gamma et al. (1995, p. 298)

[2] Such process is called the *pull* mode. Also in Gamma et al. (1995, p. 298)

application. For different applications, the concrete DP must be modified to fit into the environment of the specific system. This is a process of renaming those elements in concrete DP, and adding new elements for system requirements. After the specific design step, the DP are ready to be implemented.

The following is an example of applying the *Observer* pattern in a design for a gas station system. In a gas station, pumps control the dispensing of petrol, and screens show a volume of the petrol delivered. Figure 4 in below (Khriss et al. 1999) is a UML diagram for a specific design pattern of the *Observer* pattern.

In Figure 4, *Pump* class and *Screen* class are concrete classes or subclasses of *Subject* and *Observer* classes, respectively. Compared to Figure 3, some of the classes and methods in the concrete DP are renamed, and some application-specific methods are added to the classes.

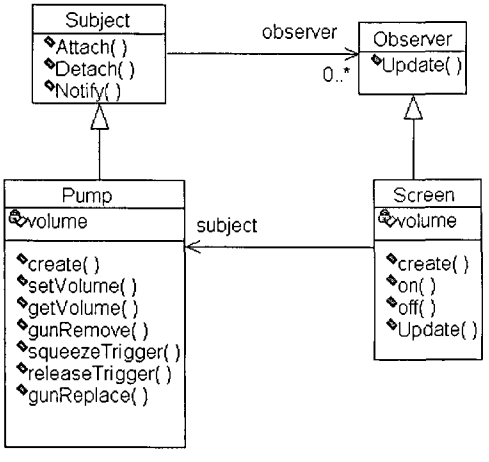


Figure 4: *UML diagram of a specific Observer pattern.*

The specification of DP includes renaming of classes and methods, cloning of classes, and changing and adding attributes, methods, and classes. The specific DP are detailed software designs before implementation, and the DP are embedded in the applied environment.

The above figure is a UML class diagram. The specific DP may also include sequence diagrams for the specific application that is based on the concrete DP. These diagrams form the analysis part of the specific DP.

3.2.5 Integrated design patterns

Integrated DP combine more than one specific DP. One DP may provide solutions to only one problem. For more complex systems, there may be more than one applicable DP. Integration is a process to combine all those applicable DP.

Various ways to integrate DP may be applicable, including *aggregation, composition,* and *containment.* Fowler & Scott (2000) describe *aggregation* as the *"part-of"* relationship, and *composition* as the *"part object"* belonging to only one *"whole". Containment* incorporates *layers* of DP into a particular DP

After the concrete and specific DP stages, the integration step needs to consider only the *interfaces* and *relationships* among specific DP. The integrated DP is a completed software design. In this paper, we will provide an example of an integrated DP using two specific DP.

3.2.6 Implemented design patterns

The implemented DP are programming codes. The implementation is based on the integrated DP, which may be implemented in C++, Java, or other object-oriented programming languages. It will be hard to recognize the original DP in the programming codes. But with the documentation of the whole process in the DEPA model, it will be easier to understand and maintain the implemented product.

3.2.7 Putting all things together – the DEPA model

Figure 5 in below shows the DEPA model and the five steps in the model:

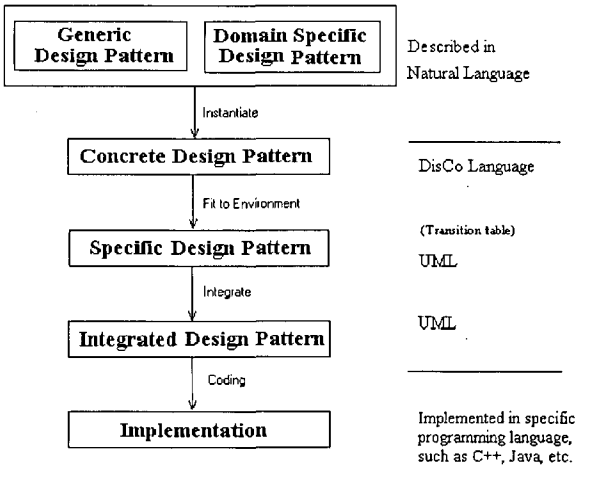


Figure 5: *The DEPA Model*

4 The DEPA Model – An Illustration

Figure 5 above shows that the generic DP are described in natural languages **with ambiguities**. In the DEPA model, different states of DP must be expressed in certain specifications. With the detailed specifications, DEPA is not only a tool for DP selection guidance, but also is a method to document every step in such selection process

In the DEPA model, the DisCo language is used to represent concreteDP. DisCo is a clear, easy to use language for representing concrete DP. With DisCo, the concrete DP can extract all key elements from the generic DP without ambiguities for the concrete DP.

Table 1 in below shows the representation of elements in a concrete DP:

Name	Expression
Class	**Class** class-name = { class-attributes}
Extended-class	**class** class-name = class-name + {class-attribute}
Relation	**Relation** (n-instance-class1) • Relation-name • (m-instance-class2) : class-name1 X class-name2
Method	Method-name(role-name: class-name; parameter): enabling_conditions → result_states
Extended-method	Method-name(role-name: class-name; parameter): **Refines** method-name(role-name: class-name; parameter): enabling_conditions → result_states
Inheritance class	**class** subclass-name = superclass-name + { class-attributes}
Method in Inheritance class	Method-name (role-name, class-name; parameter): **Refines** method-name1(role-name1: class-name1; parameter1) **for** role-name1 ∈ class-name

Table 1: *Representation a Concrete DP*

As a middle state of DP in the DEPA model, the concrete DP does not have ambiguities comparing with the generic DP. After considering the applicability, structure, participants, collaborations, consequences, and implementation in the generic DP, decisions concerning trade-offs, hints, suggestions, and the applied system were made. The resulting concrete DP is a concrete solution for the application system being developed. The concrete DP complete the following two tasks:

1. Keeping all the key information in generic design patterns.

2. Choosing a concrete solution for the applied system based on considerations of the trade-offs, hints, and suggestions in generic design patterns.

The concrete DP in the DEPA model have the following two unique features:

1. They provide reasons for the existence of all classes and methods.

2. The require documentation in the abstract level in the software design. Without the concreteness of eliminating the ambiguities in generic DP, it is unfeasible to apply the generic DP to software design.

The following sections provide a step-by-step illustration of how the DEPA model works.

4.1 From Generic/Domain-specific DP to Concrete DP

Generic/domain-specific design patterns include all helpful information for the concreteness. Concrete DP is the first step to apply generic sign patterns. The concrete DP only focuses on those elements that contribute to solutions in Generic/Domain-specific DP. The elements are:

- Classes in the DP.
- Relationships between the classes in the DP.
- Methods and attributes which contribute to the purpose of the DP.

The following is an example of concrete DP from an *Observer* DP introduced in Section 3.2.3.

From the action Notify(s:Subject, d) in the formalizing expression; we have

Expression 1

Notify(s:Subject, d):
→ ¬s.Updated . **class** Observer
∧ s.Data' = d

The *subject* is responsible to trigger the update.
The Update(s:Subject; o:Observer; d)
Update (s:Subject; o:Observer; d):
s.Attached.o
∧¬. Updated.o
∧ d = s.Data
→ s.Updated'.o
∧ o.Data'=d.

This expression indicates that this is a pull model. First, the *Subject* sends the notification, then the *Observer* asks for details later.

The following is another possible concrete DP from the same *Observer* pattern: the concrete solution is that the *Observer* is attached to the *Subject* according to the specific events of interest.

Expression 2

class Subject = {Data},
class Observer = {Data}.

Relation (0..1) . Attached . (*): Subject X Oserver.
Relation (0..1) . Updated . (*): Subject X Observer.

Attach(s:Subject; o:Observer; interest):
¬s. Attached.o ^ o.interest
→ s.Attached'.o
Detach(s:Subject; o:Observer; interest):
s.Attached.o ^ o.interest

```
        → ¬s.Attached'.o
          ∧ ¬s.Updated'.o
Notify(s:Subject, o.Observer; d; interest):
            o.interest
        → ¬s.Updated' . class Observer
          ∧ s.Data' = d,
Update (s:Subject; o*:Observer; d: interest):
            s.Attached.o ^ o.interest
        ∧¬. Updated.o
        ∧ d = s.Data
        → s.Updated.o
        ∧ o.Data'=d.
```

As we have introduced in section 3.2.6 about the concrete DP, there are multiple Concrete DP from the generic/domain-specific DP. The above expressions are only two possible Concrete DP from the *Observer* DP. With DisCo language, the concrete DP can express the solution clearly without ambiguities, and emphasize on the behaviors among classes. The concrete DP will then evolve to the specific DP with details in the individual classes.

4.2 From Concrete DP to Specific DP

The specific DP is the next step after the concrete DP with solutions for a problem. The specific DP direct software developers to focus on applying the concrete DP to the specific environment and on adding other features for the applied system.

The considerations in the specific DP are different from those in the concrete DP. While the former focuses on eliminating ambiguity in the generic/domain-specific DP and, the later concentrates on the application of the concrete DP to SD.

Modifications included in the specification are:

- Rename elements in the concrete DP, such as class, method, attributes, and parameters.
- Extend the classes and methods, when it is necessary for the applied system.
- Add classes or methods for other system requirements.
- Add more details in the classes or methods according to the specific applied system.

After the modification, the concrete DP will be embedded in the specific DP, and it will be hard to distinguish key elements originated from the concrete DP. To keep documentation of the evolution of a generic DP, the DEPA model keeps a *transition table* between the concrete DP and the specific DP, which is a mapping from class names, method names, and attribute names in the concrete DP to those in the specific DP. This table helps tracing back from the specific DP to the concrete DP.

In the DEPA model, the specific DP is expressed by UML, because UML emphasizes on the individual classes and their behaviors. After detailed design for the application requirements is established, the specific DP is ready to be implemented.

To use UML language, there is a transformation from the DisCo language to the UML. Table 2 in below gives more details about such transformation process:

DisCo Language	UML language
Class-name	Class-name
Method-name	Operation-name
Attribute, parameter	Parameter
Conditions and results	Pseudo code
Role name	Role name

Table 2: *Transformation from DisCo language to UML*

In a specific DP, the structure given in the generic DP has been embedded. By keeping the concrete DP in the DisCo language and the transition table, the contribution of the classes, methods, and why they exist in the specific DP are clearly documented. The documentation provides a method to understand the design during and after the SD.

The evolution of a DP from the Generic/Domain-specific, to the Concrete, and then to the Specific state is a process for applying all individual DP. The integration of more than one DP in the system development is the next step in the DEPA model.

4.3 From Specific DP to Integrated DP

DP provide core solutions for some recurring problems. However, most of the time, there are many problems to be solved in software design. After the Concrete and the Specific DP steps, integration of the Specific DP is the last step to form a component in SD.

Figure 6 in below shows an example of integrating *Template Method* and *Builder Pattern* in an application of the data set construction. Both the *Template Method* and the *Builder Pattern* are Generic DP in Gamma et al. (1995). The purpose of this design is to create a data set independent of the data source (Masuda et al. 1998)

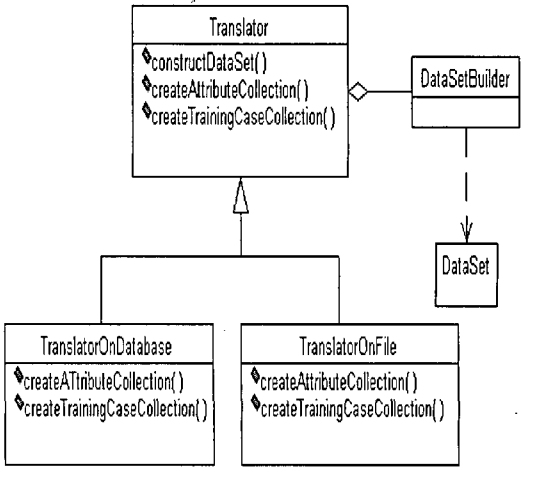


Figure 6: *Integration of Template Method & Builder DP*

In Figure 6, the *Builder* Pattern solves the problem of separating the construction of a complex object from its representation for the creation of *Dataset.* This solution makes it possible to change the internal representation of a data set object. The *Template Pattern* is applied to solving the problem of creating the *Dataset* from different data sources. It reuses the skeleton of the data creation algorithm.

By following the DEPA model, the software design is a process of problem solving with available solutions in Generic DP. In the Integration phase, all problems have been solved and the software design is ready to be implemented.

4.4 Implementing the Integrated DP

After integrating all the DP, the next step is to implement the design in object-oriented programming code. The implementation can be coded in specific programming languages such as C++ or Java.

With the guidance in the DEPA model and the documentation at each step, software developers have understood why and how the system is designed before implementation.

5 Conclusion and Future Directions

This research reports the DEPA (Design Pattern Application) model, a component-based model for applying design patterns (DP) in software development (SD). We have discussed the lack of a formal model in applying DP as an area that merits further research, and have developed the DEPA model that allows a systematic way of applying DP in SD projects, particularly to those SD projects with resource constraints. We have also provided a step-by-step illustration to show how such model will work in a realistic setting.

The DEPA model is a guidance of a procedure of clarifying the ambiguities and imprecision in the original DP. Without eliminating those ambiguities, the Generic/Domain-specific DP cannot be applied in a specific software design context. We believe that the DEPA model has achieved our objectives by supplying ways to convert Generic/Domain-specific DP to Concrete DP, to Specific DP, and to Integrated DP. With the developed Integrated DP, the last stage in SD – implementation and coding – will be easily achievable even for novice software engineers.

Although there has been prior research that intends to come up with a general method of formalizing design patterns, there still lacks a standardized DP representation. For example, Pree (1994) introduced the concept of meta-patterns. He used seven basic meta-patterns to represent design patterns on a meta-level. Florijn (1999) proposed fragment model that uses fragments to represent the structure of DP. Mikkonen (1998) discussed how to formalize DP by using DisCo Language. Eden & Hirsheld (1999) used meta-language and proposed a pattern-wizard of transforming DP from one language to another. However, none of the above research settles the formal representation issue.

According to Eden & Hirsheld (1999), there are following reasons for the formalization of DP:

1. Existing specifications contribute little or nothing to the understanding of when and how to use a design pattern.

2. Patterns are abstractions, or generalizations, and therefore are vague, ambiguous, and imprecise.

3. Formalization is impossible because there is no fixed elements in patterns, and everything can be changed.

4. Patterns are core solutions or concepts whose essence is intangible, elusive, and hence beyond the scope of a literal expression.

We believe that the aforementioned reasons fully justify the need of a generalized and standardized DP presentation, so that software engineers and researchers alike could use the presentation to develop more DP in the future.

6 REFERENCES

[1] Eden, A. E. & Hirsheld Y (1999) On the understanding of software patterns. http://www.math.tau.ac.il/~eden.

[2] Forijn, G. (1999) Tool support for object-oriented (design) patterns. http://www.serc.nl/people/florijn/work/patterns.html

[3] Fowler, M. & Scott, K. (2000) *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Publishing Co.

[4] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Co.

[5] Grand, M. (1998) *Patterns in Java* (Vol. 1). John Wiley & Sons, Inc.

[6] Khriss, I., Keller, R. K. & Hamid, I. A. (1999) Supporting Design by Pattern-based Transformations. *Proceedings of the Second International Workshop on Strategic Knowledge and Concept Formation*, pp. 157-167.

[7] Keller, R. K., Schauer, R., Robitaille, S. & Page, P (1999) Pattern-based Reverse-Engineering of Design Components. *Proceedings of the International Conference on Software Engineering,* pp. 226-235.

[8] Lange, B. L. & Oshima, M. (1998) *Programming and Deploying Java Mobil Agents with Aglets.* Addison-Wesley Publishing Co.

[9] Masuda, G., Sakamoto, N. & Ushijima, K. (1998) Applying Design Patterns to Decision Tree Learning System. *Proceedings of the SIGSOFT 1998Conference.* pp. 111-120.

[10] Mikkonen T. (1998) Formalizing Design Patterns. *Proceedings of the 1998 International Conference on Software Engineering*, pp. 115 – 124.

[11] Pree, M. (1994) Meta Patterns – A Means for Capturing the Essentials of Reusable OO Design. *Proceedings of the Eighth European Conference on Objected-Oriented Programming* (ECOOP), pp. 150-162

[12] Schmidt, D. C. (1996) A Family of Design Patterns for Flexibly Configuring Network Services in Distributed Systems. *Proceedings of the 1996 International conference on Configurable Distributed Systems.*

[13] Schütze, M., Riegel, J. P. & Zimmermann, G. (1999) PSiGene - A Pattern-Based Component Generator for Building Simulation, *Journal of Theory and Practice of Object Systems* (TAPOS), Vol. 5, No. 2, pp. 83-95.

An approach for modeling components with customization for distributed software[1]

X. Xie and S. M. Shatz
Concurrent Software Systems Lab, University of Illinois at Chicago, USA
Phone: +1 312 996 5488, Fax: +1 312 413 0024
shatz@cs.uic.edu

Keywords: Component Customization, Design Models, Distributed Software, Petri Nets

Received: June 4, 2001

Component-based software development has many potential advantages, including shorter time to market and lower price, making it an attractive approach to both customers and producers. However, component-based development is a new technology with many open issues to be resolved. One particular issue is the specification of components as reusable entities, especially for distributed object-oriented applications. Specification of such components by formal methods can pave the way for a more systematic approach for component-based software engineering, including design analysis and simulation. This paper discusses an approach for blending Petri net concepts and object-oriented features to develop a specification approach for distributed component software systems. In particular, a scheme for modeling behavior restriction in the design of object systems is presented. A key result of this work is the definition of a "plug-in" structure that can be used to create "subclass" object models, which correspond to customized components. Algorithms that support the automatic synthesis of these models are provided, discussed, and illustrated by examples.

1 Introduction and Motivation

There is significant interest in using components in software development. Specification and implementation of a system in terms of existing and/or derived components can dramatically decrease the time required for system development, increase the usability of resulting products, and lower production costs [1]. However, component-based development is still immature, with a lack of established procedures and support from formal modelling. Techniques and tools that are based on formal methods can pave the way for advanced software engineering capabilities such as design analysis and simulation.

Reuse principles have typically placed high demands on reusable components. Such components need to sufficiently general to cover the different aspects of their use, while also being simple enough to serve a particular requirement in an efficient way. This has resulted in a situation where developing a reusable component may require three to four times more resources than developing a component for particular use [2]. Thus, component vendors desire to make full use of these components in order to achieve reasonable profit levels. Such component use requires the customization of general components, a process that is aided by applying different constraints to functionality to support different price policies and different user groups.

Often a component is not effectively reusable because its interface or part of the implementation does not match the specified requirements of a target application. To achieve the reuse, the component needs to be customized into another component that fulfills the requirement [3]. One purpose of the customization is to apply constraints in situations where the functionality of a "base component" is more general than is actually needed, or when some base-component's features exhibit characteristics not suitable for a particular application – for example some functions (or methods) may not be fault tolerant or may be resource hogs. Thus, the component's behavior must be restricted before it can be reused in a new design.

One potentially efficient and natural technique to support constraints is a particular type of inheritance known as restriction inheritance [4]. Since subclassing by restriction often conflicts with the semantics and intention of inheritance, where an instance of a subclass should be an instance of the superclass and should behave like one, some researchers have suggested that restriction inheritance be avoided [1][5]. But, in our own experience, which does involve development of commercial component-based software, we have observed benefits of restriction inheritance for

[1] This material is based upon work supported by, or in part by, the U.S. Army Research Office under grant number DAAD19-99-1-0350 and by NSF under grant number CCR-9988168

customizing components. First, most commercial component-based software is based on middleware technologies such as CORBA [6] and/or COM [7]. As a result, these systems mostly consist of classes. In COM, even the interface of a component is a class. So, it seems natural to use inheritance techniques (defined for class-based systems) to handle constraints. Also, restriction inheritance is efficient, simple and straightforward. Finally, since restriction inheritance is being used for the purpose of defining a wrapper for components, the original components and/or class is not intended to be used directly, which limits any potential disadvantage associated with the use of restriction inheritance.

To develop a systematic design process with the capability for automated simulation and analysis, it is valuable to define a design method's syntax and semantics in terms of some formal notation and method. For engineering of distributed object systems, it is desirable for the formalization to provide a simple and direct way to describe component relationships and capture essential properties like non-determinism, synchronization and concurrency. Petri nets [8] are one formal modelling notation that is in many ways well matched for general concurrent systems. In particular, the standard graphical interpretation of Petri net models is appealing as a basis for a design notation. But, standard Petri nets do not provide direct support for high-level design and object oriented features. This has motivated some recent research into methods for combining Petri net modelling and object-oriented design. In general, the proposed methods use enhanced forms of Petri nets as a base of the combination, and pursue two main approaches [9]. One is the "objects inside Petri nets" approach, in which the semantics of tokens in Petri nets are expanded to include other information, which could include object definitions (e.g., [10]). The other approach is the "Petri nets inside objects" approach, in which traditional Petri net constructs are used to model the internal semantics of object (e.g., [11]).

In this paper we introduce a model called a State-Based Object Petri Net (SBOPN), which is developed from the basic idea introduced in [12]. An example of using SBOPN concepts in the domain of aspect orientation is described in [13]. In this paper we extend the basic SBOPN model to directly support restriction inheritance modelling for the purposes discussed earlier. SBOPN is most similar in spirit to Lakos' Language for Object Oriented Petri Nets, LOOPN [10]. LOOPN's semantics are richer, but SBOPN provides a more specific, and thus more intuitive, notation for capturing the behavior of distributed state-based objects. Like LOOPN, SBOPN is based on a generalized form of Petri net called colored Petri nets [14]. One other difference between LOOPN and SBOPN is that the primary encapsulator of object behavior in LOOPN is tokens, while SBOPNs use separate Petri net objects whose states are captured by special colored tokens. Another language, namely CO-OPN/2 [15], is also a "Petri nets inside objects." CO-OPN/2 uses high-level Petri nets that include data structures expressed as algebraic abstract data types and a synchronization mechanism for building abstraction hierarchies to describe the concurrency aspects of a system. CO-OPN/2 is a general model that focuses on concurrency. SBOPN focuses more on the architectural modeling of state-based systems; thus it is simpler and more domain-specific.

The structure of this paper is as follows. Section 2 provides details on SBOPN modeling and discusses the restriction subclasses and SBOPN control places. Section 3 describes our approach for synthesis of subclass models that capture instances of restriction inheritance. The approach is characterized by the use of special net structures called "plug-in structures." Finally, Section 4 provides a conclusion and mentions some future work.

2 Subclass Component Models and Control Places

In this section, we discuss how to derive design models for subclass components. Due to lack of space, we omit the formal definition of the SBOPN model, which can be found in [16]. A SBOPN model consists of a set of individual object models, called State-Based Petri Net Objects (SBPNO).

A SBPNO is denoted graphically as a Petri net (a subnet) inside a box and a State-Based Object Petri Net, SBOPN, is a Petri net consisting of connected SBPNOs, which are components of the system being considered. A marking of a SBOPN is the distribution of state tokens to the SBPNO components, and an SBOPN system (N, M_0) is an SBOPN, N, along with an initial marking M_0 (the initial states of the objects). In a SBOPN system, a transition t is said to be *enabled* if and only if, for each $p \in {}^\bullet t$ (where ${}^\bullet t$ is the set of input places for transition t), p contains a token whose state value is an element of the state-filter for arc (p, t). When an enabled transition fires, it removes from each input place a token whose state value satisfies the corresponding state filter, and then deposits a token in each output place. The state value assigned to a deposited token is one of the elements given as an output of the corresponding state-transfer function. For example, assume an arc (t, p) with the state-transfer tuple (q, f), where the state-transfer function $f(x) = \{x\}$. Then the firing of transition t will deposit a token into place p and the state-value of this deposited token will be equal to the state-value of the token removed from place q.

Consider the classic example of a system that uses a bounded buffer to temporarily hold items, such as messages. In this version we allow an operator to enable and disable the buffer, in addition to the standard producer and consumer components. The four system components – buffer, producer, consumer and operator – operate asynchronously and only interact via messages initiated by the producer (*put* message), consumer (*get*

message) or operator (*enable* and *disable* message). In particular, the producer sends *put* messages to the buffer when the producer has some new item to be deposited into the buffer and the consumer sends *get* messages to the buffer when the consumer desires to remove an item from the buffer. Also, the operator can send *enable* or *disable* message to enable or disable the buffer. At any point in time, the buffer should be in one of four states: *Empty*, *Full*, *Partial* (means *Partially Full*) or *Disabled*. Depending on its state, the buffer may or may not be able to accept the messages *put*, *get*, *disable* and *enable*. When the buffer is in *Empty* or *Partial* state, it can accept the *put* message and change to *Partial* or *Full* state. When it is in *Partial* or *Full* state, it can accept the *get* message and change to *Empty* or *Partial* state. When it is in any state except the *Disabled* state, it can accept the *disable* message and change to the *Disabled* state. Finally, when it is in the *Disabled* state, it can accept the *enable* message and change to its previous state (before it was disabled): *Empty*, *Partial* or *Full*. To simplify the example, we simply assume that after accepting an enable message, the buffer is reset to *Empty* state. Figure 1 shows a simple SBOPN model for this system. Because we do not model a specific buffer bound, the model is imprecise with respect to dependencies between the get and put methods.

To simplify SBPNO models, implicit state filters and implicit state-transfer tuples are allowed, i.e., definitions are assumed if they are not explicitly specified. For an implicit state filter, the state-filter is *States*. Note that in Figure 1, the state filters are implicit in the producer, consumer, and operator objects. An implicit state-transfer tuple can be used only when the output place associated with the arc is an input place of the transition associated with the arc – the arc is part of a self-loop. The state-transfer place is the place in the self-loop. We also require an implicit state-transfer function's output to be the state-value of the token removed from the place in the self-loop. Due to the simplicity of the producer, consumer, and operator object models, the state-transfer tuples are also implicit.

Now we can identify properties of a restriction subclass and present the definition of a restriction subclass model. First, the methods of a restriction subclass object should be a subset of the methods of the superclass object. Second, the externally observable behavior of a restriction subclass object should be observable in the behavior of the superclass object. In other words, any firing sequence (defined as in standard Petri nets) of a SBPNO subclass model should be a firing sequence of the superclass model when we only consider the shared transitions. In the following definition we use the notation $\sigma|_T$, a projection of σ onto T. As an example of this projection, let $\sigma = t_1t_2t_1t_3t_2$, and $T = \{t_1, t_3\}$, then $\sigma|_T = t_1t_1t_3$..

Definition 1 (Restriction Subclass Model): Let $N_1 = (Type_1, NG_1, States_1, ST_1, SFM_1, STM_1)$, $N_2 = (Type_2, NG_2, States_2, ST_2, SFM_2, STM_2)$ be two SBPNOs, then N_2 is *a restriction subclass model* of N_1 if and only if:

1) $ST_2 \subseteq ST_1$
2) For any marking M_2 of N_2, there exists a marking M_1 of N_1, such that for any firing sequence σ_2 of (N_2, M_2), there exists a firing sequence σ_1 of (N_1, M_1), which satisfies $\sigma_1|_{ST1} = \sigma_2|_{ST2}$.

A particular restriction subclass model must be defined in terms of some particular superclass model and some specific method restrictions. These restrictions are captured by a restriction function, as defined next.

Definition 2 (Restriction Function): Let $N_1 = (Type_1, NG_1, States_1, IS_1, Stoken_1, ST_1, SFM_1, STM_1)$ be a SBOPN, and let function f: $SF_1 \rightarrow 2^{States1}$, where SF_1 is the domain of SFM_1, and $2^{States1}$ is the power set of $States_1$. The function f is called a *restriction function* for N_1 if and only if f satisfies: $\forall sf_1 \in SF_1, f(sf_1) \subseteq sf_1$.

Applying f to the state filters of N_1 creates a new model, which we denote as $N_1|f$. It can be shown that $N_1|f$ is a restriction subclass model of N_1, but note that $N_1|f$ features a disadvantage: the change in the interface makes it difficult to directly identify that the new object is one of many possible behaviourally restricted objects derived from a common object. Our goal is to create a "plug-in" structure that can be added to a superclass model causing it to have the same behaviour as $N_1|f$ but avoiding the disadvantage. Such a plug-in structure must be able to control the firing of some shared transition t. This is accomplished by using a so-called "control place" as the heart of the plug-in structure. The control place must ensure that the state-value of a token in the control place "tracks" the state-value of a token in one of the input places p to the transition t. We call such a place p the "controlled place."

Definition 3 (Control Place): Let $N = (Type, NG, States, ST, SFM, STM)$ be a SBPNO, and p_1 and p_2 be two places of N. We say that p_2 is a control place for p_1 (p_1 is a controlled place) if and only if:

1) $(ST \cap p_2^{\bullet} \neq \varnothing) \wedge (ST \cap p_2^{\bullet} \subseteq ST \cap p_1^{\bullet})$ (Note: $p_1^{\bullet}$ is the set of output transitions of the place p_1).
2) For any shared transition $t \in (ST \cap p_2^{\bullet})$, the associated state filter for the arc $(p2, t)$ is a subset of the state filter for the corresponding arc $(p1, t)$.
3) For any reachable marking M' from M, which satisfies $M(p_1) = M(p2)$, and any transition $t \in (ST \cap p_2^{\bullet})$, if t fires under M', then the tokens consumed by t from p_1 and p_2 should have the same state values.

3 Synthesis of Plug-In Structures

3.1 Basic Plug-in Design

A straightforward way to implement a control place is to create a duplicate place. The basic idea has two steps. First, we duplicate the controlled place, such that the new

place has exactly the same input and output characteristics as the controlled place. Obviously, any change in the marking of the controlled place is simultaneously reflected in the marking of the new duplicated place. Because the new SBPNO (created by the duplication process) has the same exact behaviour as the original SBPNO, the new SBPNO serves as a (trivial) restriction subclass. In the second step, we modify the state filters for the arcs from the new place to all shared transitions such that they satisfy the specific requirement of the particular desired restriction subclass. This creates a model for a customized component. Recall that the specific restriction requirement (i.e., the customization feature) is determined by a restriction function, as defined in Definition 2.

Although a duplicating place can be used to create a control place and thus build a restriction subclass without changing interfaces, there is one significant disadvantage: *redundancy*. For example, in creating the "disable-free synchronous" buffer model from Section 2, we do not want to change the firing conditions of the *enable* and *get* methods. But it is necessary for the control place to connect with the associated shared transitions. Also, these additional arcs must carry the same state-filters and state-transfer functions as in the superclass model. Such extra arcs, which do not change the behaviour of the methods, imply an existence of redundancy in the new model.

Since our goal is to ensure that the state-marking of a control place "tracks" that of the controlled place, we can copy the token of a controlled place into the control place, but we must be sure that this copying occurs before allowing these places to enable any shared transition. We call this type of control place a "refreshing place" since it gets refreshed (i.e., the state-value of its current state token is updated) each time the state-value of the token in the corresponding controlled place changes. Figures 2, 3 and 4 illustrate this idea by a simple example. In Figure 2, we have a SBPNO for a component *C1*. Now suppose we want to model a restriction subclass *C2* that has the property that t_1 can be enabled only when the object is in the state *a* – instead of either state *a* or *b*, as in the component *C1*. We need t_2 to remain enabled in the *a* state.

To model this subclass, we create a new place p_2 (see Figure 3) as a control place candidate. Transition t_3 is introduced for the purpose of copying the state token from p_1 to p_2. As in the duplicating place technique, the state filter associated with p_2's connection to t_1 is *{a}*. However, under the general firing rule that controls the behaviour of a SBPNO, we cannot guarantee that the tokens in p_1 and p_2 are of the same value when t_1 is enabled. For example, in Figure 2, suppose p_1 has initial state *a*, then the firing sequence is $t_1^* t_2 t_1^*$. Now consider Figure 3, where both p_1 and p_2 have initial state *a*. Once t_2 fires, p_1 has state *b*, while p_2 still has state *a*. If t_3 does not yet fire, p_1 and p_2 have different states, but t_1 is still enabled. As a result, we could get the same firing sequence as *C1*, $t_1^* t_2 t_1^*$. However, *C2* is supposed to only allow the restricted firing sequence $t_1^* t_2$, where we ignore the internal transition t_3 in the firing sequence. So the construction in Figure 3 does not yet provide for a proper modelling of the control place.

The problem is that when t_2 fires, the token in p_2 remains unchanged and thus is not "tracking" the marking of p_1. To solve this problem, we need to force t_3 to fire immediately after t_2 fires, i.e., to refresh p_2 immediately. This is accomplished by using a special form of Petri net arc called an activator arc [17]. An activator arc can be used to connect a place to a transition. For nets with activator arcs, the transition firing rules are as follows: 1) Those enabled transitions with activator arcs have the highest priority, and 2) A transition that has activator arc input(s) cannot fire twice in succession for the same input marking, i.e., the net's marking must be modified in some manner before the transition can fire again. For example, in Figure 4, t_1, t_2 and t_3 are enabled, but t_3 has an activator arc (denoted by the arc with a solid bubble), so it fires first. After firing t_3, we get the same marking, so t_3 cannot fire again. As a result, only t_1 or t_2 can next fire. Now, if t_1 fires, because the marking remains unchanged, we have the same situation as before t_1 fires. But if t_2 fires, both t_1 and t_3 are enabled. Since the marking has changed, only t_3 can fire, which copies the token *b* from p_1 to p_2, i.e., p_2 is refreshed. This copying of the state-value from *p1* to *p2* is due to the state-transfer function *F3*. Note that t_1 is not enabled any more after t_3 fires. As we can see, now p_2 serves as a proper control place to ensure we have only one firing sequence $t_1^* t_2$ (again, ignoring the internal transition t_3 in the firing sequence).

We now present two algorithms for synthesis of restriction subclass models using plug-in structures. The first algorithm is used to create a refreshing place. Its purpose is to support the second, more important, algorithm, which synthesizes a restriction subclass model.

Algorithm 1: Create a refreshing place in a SBPNO.
Input: A SBPNO N = *(Type, NG, States, ST, SFM, STM)*, and a place p_1 that satisfies $p_1^* \in ST$.

Output: A new SBPNO (a modified version of N) with a refreshing place p_2 for p_1.

Procedure:
1) Add to N_1 a place p_2 and a transition t'.
2) Add an arc r_1 from p_1 to t', and an arc r_2 from t' to p_1.
3) Add an arc r_3 from t' to p_2, and an arc r_4 from p_2 to t'. Use (p_1, F) as the state-transfer tuple for r_3, where F is defined as $F(x) = \{x\}, x \in States$.
4) Add an activator arc from p_1 to t'.

As an example, applying Algorithm 1 to the SBPNO in Figure 2 creates part of the SBPNO shown in Figure 4 –

all of the model except the arcs *(p2, t1)*, *(t1, p2)* and the state-filter *{a}* for the arc *(p2, t1)*.

Algorithm 2: Model a restriction subclass by use of plug-in structures

Input: 1) A SBPNO $N_1 = (Type_1, NG_1, States_1, ST_1, SFM_1, STM_1)$.

2) A restriction function (see Definition 2), $f: SF_1 \rightarrow 2^{States1}$

Output: A restriction subclass model N_2 of N_1 (N_2 has the same externally observable behavior as the model $N_1|f$ identified in Section 3).

Procedure:

1) Make a copy N_1. Call this new model N_2 and let N_2 be the source net for the following step:
2) For each transition t in ST_1:
   For each $p_1 \in t^{\bullet}$, let $S1$ be the state filter for the arc (p_1, t). If $S2 = f(S1)$ is a proper subset of $S1$, i.e., $S2 \neq S1$, then create a control place p_2 of p_1 by applying the following steps:
   A. Use Algorithm 1 to create a refreshing place p_2 of p_1.
   B. Add an arc r_1 from p_2 to t. Use $S2$ as the state filter for r_1.
   C. Add an arc r_2 from t to p_2
   End For
   End For

The initial marking of a subclass model created by Algorithm 2 is determined by the initial marking of the superclass used to create it. All places except the created control places have the same initial marking as in the superclass model. The control places take on the same initial marking as their corresponding controlled places. As an example, applying Algorithm 2 to the SBPNO in Figure 2 creates the SBPNO shown in Figure 4. In this case, N_1 is the model shown in Figure 2 and the restriction function f is defined as $f(\{a, b\}) = \{a\}, f(\{a\}) = \{a\}$. Note that the structure within the dashed box in Figure 4 is the plug-in structure. As we can see, Figure 4 is more complex than Figure 2. And the switchable plug-in structures introduced in next subsection are even more complicated. Our proposal of modeling restriction inheritance would not be practical if we have to manually handle this complexity introduced by plug-in structure. Fortunately, since the synthesis of restriction subclass models is based on an algorithmic process, automated tools can be used to hide the internal details of modeling and analysis.

3.2 Switchable Plug-in Structures

One advantage of Algorithm 2 is that the plug-in structures created are potentially controllable. By controllability we mean that a switch can be added to the structure to control its activity, i.e., the switch can be used to "turn on" or "turn off" the functionality of the plug-in structure. We call such a plug-in a "switchable plug-in." Switchable plug-ins offer a key advantage: They allow a model to represent a family of restriction subclass models, corresponding to a family of components. The basic idea is that a single component-model with n potential customisations (defined by n plug-in structures) can in fact model a family of 2^n customized components. The family members correspond to the various combinations of enabled customisation features. This technique will be discussed shortly by a specific example.

To transform a plug-in structure into a switchable plug-in, a new place node must be added. For example, Figure 5 shows the same model as Figure 4, but with a switchable plug-in. Place p_3 serves as this new switch place. When there is a token in the switch place p_3, the "plug-in" structure is active. In this case, the plug-in behaves as before we introduced the switch place, i.e., like Figure 4. But when there is no token in p_3, the transition t_3 will never be enabled. So, in this case, the model behaves as before we introduced the plug-in, i.e., like Figure 2. Notice that we have introduced a new state value called *internal* to the state set. Although it is possible to create the switching capability for this particular example without introducing this new *internal* state, use of this special state is required for creating general-purpose switchable plug-ins. To explain this point, consider the following situation.

Suppose that we wanted to create a subclass *C3* of class *C1*, where *C3* does not support method *t1* at all. In this case, by Algorithm 2, the SBOPN for class *C3* would look like the model in Figure 4, except that the state filter for the arc (p_2, t_1) would be ϕ instead of *{a}*. Now, to make the plug-in of this model switchable, we would introduce a switch place *p3* as was done in Figure 5. But, since the state filter is the empty set, there is no way for the switch place to enable transition *t1* – it is always disabled, regardless of the state value of the token we put in *p2*. So, it is clear that in a *switchable* plug-in we cannot allow ϕ as the state filter for a restricted transition. A simple solution is to introduce a new state value that is reserved for use within the switchable plug-in structure. This is the *internal* state referred to earlier. Now, the state filter can become *{internal}*, as opposed to ϕ. To create the initial marking of this subclass *C3*, it is necessary that the initial markings of the control place *p2* and the switch place *p3* have the state-value *internal*. In general, to model a restriction subclass using switchable plug-ins, we can use Algorithm 2 with the following two simple modifications:

1. For each plug-in, create a switch place (connected to/from the transition for the refreshing place).
2. For each plug-in, modify the state filter (for the arc from the control place to the restricted transition) to include the state *internal*.

As an example, let us revisit the buffer example from Section 2. Now, the modified algorithm mentioned above

can be applied to the model in Figure 1 to create a model for a "disable-free synchronous" buffer. The resulting model (with two switchable plug-ins) is shown in Figure 6. Note that the initial marking of all places belonging to plug-ins are *internal*. Note that the plug-in associated with the disable method employs a state filter of {*internal*}. Thus, if this plug-in is "turned-on" (by marking the switch place), the disable method will become inactive. For the plug-in associated with the put method, the state filter is set to {*i*, *Empty*}. Thus, the put method is active only when the buffer is in the empty state. Most importantly, note that this one subclass model actually models a family of buffer types. The binding of the model to a specific buffer behavior is accomplished by varying the initial markings of the switch places (*p2'* and *p3'*). The following table defines the options:

p2'	*p3'*	Model
Marked	Marked	A "disable-free synchronous" buffer
Marked	Unmarked	A "disable-free" buffer
Unmarked	Marked	A "synchronous" buffer
Unmarked	Unmarked	A general buffer

The ability to model a family of components can be very helpful for commercial component-based development. It supports flexible analysis of varying configurations of customized components in the design phase, which can reduce the overall cost of development. This has the potential to aid configuration management and support, which is becoming a major challenge that organizations face in component-based software development [18].

3.3 Some Analysis Issues

Basic SBOPN models (without plug-in structures) are derived from standard colored Petri nets. Basic SBOPN models, with state filters and state-transfer functions, can be transformed into colored Petri nets [12]. This is important since we want SBOPN models to be able to use a full set of analysis techniques already existing for mature models like colored Petri nets or ordinary Petri nets. But, the subclass models that correspond to customized components in this paper use activator arcs. Thus, we must understand the impact of these arcs in terms of analysis potential. After all, activator arcs are special arcs with unique semantics. In the generally case, there is no equivalent ordinary Petri net structure for a Petri net with activator arcs. But, in our models, activator arcs are used only in plug-in structures. Thus, it is possible to convert an SBOPN model with activator arcs to a general SBOPN model and preserve liveness, safeness and boundedness of the model. To simplify our discussion, we use Figure 5 as an example to explain some key aspects of this translation. The results apply in general.

Consider the switch place *p3* in Figure 5. In the case that *p3* is not marked, it can be observed that removal of the plug-in will not change the liveness, safeness and boundedness properties of the model. Now consider the case when *p3* is marked. In this case, *p3* can never disable *t3*. Thus, *p3* and the corresponding arcs can be removed without changing the model's behavior. From the structure of the plug-in, it is clear that the plug-in will not affect the safeness or boundedness of the model. A similar analysis of *p2*'s impact on the liveness of the model confirms that that both state-filters (on the arcs *(p1, t1)* and *(p2, t1)*) can be changed to *{a}* without changing the liveness property of *t1*. Now, since both state-filters associated with *t1* are equal, and whenever *t1* fires, the tokens in *p1* and *p2* have identical state-values, the plug-in structure can be removed without impacting the liveness of *t1*. Furthermore, because of the 1-to-1 correspondence between a plug-in and a shared transition, the translation just described does not impact the liveness of transition *t2*. Further conversion of an object model to a colored Petri net or ordinary Petri net is now assured, providing a basis for various analysis capabilities. Further discussion on specific analysis techniques using these lower-level, basic net models is beyond the scope of this paper.

4 Conclusions and Future Work

One challenge in component-based software engineering is to find techniques and tools that are effective in aiding the specification and design of component-based systems. One way to increase the effectiveness of these design techniques is to employ formal methods that provide a well-defined design notation and support design analysis. From our research, and experience with commercial component-based software development, we noticed that restriction inheritance seems to have practical use when customizing general components to define special components.

In this paper, we have discussed our research to blend Petri net concepts and object-oriented design in order to develop a design approach for component-based software systems development. We have selected Petri nets as our underlying design model because we have experience and expertise in applying this formalism (e.g., [19][20]), and because the formalism is mature and with strong support from theory and tools. Finally, Petri nets have an intuitively appealing graphical interpretation. A unique feature of this work is the idea of a "plug-in" control structure to allow for modeling restriction inheritance.

For future work, we plan to develop some prototype tools that can be used to automate the creation of SBOPN designs for complex systems, including support features for synthesis and management of customizing general components to particular components. In addition, we plan to widen the scope of the work on inheritance modeling to include capabilities for modeling other types of inheritance.

5 References

[1] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998, ISBN 0-201-17888-5.

[2] M. Larsson and I. Crnkovic. Development Experiences of a Component-based System. *Proceedings of Engineering of Computer Based Systems (ECBS 2000)*, IEEE, 2000.

[3] R. Karl. Design Patterns for Component-Oriented Development. *Proceedings of the 25th EUROMICRO Conference*, IEEE, ISBN 0-7695-0321-7, 1999.

[4] G. Booch. *Object-Oriented Analysis and Design, with Applications* (2nd ed.). Benjamin/Cummings, San Mateo, California, 1994.

[5] B. Henderson-Sellers and J. M. Edwards. *Book two of Object-Oriented Knowledge* : The Working Object : Object-Oriented Software Engineering : Methods and Management, Prentice Hall, 1994.

[6] CORBA. http://www.corba.org.

[7] D. Rogerson. *Inside COM*, Microsoft Press, ISBN 1-5731-349-8.

[8] T. Murata. Petri Nets: Properties, Analysis, and Applications. *Proceedings of the IEEE*, April 1989, pp. 541-580.

[9] R. Bastide. Approaches in Unifying Petri Nets and the Object-Oriented Approach. *Proceedings of the 1st Workshop on Object-Oriented Programming and Models of Concurrency*, June 1995.

[10] C. A. Lakos. Pragmatic Inheritance Issues for Object Petri Nets. *Proceedings of TOOLS Pacific '95 Conference* (The 18th Technology of Object-Oriented Languages and Systems Conference), C. Mingins, R. Duke, and B. Meyer (Eds), Prentice-Hall, 1995, pp. 309-322.

[11] Y. Deng, S. K. Chang, J. C. A. Figueiredo and A. Perkusich. Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information Systems. *Proceedings of the 14th International Conference on the Application and Theory of Petri Nets*, Chicago, IL, USA pp. 203-223, June 1993.

[12] A. Newman, S. M. Shatz, and X. Xie. An Approach to Object System Modeling by State-Based Object Petri Nets. *Journal of Circuits, Systems, and Computers*, Vol. 8, No. 1, Feb. 1998, pp. 1-20.

[13] X. Xie and S. M. Shatz. An Approach to Using Formal Methods in Aspect Orientation. *Proceedings of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications* (PDPTA), (Special Session on Architectural Support for Aspect-Oriented Software Systems), Vol. 1, June 26-29, 2000, Las Vegas, Nevada, pp. 263-269.

[14] K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. *Advances in Petri Nets 1990*, G. Rozenberg (Editor), in *Lecture Notes in Computer Science*, 483, Springer-Verlag, 1990.

[15] D. Buchs and N. Guelfi. A Formal Specification Framework for Object-Oriented Distributed Systems. *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, July 2000, pp. 635-652.

[16] X. Xie. Design Support for State-Based Distributed Object Software. PhD Dissertation, EECS Department, UIC, 2000.

[17] S. Ramaswamy and K. P. Valavanis. Hierarchical Time-Extended Petri Nets (H-EPNs) Based Error Identification and Recovery for Hierarchical Systems. *IEEE Transactions on Systems, Man and Cybernetics*, Feb. 1996.

[18] A. W. Brown and K. C. Wallnau. The Current State of CBSE. *IEEE Software*, Vol. 15, No. 5, September, pp. 37-46, 1998.

[19] A. Khetarpal, S. M. Shatz, and S. Tu. Applying an Object-Based Petri Net to the Modeling of Communication Primitives for Distributed Software. *Proceedings of the High Performance Computing Conference* (HPC98), Boston, Mass., April 1998, pp. 404-409.

[20] S. M. Shatz, S. Tu, T. Murata, and S. Duri. An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 12, Dec. 1996, pp. 1307-1322.

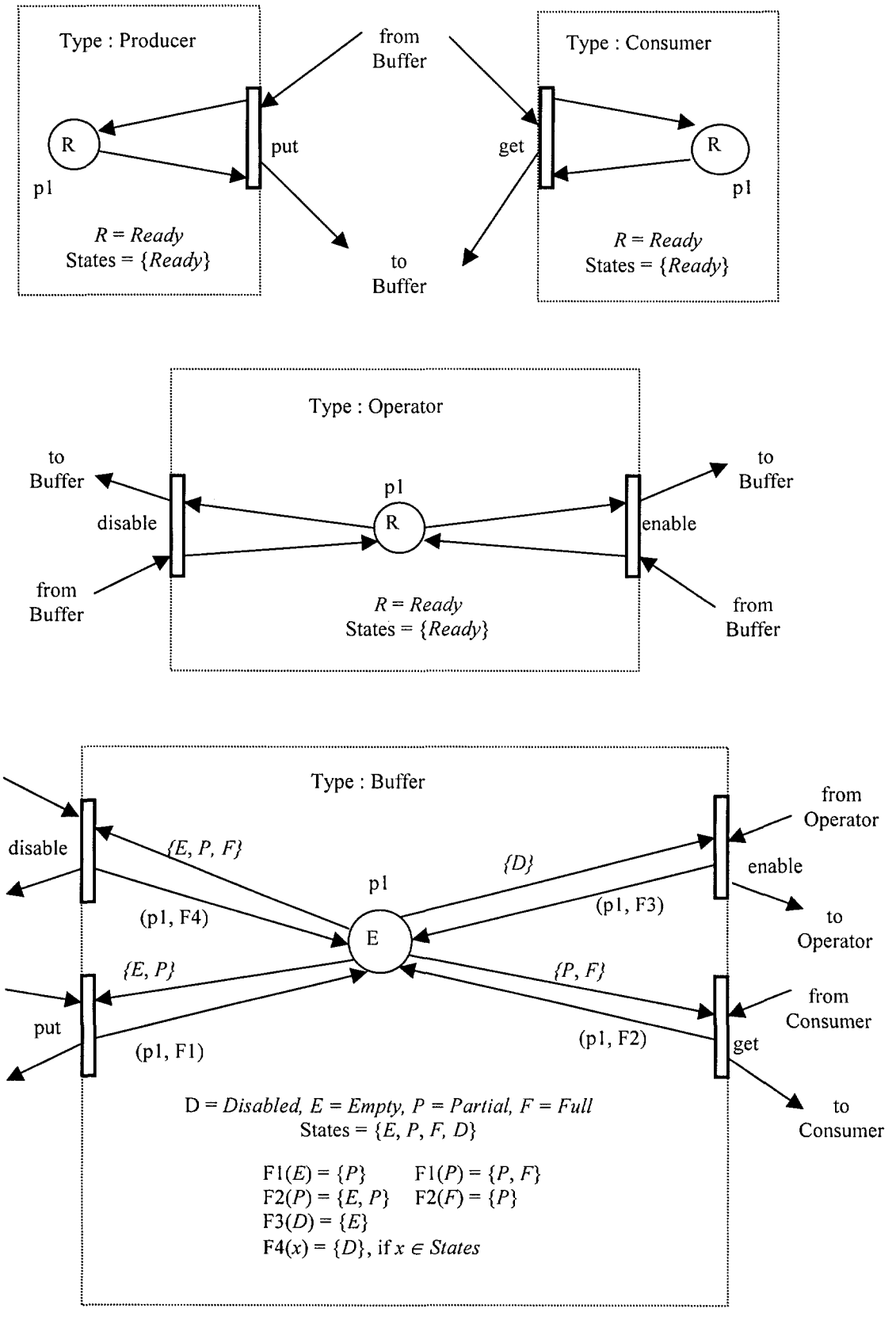


Figure 1. A SBOPN for the Buffer, Producer, Consumer, and Operator System

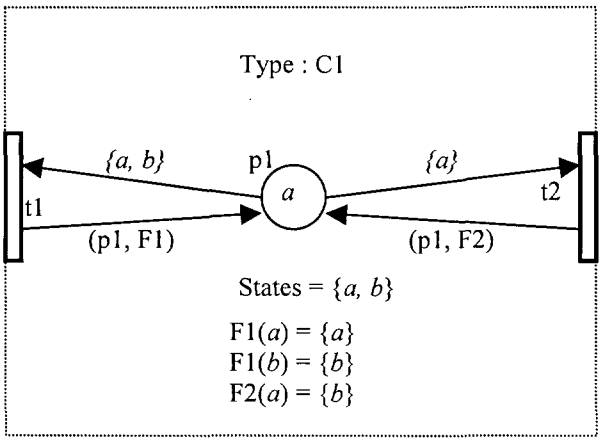


Figure 2. A SBPNO for Class *C1*

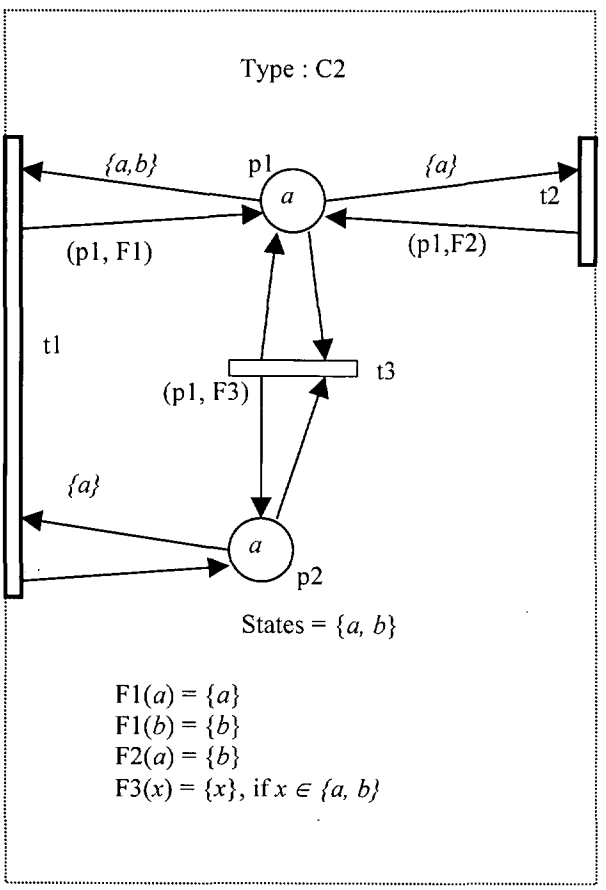


Figure 3. A SBPNO for Subclass C2 (Incomplete)

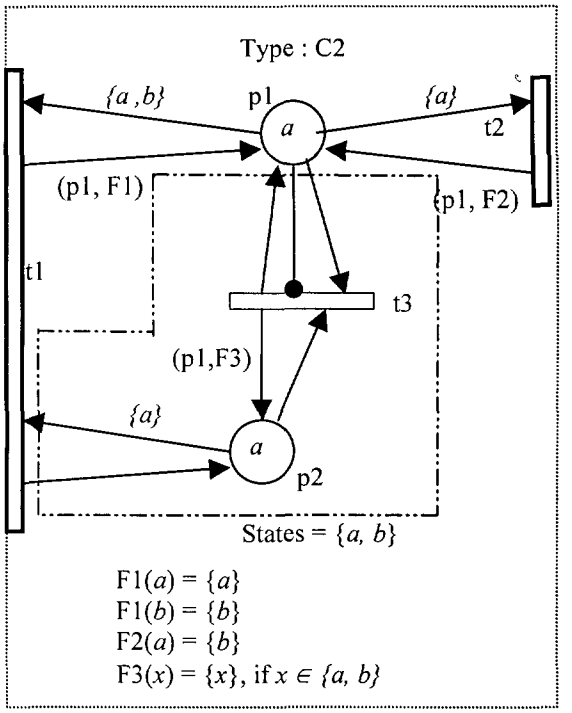


Figure 4. A SBPNO for Subclass *C2* Using a Plug-in

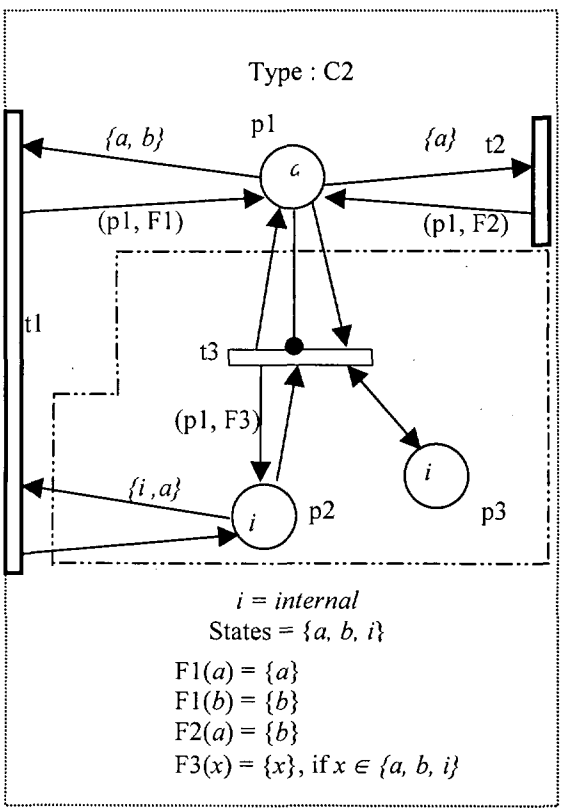


Figure 5. A SBPNO for class *C2* Using a Switchable Plug-in

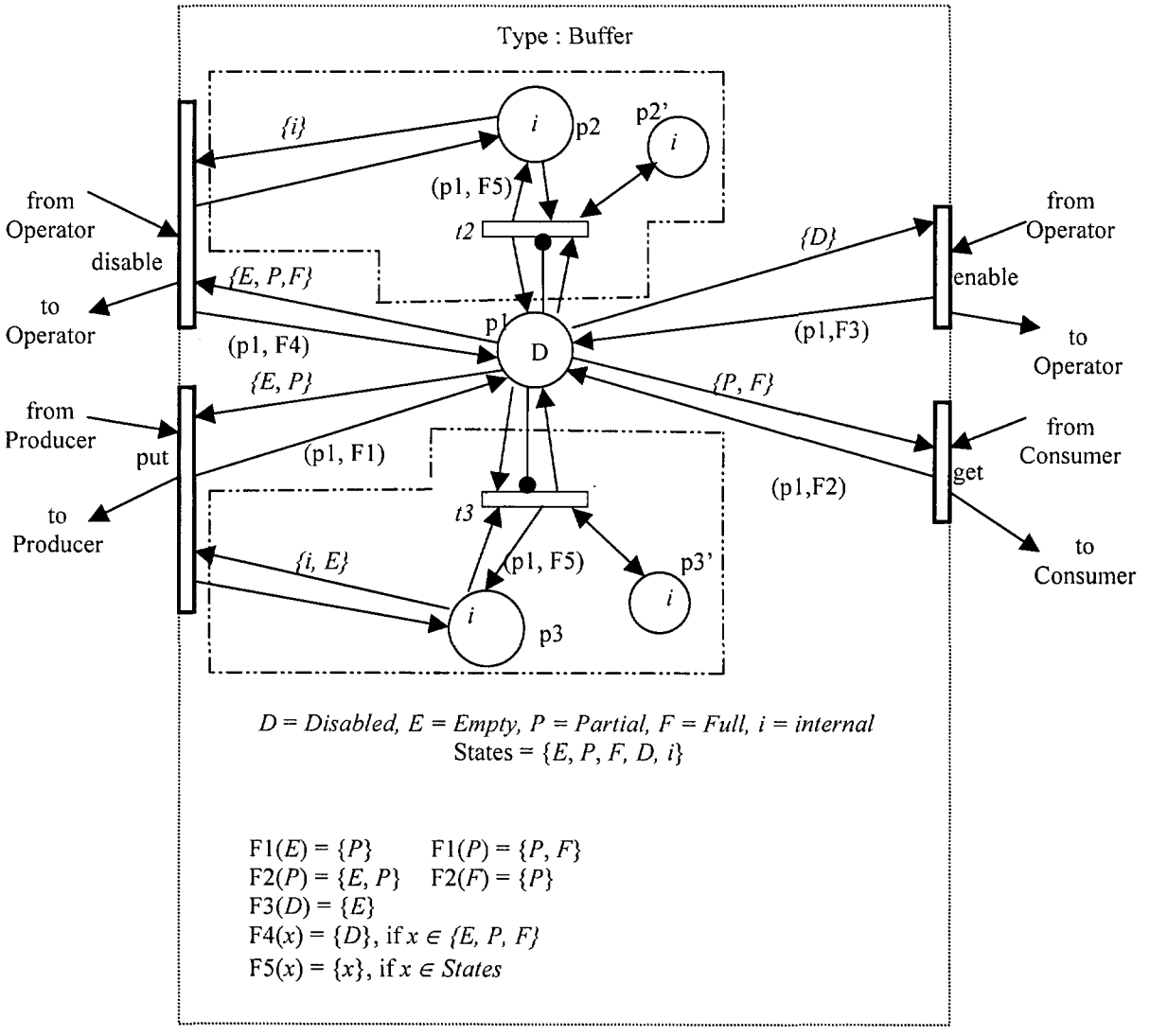


Figure 6. The SBPNO for a "Disable-Free Synchronous" Buffer Using a Switchable Plug-in

A uniform component modeling space

Duane Hybertson
The MITRE Corp., McLean, VA, USA
Phone: 703-883-7079, Fax: 703-883-1339
dhyberts@mitre.org

Keywords: component, composition, generalization, interaction, model, modeling space, representation, specification

Received: June 5, 2001

This paper presents a component modeling space as a context for supporting component-based software development and accumulating component-related knowledge. The modeling space is structured in three dimensions: A representation dimension that ranges from the languages of problem domains to computer processor languages; a composition dimension that supports a repeating pattern of the whole-part, or system-component, hierarchy; and a generalization dimension that supports reuse of models and components. Also integrated into the modeling space are an interaction model of components and connectors, an approach to component specification, and a provision for relating models via mappings. Each of these elements is characterized as applying in a uniform way throughout the modeling space.

1 Introduction

The goal of component-based software development (CBSD) is to build software systems by integrating pre-existing software components. It is understood that reaching this goal requires reusable components that interact with each other and fit into system architectures. This sounds relatively straightforward but has proved difficult to achieve.

This paper briefly discusses some of the difficulties, and then presents elements of a modeling space intended to facilitate the resolution of these difficulties. The contribution of this paper is not based on the individual elements of the modeling space, because most of them have been described elsewhere. Rather, it is based on the selection and organization of these elements into an integrated structure, and on the uniform modeling approach emphasized in this structure.

Significant issues in CBSD include:

- **Integration**: How can components developed independently be integrated into a workable system?
- **Scope**: Are components restricted to a certain scope or size? Are "objects" or "procedures" too small? Are "subsystems" too large? How can we market and use integrated collections of components?
- **Problem set**: A component is intended to be used in multiple systems. How can we develop a component to support solving multiple problems, instead of just one problem? This is the basic reuse issue.
- **Shared understanding**: How do we know what a component does, and whether it will fit into our architecture? This is the basic specification problem.
- **Semantic gap**: Software development, including CBSD, must cover the spectrum from a problem domain representation to a machine language solution. How can models bridge this gap, and what are necessary constraints on component representations?

Some of the issues listed above are not specific to CBSD. The discussion of these issues in this paper will focus on how the proposed modeling space benefits CBSD, but will also indicate broader software engineering benefits.

Goals and building blocks of the modeling space are presented in Section 2. Modeling space elements are then described in Section 3. Section 4 reviews related work, and concluding remarks are presented in Section 5.

2 Goals and building blocks

The modeling space definition is based on four goals or principles: (1) *Separation*: isolate elements important for successful long-term CBSD and define each element separately. (2) *Integration*: define the elements in a compatible way so they add up to a unified whole. (3) *Simplicity*: keep each definition as simple and uniform throughout the modeling space as possible. (4) *Universality*: find elements and definitions that apply to the full range of the component paradigm, rather than restricting the scope to any specific problem domain, life cycle phase, framework, or component model.

Several building blocks support the modeling space elements described in Section 3. These building blocks consist of three types of entities: problem domain entities, software entities, and description entities.

Problem domain entities: Elements or objects of interest in a problem domain, such as a bank account in the financial domain

Software entities:
- **Data**: Values interpreted as the state or properties of an entity or set of entities
- **Component**: Computational entity, i.e., performs operations on data
- **Port**: Point of interaction of a component with its environment, and through which a component provides or receives a service; structural part of component interface
- **Service**: Data operation(s) that may be performed by one component on behalf of another component; behavioral part of component interface
- **Connector**: Interaction entity, i.e., mediates communication and coordination among components; examples: remote procedure call, pipe, event broadcast
- **Role**: Name of behavior pattern that may be performed by a component in an interaction context; structural part of connector interface; examples: client, server
- **Protocol**: Specification of behavior pattern that may be performed by a component in an interaction context; behavioral part of connector interface
- **System**: Configuration of software entities

Description entities:
- **Model**: Explicit description of an entity or set of entities; may include entity properties
- **Specification**: Precise shared understanding of an entity or set of entities and entity properties; includes semantics
- **View**: Useful subset of an entity or set of entities

Software and description entities exist in the modeling space; problem domain entities do not. Services are defined separately from ports because services can be specified in the form of APIs that are defined separately from the components that may provide (implement) them or use them. Whether a description entity is a specification depends on the parties involved. If they share a common understanding, it is a specification.

3 Modeling space elements

The foregoing issues, goals, and building blocks led to these modeling space elements:
- An **interaction model** of components and connectors that addresses component interaction, coordination, and integration in a uniform way throughout the modeling space. This element addresses the CBSD integration issue.
- A **composition** spectrum that represents a whole-part hierarchy ranging from the most inclusive system of systems to the lowest level indivisible unit. It is recursive in that a given whole can be part of a larger whole. This element is related to the scope issue.
- A **generalization** spectrum that represents a "kind-of" or "is-a" hierarchy ranging from universal models to instance models. It is recursive in that a model that is a generalization can in turn be further generalized. This element addresses the problem set issue.
- A **specification** approach that emphasizes contracts, precision, and semantics, and has two primary specification types or views for each component and connector: *external* and *internal*. The same kinds of specification information apply throughout the modeling space. This element addresses the shared understanding and integration issues.
- A **representation** spectrum that ranges from problem domain languages to computer processor languages. This spectrum covers not only a range of representations but also a range of conceptualizations. This element is related to the semantic gap issue.
- **Mappings** that capture knowledge about the relations among models, specifications, and views throughout the modeling space. This element is related to the semantic gap issue.

Composition, generalization, and representation collectively structure the modeling space into three dimensions as shown in Figure 1. They are separate dimensions because two entities can be at the same point on any two dimensions but differ on the third. This structure is proposed in place of the traditional temporal life cycle.

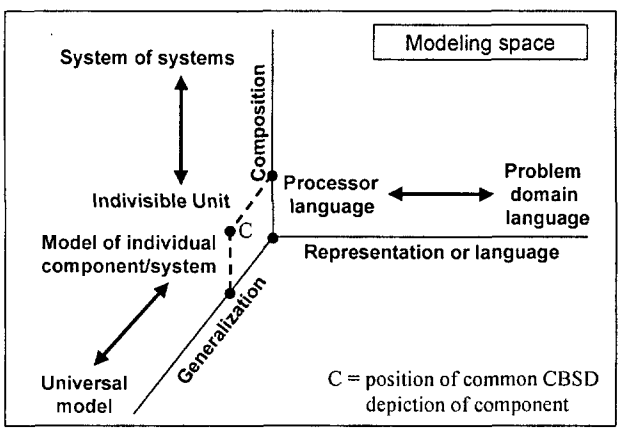


Figure 1. Modeling space dimensions

The definition of component is still a matter of debate in the CBSD community. A common view is that a component is a deployable (physical) entity that is larger than a class or object but smaller than a subsystem, and provides a specifically defined set of services but is reusable in multiple systems. This depiction corresponds to an area around point C in Figure 1, at or near the processor language end of the representation dimension and at intermediate levels of the composition and generalization dimensions. In contrast, the definition of component in this paper allows it to be anywhere in the modeling space.

Abstraction and levels of abstraction are important concepts in the modeling space, but the terms are rarely used in this paper. The reason is that four kinds of abstraction are part of the modeling space, corresponding

to the three dimensions plus views. Instead of using the general 'abstraction' term, each kind is discussed in its own context. Each dimension has a range of levels of its kind of abstraction.

A brief example will illustrate the modeling space separation of concerns. Suppose we design and build a financial system for First National Bank (see Figure 2). The system interacts with customers in setting up and using accounts, and sends certain reports to the Federal Reserve Bank (FRB) on a periodic basis. The system consists of hardware, software, and manual operations performed by customer service representatives. The software portion of the system consists of customer management and account management. Account management is composed of two parts: accounts and account transfers. In the course of developing this system, we specialize the party management facility defined by OMG for our customer management need, and then as we move into implementation, we find and incorporate a customer management component that satisfies our requirements for that part of the system. We develop the account management part, but when we are done, we decide this is a general capability that multiple banking systems could use. We generalize account management and make it available as a component with accompanying specification.

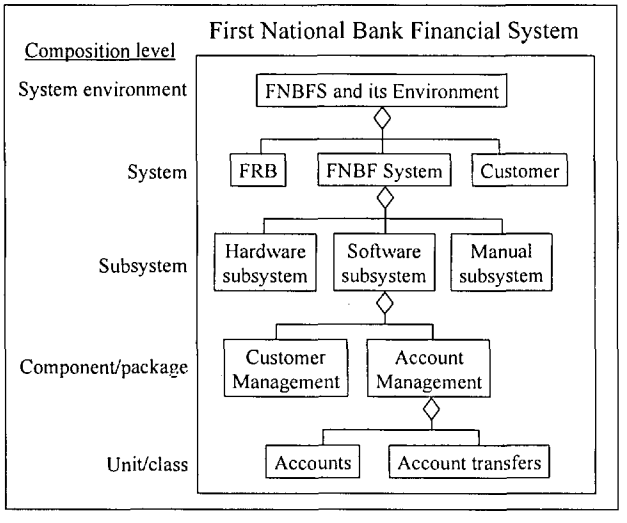


Figure 2. Example system

We will now briefly map the example to the modeling space dimensions. Figure 2 shows the *Composition* perspective. Using familiar terms for each level, the composition levels range from the system environment to the unit or class level. In the *Representation* dimension, models range from use cases expressed in English (on the problem domain end) to machine instructions expressed in binary on the machine end. Between these end points are design and implementation models in UML and Java. The Java bytecode is very close to the machine code in this spectrum. In the *Generalization* dimension, the focus of the example is on a single system, which places it at the specific end of the dimension. However, there are a few generalized elements. The party management facility was a general model that we specialized for customer management, and the more specialized customer management was general enough to find an existing component to satisfy the need. We also generalized account management into a reusable component.

Each modeling space element in the list above will now be described in additional detail.

3.1 Interaction model

This section describes an interaction model of components and connectors that addresses the CBSD integration issue. At its most basic level, the component paradigm is about developing components and integrating them into systems in which the components interact. The interaction model supports the modeling of component interaction with two entity types: **components**, which serve as a locus of computation and decision-making, and **connectors**, which serve as a locus of interaction between components. Both entity types exist throughout the modeling space in all dimensions. Every box shown in the hierarchy in Figure 2 can be a component. Correspondingly, connectors define and facilitate interactions ranging from a procedure call or message passing to UNIX pipe-filter interactions to distributed system interactions. As a locus of interaction, a connector provides not just an exchange medium, but also specification of interaction roles and protocols.

Components and connectors have respective interface points called ports and roles, as shown in Figure 3. The left side of the figure shows a basic interaction of components A and B via a connector. The center is a visualization of the component-port-role-connector model. The right side shows specification elements in this structure (see Section 3.4). A key point is that *roles that a component plays in an environment are defined not by the component, but by the connector-specified interactions in which the component participates.*

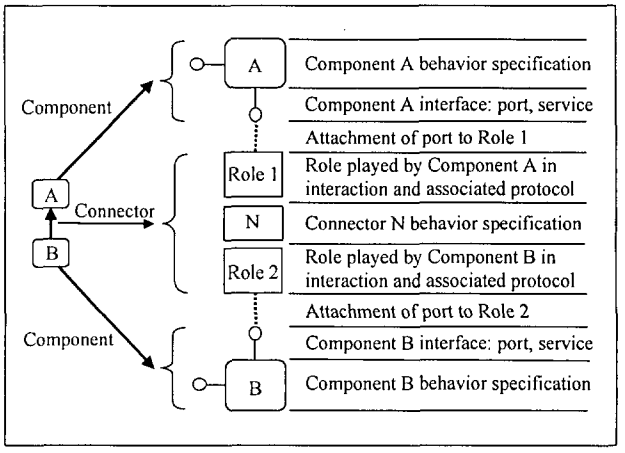


Figure 3. Anatomy of an interaction

The interaction model supports CBSD in two ways. First, the explicit treatment of interaction, connectors, and

coordination provides a basis for integrating components into systems by clearly defining the integration context. Second, the uniform nature of this model throughout the modeling space facilitates component modeling and the use of the component paradigm throughout the full spectrum from problem definition to deployment. Both of these benefits will become clearer in the ensuing discussion of the remaining modeling space elements.

3.2 Composition spectrum

This section describes a composition spectrum that addresses the CBSD scope issue. Software systems and components typically exhibit a whole-part hierarchy. The end points of this spectrum are the smallest component or unit that is not further divided and the most inclusive component or system of systems. Formally, system and component are synonyms. Informally, they can be used as relative terms. A system at one level may be a component of another system at the next higher level, and the same relations repeat at each level. Figure 4 illustrates the repeating pattern. The Figure 2 example shows the pattern repeated four times.

'A' represents a system of components B, C, D, and E interacting via connectors K, L, M, and N. 'E' in turn represents a system of its interacting components and connectors F, P, etc.. The figure shows a component as a composition of components and connectors. A connector can also be a composition of connectors and components. One mapping of the applicable parts of Figure 4 to the example in Figure 2 could be: E = FNBS system, D = FRB, N = asynchronous interprocess connector, C = customer, M = human-computer interaction, F = software subsystem, G = hardware subsystem, H = manual subsystem. Another mapping could be: E = account management, D = customer management, F = accounts, G = account transfers, P = message passing connector.

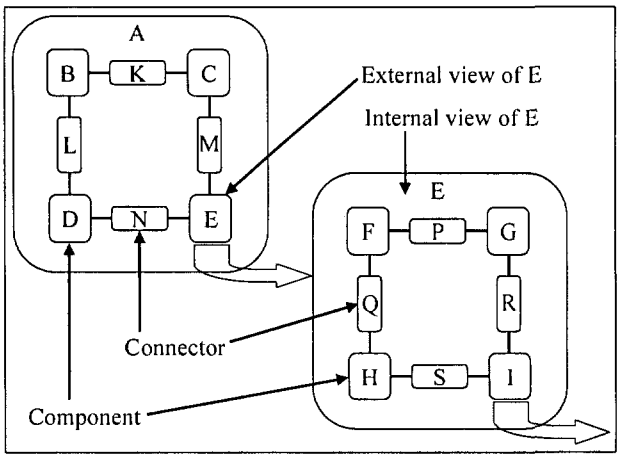


Figure 4. Recursive composition pattern

This spectrum provides a clear context for internal and external specifications (discussed in Section 3.4), and offers a uniform way to treat components at multiple levels. From the Figure 2 example, account management could be a CBSD component in other systems in addition to the FNBF software system. The FNBF software system, since both of its components are generalized, could itself be a CBSD component in other financial systems. At each level, the internal view of each component is seen by the component developer, but is hidden from the system composer. The person producing E is a system composer when integrating I into E, but is a component developer when preparing E for users such as the system composer of A.

3.3 Generalization spectrum

This section describes a generalization spectrum that addresses the CBSD problem set issue. A key potential benefit of the component paradigm is reuse—a component is intended to be usable in multiple systems. The generalization spectrum supports this goal with the idea of general models and specifications. Generalization is a form of abstraction in which information is removed to make a more general component or model that is useful in multiple environments or that allows multiple implementations. General models include abstract data types, classes in class hierarchies, generics, templates, component and connector types, frameworks, reference models, domain specific architectures, product line architectures, analysis patterns, architecture/design patterns, architecture styles, and programming idioms.

Each of these is aimed at a goal that is difficult to achieve: solve a *group of problems* rather than a single problem. If we characterize a problem as a set of features or aspects, then the union of problem sets yields a problem space, and the intersection of the problem sets defines the features that are common among the problems in that space. The difference between the union and the intersection represents variation among the problems. If the intersection is small, the problem space is heterogeneous. The class of problem domains supported by software engineering is an important example of a heterogeneous problem space. If the intersection is large, the problem space is homogeneous. The class of hardware processors is an important example of a relatively homogeneous problem space.

In any problem space, we can identify subspaces that are more homogeneous than the complete space, and increase reuse in that subspace. Domain specific engineering is targeted to a homogeneous subspace of the overall problem space. There is a general tradeoff. We can achieve limited reuse across the whole set of problems, or we can achieve greater reuse within a more homogeneous subset of problems.

The modeling space approach to this tradeoff is a principle we will call maximum ***leverage***. Leverage of a solution (e.g., a model or component) is defined as the degree to which it satisfies these two conflicting criteria: (1) number of problem situations to which it applies; and (2) proportion of solution it provides—i.e., extent to

which it provides the complete solution needed for the applicable problem set. Leverage as a metric is the product of these two criteria. This makes the tradeoff explicit. The concept is illustrated in Figure 5.

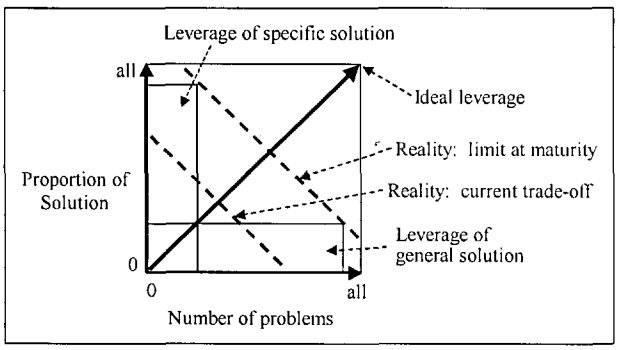


Figure 5. Leverage

Ideally, one solution would completely solve all problems (upper right corner of box). However, there is a limit even when a discipline reaches maturity—the tradeoff still exists. In an immature problem domain, the limit is not yet reached, and leverage is even more restricted. The current and ultimate tradeoffs are represented by the diagonal dashed lines in Figure 5. Two models of equal leverage may differ in that one may provide a small part of the solution for a large number of problems (shown as the box labeled "Leverage of general solution" in Figure 5), while the second may provide most of the solution for a small number of problems (shown as "Leverage of specific solution"). The first criterion reflects the perspective of the person with the problem—the consumer or client: I want a solution that completely solves my specific problem. The second criterion reflects the perspective of the person with the solution—the producer or provider: I have a solution that will help solve everyone's problems.

Achieving leverage is critical to CBSD in terms of market viability. A component developer must produce a component that simultaneously satisfies enough of the specific needs ('proportion of solution') of a sufficient number ('number of problems') of individual system composers, to establish a market.

Generalization and the techniques for increasing leverage support CBSD and component reuse. Leverage will increase as CBSD and software engineering in general mature. But how can we increase leverage in the meantime? Within a given problem space, one can go beyond what is common and also capture some of the variability of a set of problems. We might call this predefined variability. A simple example of this is parameterization. Adding a parameter to a component interface increases the variability it can accommodate. Another example is a general model that defines a "component product line" from which multiple component variants can be instantiated. A third example is the interaction model described in Section 3.1, which has been specialized into architecture styles such as object-oriented, pipe-and-filter, event-based, and blackboard systems [12]. Each style defines component and connector types for a class of systems.

3.4 Specification

This section describes a specification approach that addresses the CBSD shared understanding and integration issues. A specification is a precise shared understanding of an entity or set of entities, as defined earlier. This means that a specification involves an entity such as a model and at least two parties communicating about the model. Typically one party writes the model (e.g., a programmer), and the other party reads the model (e.g., a compiler). The two parties must understand the language used to represent the model. If the two parties share the underlying concepts or semantics of the model, much of the specification can be implicit. If the parties do not share these concepts, more of the specification must be explicit. In a mature discipline, small models are sufficient to represent specifications, because most of the shared information is implicit.

The modeling space approach to specification emphasizes the basic principles of modularity, encapsulation, and precision. A specification consists of a set of rules, where 'rule' is used in a very general sense that includes everything from system requirements to code. Examples of types of rules: required data types; required functions; performance properties; provided services; dependencies; policies; types of permitted components in a system; specific components and connectors in a system; attachment of components to connectors (ports to roles); required properties or attribute values; invariants, preconditions, and postconditions; exception handling; state transitions.

An important element of component specification in the modeling space is design by contract, as defined by Meyer [9] but extended to include non-functional (e.g., performance, quality of service/QoS) information.

Specification types are derived from the interaction model and the composition dimension—specifically, the intertwining of internal and external views. An *internal specification* of a composite component or connector entity is a set of rules—including policies—about the data, components, and connectors that are within the composite, and their structure and interaction. An *external specification* of a component or connector entity is a set of rules about the external view of that entity. For a component, that includes observable data, behavior, ports, and services. The relation between the two specification types (shown in Figures 3 and 4) is that an internal specification of a composite includes the external specifications of its components and connectors. The external view corresponds to what we typically call requirements, and the internal view corresponds to what we typically call architecture, design, or implementation.

An external component specification has two contract types. The first is a user specification, which specifies what the component provides to users (services offered). The second is a provider specification, which specifies what it requires of providers (dependencies). Note that this pattern can set up a dependency chain of indefinite length, in which a component can be a provider or server to one component and a user-of or client-to another component.

The interaction model in Figure 3 will now be described further, in terms of informal specification examples. Suppose connector N is a function call, A is a procedure, and B is a square root function. Role 1 is caller, and its associated protocol is as follows. It decides to initiate a call, which involves transferring data and control, and then it waits for a return, which involves receiving data and control. Role 2 is "callee" or server function, and its associated protocol is: It waits for a call, which involves receiving data and control, and then it initiates a return, which involves transferring data and control. The connector N behavior specification is: It receives a call at the caller role and initiates a transfer of this call to the server function role; then it receives a return at the server function role and initiates a transfer of this return to the caller role. Each receipt and transfer of a call or return includes a transfer of data and control. Thus, connector N blocks control at the caller role from the time it receives a call to the time it transfers a return. The Component B behavior specification is: Precondition: Received data X ≥ 0. Postcondition: Returned data Y $=\sqrt{X}$ within tolerance T and time delta D. B receives control and a data value X. It then returns control and a data value Y.

Most of these details of a function call interaction specification are usually implicit, because function call is a mature connector type and we have a shared understanding of it. However, more details of complex or higher-level interaction specifications need to be explicit to avoid component mismatch.

The specification pattern of relating external specifications to a larger internal specification addresses the CBSD problem of fitting a component into a system. The inclusion of connectors, along with the modeling space approach to component and connector specifications, defines this problem in a precise way and helps determine if a component matches a system or will interoperate with other components. Specifically, suppose that C is an available component, and S is a composite component or system that could potentially use C. That is, the internal specification of S includes an external specification of a needed component we will call SC, and we want to determine if available component C satisfies the needed SC. (In Figure 2, S = FNBS software subsystem and C = customer management component.) The determination is based on a comparison of the external specifications of C and SC. From a contract perspective, we can say that C satisfies SC if and only if these two conditions hold: C provides *at least* all the services that SC provides, and C requires *at most* all the services that SC requires ($C_{prov} \geq SC_{prov} \wedge C_{req} \leq SC_{req}$).

To be able to determine this, however, both the system and the component need to be adequately specified. Most current programming languages lack support for this specification approach in the areas of semantics and external specification of required services.

3.5 Representation spectrum

This section describes a representation spectrum that addresses the CBSD semantic gap issue. The artifacts of software engineering have traditional names such as requirements specification, architecture description, design description, and code. In the modeling space, all these are regarded as models of one or more software entities such as system or component. Each model is represented in some notation or language, or combination of languages. The general categories of languages are textual, graphical, and mathematical. The representation spectrum ranges from problem domain models (such as banking or geospatial information) to computer processor models. Corresponding to the language differences are differences in concepts, terms, and domain ontologies. It is really the latter set of differences that establishes the large conceptual gap between problem domains and computer processors, and defines the range of this spectrum. Example: In the banking domain used in the earlier example, key concepts are account, withdraw, deposit, balance, and transfer. In the geospatial domain, key concepts are map, contour, elevation, feature, thematic layer, and projection. In the computer processor domain, key concepts are load, store, add, branch, memory address, and register (actually 01011000, 0101000, etc. but we will use translated terms). The computer processor uses this basic set of concepts to solve problems in banking, geospatial information, and all other problem domains. Note that we listed the concepts in all these domains using English, but the conceptual distance between them remains large.

The relation between models in the representation dimension is translation from one representation to another—for example, problem domain notation to formal specification to UML to Java to machine language. Note that a translation may be combined with relations in other dimensions. In Figure 4, assume that the internal view of A and external view of E are represented in UML, while the next composition level—the internal view of E—is represented in Java. In this example, the respective models of the external and internal views of E have two relations: translation in the representation dimension, and composition in the composition dimension.

The representation spectrum brings into focus several CBSD issues related to language, notation, terminology, and semantics. One issue is sufficiency. Is a specific language sufficient to express the necessary specification

information (described in Section 3.4)? As indicated earlier, most current programming languages are not sufficient in this regard. Further investigation of the use of declarative languages for external specifications may be useful.

Another representation issue is how to determine whether the specification of an available component satisfies the specification of a needed component if the two specifications are in different languages. Do we need to try to adopt a common external specification language—e.g., a formal language, or UML, or IDL, or XML, or natural language? How do we deal with differing ontologies or paradigms, such as procedural versus object-oriented versus functional? An example of recent research is an approach to component search in the context of differing ontologies [4]. The representation spectrum does not resolve the issues, but it does provide a focal point for addressing representation and the semantic gap issue separately from other CBSD issues.

3.6 Mappings

This section briefly describes mappings that capture knowledge about the relations among entities throughout the modeling space. Mappings address the CBSD semantic gap issue. The knowledge to be captured in the modeling space includes not only a large number of reusable models, but also reusable mappings among the models. Mappings are commonly used relations that tie together existing models throughout the modeling space, and help navigate the space when solving a specific problem. Relations include composition, decomposition, generalization, specialization, translation, optimization, and view. For example, suppose we start with a given problem that matches a general model near the problem end of the representation spectrum. A translation mapping might lead us to a model represented as a formal external system specification. A decomposition and translation mapping might lead us to a model representing interacting components of the system in UML. We may then go to our component catalog and match our needs (the specified components of our system) with the specifications of available components, and pick a set that matches. The catalog may exist in the modeling space in the form of external component specifications that provide purchasing or leasing information for associated deployable components.

4 Related work

The generalized view of components and connectors is consistent with software architecture literature, which has promoted connectors as first-class entities [12, 1]. The Real-Time Object-Oriented Modeling approach [11] shares some of these features, and its composition approach is recursive and hence more compatible with the modeling space composition dimension than are most object-oriented treatments.

The connector, as a locus of interaction and coordination, is consistent with literature on coordination models and languages. This literature recognizes coordination as distinct from computation and as a subject of study in its own right [6, 10], and it also addresses the issues of heterogeneous systems. A preliminary taxonomy of connectors is proposed in [8]. Taxonomies and classification schemes are important steps toward reducing artificial variability and accumulating a body of knowledge.

The product-line approach and domain engineering [2, 15] exploit extensive commonality within a homogeneous problem class, which positions both in the generalization dimension. The KobrA approach [3] is an example of the product line approach. KobrA also has other similarities with the modeling space elements presented here. The KobrA framework captures what is common and also captures "concrete variants" (predetermined variability). The dimensions that embody separation of concerns are in partial agreement with the modeling space dimensions. The primary differences between the two approaches are (1) greater emphasis on interaction and connectors in the modeling space, and (2) the modeling space representation dimension as opposed to the development process emphasis in KobrA.

Szyperski's approach to component specification [14] is consistent with the approach in this paper. Szyperski also discusses the specific "wiring standards" defined in three primary approaches to component software: CORBA, JavaBeans, and Microsoft's COM/DCOM. However, his treatment does not provide a general approach to connectors or interaction. The new CORBA component model (CCM), described in [13], is a generalized and extended form of Enterprise JavaBeans or Java 2 Enterprise Edition. The CCM is consistent with a number of elements in the modeling space, including specification contracts and modularity. The concept of container has some of the mediation features of a connector, but is more specialized for the CCM environment. CCM appears to be focused on the programming region of the representation spectrum rather than the full spectrum.

Catalysis [5] is an approach to objects, components, and frameworks that emphasizes connectors as well as components and covers a significant part of the modeling space. Catalysis uses the concept of *object* as the locus of static functionality and data, and *action* as the locus of dynamic activity. It supports composition of both objects and actions. Generalization is supported via model frameworks.

RM-ODP [7], an ISO standard for distributed processing systems, has a number of similarities with the modeling space. Many of the foundation concepts, such as encapsulation, interface, and contract, are compatible. The RM-ODP architecture concepts include a list of distribution transparencies, which maps to the

generalization dimension. RM-ODP presents five viewpoints of a distributed system: enterprise, information, computational, engineering, and technology. The information view maps to data specification in the modeling space. The other four viewpoints all map to some degree to an internal component specification, at different levels of composition and generalization.

What the modeling space adds to this related work is a broad context in which these various approaches can be positioned and compared. The modeling space also adds a structure for compiling and organizing models that describe components, their interactions, and the larger configurations into which they can be integrated.

5 Conclusion

Benefits. The modeling space described in this paper supports CBSD and component modeling, both in the near term and the long term. In the near term, it provides a uniform structure for modeling components and modeling systems in which the components may be integrated. Modeling the systems supports the system composers. Modeling the components supports both the component developers and the system composers.

In the long term, the uniform structure can serve as the basis for an organized repository of knowledge of components and systems in which they can be used. This knowledge will be in the form of a large number of well-understood models that will exist throughout all dimensions of the modeling space, and relations or mappings among the models.

In addition, the modeling space elements apply to software engineering in general, not just to CBSD. Many large systems require a combination of the component paradigm and other approaches such as custom development. The modeling space defined in this paper can reconcile these approaches.

Validation. The modeling space approach described in this paper has not yet been directly validated in CBSD practice. However, the approach represents a consolidation of elements with a solid foundation in software and systems engineering practice. Conceptualizing software engineering as modeling is fairly well established. Generalization and composition are well established in software engineering and also have a long tradition in other disciplines such as ontology, biology, and mathematics. Composition and representation have long been the primary elements of the software life cycle. Thus the argument for the validity of the modeling space at this point is based on the pedigree of its elements. Further work in direct CBSD validation is anticipated.

Acknowledgments

The MITRE Corporation provided support for the research reported in this paper.

6 References

[1] Allen R. & Garlan D. (1997) A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering & Methodology*, 6, 3, p. 213-249.

[2] Arango G. & Prieto-Diaz R. (1991) *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press.

[3] Atkinson C., Bayer J., Laitenberger O. & Zettel J. (2000) Component-Based Software Engineering: The KobrA Approach. 2000 Int. Workshop on CBSE, Limerick, Ireland. Available at http://www.sei.cmu.edu/cbs/cbse2000/papers/21/21.html

[4] Braga R., Mattoso M. & Werner C. (2001) The Use of Mediation and Ontology Technologies for Software Component Information Retrieval. 2001 Symposium on Software Reusability, Toronto, Canada, May 18-20, 2001. Published in *Software Engineering Notes*, 26, 3, p. 19-28.

[5] D'Souza D. & Wills A. (1998) *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley.

[6] Gelernter D. & Carriero N. (1992) Coordination languages and their significance. *Communications of the ACM*, 35, 2, p. 97-107.

[7] International Organization for Standardization (1995) *Basic Reference Model of Open Distributed Processing*. ITU-T Recommendation X.902 | ISO/IEC 10746-2: *Foundations* and ITU-T Recommendation X.903 | ISO/IEC 10746-3: *Architecture*.

[8] Mehta N., Medvidovic N. & Phadke S. (2000) Towards a taxonomy of software connectors. *Proceedings of the 22nd Int. Conference on Software Engineering*, Limerick, Ireland, p. 178-187.

[9] Meyer B. (1997) *Object-Oriented Software Construction* (2nd ed.) Prentice Hall.

[10] Papadopoulos G. & Arbab F. (1998) Coordination Models and Languages. CWI Report SEN-R9834. Available at: http://citeseer.nj.nec.com/papadopoulos98coordination.html

[11] Selic B., Gullekson G. & Ward P. (1994) *Real-Time Object-Oriented Modeling*. John Wiley.

[12] Shaw M. & Garlan D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall.

[13] Siegel J. (2001) *Quick CORBA 3*. John Wiley.

[14] Szyperski C. (1998) *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

[15] Weiss D. & Lai C. T. R. (1999) *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley.

An agent-based component platform for dynamically adaptable distributed environments

Rainer Weinreich and Reinhold Plösch
Software Engineering Group, Dept. of Business Informatics,
Johannes Kepler Universität Linz, Austria
rainer.weinreich@jku.at, reinhold.ploesch@jku.at

Keywords: component models, distributed components, mobile agents, deployment, remote configuration, component frameworks

Received: June 16, 2001

Component-based systems require standardization by component models and component platforms providing both an execution environment for software components and core component services. Remote administration and mobile computing require additional support from component platforms. We argue that increased flexibility, especially at run-time, can be achieved by using agent technology and agent platforms as powerful component environments. We present an adaptable component platform which incorporates mobile agent platforms and describe how important issues of component deployment, configuration and security are supported by our environment.

1 Introduction

Current software systems are increasingly assembled from reusable software components, written at different times by various companies and developers. Software components are units of independent deployment and composition [1][2] and conform to a component model. A component model provides standards for component implementation, naming, interoperability, customization, composition, evolution, packaging, and deployment [3]. The component model is not only the basis for the development of reusable components, but also for constructing execution environments and basic services for components conforming to a particular component model. We use the term *component platform* to denote an execution environment and basic component services for software components.

As the Internet gains steadily in importance and wireless computing and mobile devices penetrate all areas of business and private life, additional demands on component models and platforms are raised. This includes remote administration via Internet connections, support of different end-user devices, dynamic configuration, dynamic adaptation to different environmental conditions, security, and interoperation among components from different component models and platforms.

Many of these demands are supported by software agent technology [4]. Currently there is no generally accepted definition of a software agent. The basic idea underlying most definitions is the vision of intelligent, sometimes mobile, programs that are able to act autonomously on behalf of a user. Distinguishing agent features that are often mentioned are autonomy, intelligence, mobility, personality, adaptability, knowledge and cooperation (e.g., [5] [7] [8]). Petrie [10] calls this an anthropomorphic view, because human cognitive traits like environmental awareness, autonomy, and intelligence are ascribed to software. Agents are used for information retrieval [11], network management [12][13], telecommunication [14][15] and E-commerce [9][16].

From a technical perspective, agent-based systems have similar characteristics as component-based environments. Issues like naming, interoperability, customization, evolution, and packaging are equally important in agent-based systems and have to be supported by agent platforms and development environments. In addition, agent technology emphasizes support for heterogeneity, adaptation to different environments, code mobility, and collaboration. Thus, software agents can be viewed as flexible and adaptable software components. Sometimes they are even called next-generation components [5][6].

We use agent technology as the basis for an adaptable component platform supporting deployment, configuration, and remote access of components for monitoring and information retrieval in heterogeneous distributed environments. The system is called *Insight ACS* and is currently used as the basis for remote administration and control of process automation systems over Internet connections. Main points of our environment are dynamic services, mobility support, native-code management, dynamic configuration, and

multi-protocol remote access of various types of components.

The remainder of this paper is structured as follows: In the next section we outline the application domain, which gives an impression of the various demands on our component platform. In Sections 3 and 4 we give an overview of the system structure and the platform's architecture. We will show that an agent platform is a major part of our environment, though not all components in our system are agents. In Sections 5-9 we concentrate on issues like deployment, configuration, mobility support, service management, security and domain-specific component frameworks. In Section 10 we describe related work.

2 Background and Application Domain

The environment presented in this paper is the result of a research cooperation with Siemens Germany and is currently used for remote system diagnosis and remote supervision and control of steel plant automation systems.

Remote system diagnosis is a long-term process. Data about certain aspects of an automation system is continuously collected, analyzed, and compiled to reports. These reports are made available to remote plant supervisors, which may identify upcoming problems and timely take adequate actions.

Remote system supervision and control supports monitoring of individual critical aspects of an automation system. If problems like exceeded quality limits are detected, autonomous and configurable actions may be performed by the supervising software. Typical types of actions are automatic notification of plant supervisors, messages to local operators or autonomous changes of parameters for temperature models or geometry models.

The architecture of our system and many of its features were influenced by its application in this domain. Automation systems are heterogeneous distributed systems consisting of multiple hosts with probably different operating systems within a local area network. Components for collecting data during system diagnosis and for system supervision and control need to be installed and configured remotely and dynamically. Remotely means that components may be installed and configured over Internet connections. Dynamically refers to the ability to perform these tasks at run-time. Components have to move within the system to collect information, have to adapt to different environmental conditions, must be able to communicate with legacy systems, and need access to native operating system services.

Some of these characteristics, like distribution, heterogeneity, dynamic installation, mobility and dynamic adaptability, are especially supported by agent technology. Other requirements like remote configuration via temporary Internet connections, thin clients, and interaction with legacy systems have additionally to be considered in system architecture and design.

3 System Structure and Overview

The two main parts of the structure of a system supported by our environment are depicted in Figure 1. The left part of the figure shows the elements of the system for remote administration and configuration. The right part depicts the target environment for supervision and retrieval components. Both are connected through the Internet, although they may be co-located within the same LAN or Intranet, also.

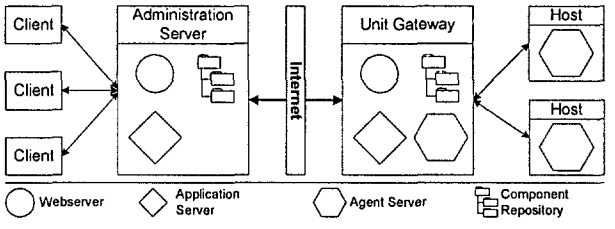


Figure 1: System Structure

The structure of the administration subsystem reflects specific requirements like thin clients, central administration management, and security. An Administration Server hosts component repositories for various target environments, called units in our terminology. Each repository contains various kinds of components for supervision, information retrieval and other tasks to be performed at a specific target environment. We use the term *domain components* for such components to distinguish them from *system components*, implementing functionality and services of the component platform itself.

Components within a repository are organized into different categories. As we will outline in the following sections, domain components are implemented as portable mobile agents in our system, where portable means that they are independent from a specific agent platform. The Administration Server further hosts a Web Server and an Application Server containing tools for installation and configuration as well as security services for authentication and authorization. Administration tasks are performed from thin clients. This means that tools need not be pre-installed at administration hosts. Instead, they are loaded dynamically from the Administration Server and can be updated at a central location (see Section 6 on dynamic configuration). The user interfaces of theses tools are decoupled from their application logic. Currently, we support pure HTML-based interfaces as well as more elaborate interfaces based on Java GUI libraries. The system structure is also influenced by security issues. For example, remote administration tasks are always performed via a

connection between the Administration Server and the Gateway Server at the target platform. We will discuss security issues in more detail in Section 8.

The target environment (or unit) for components is a heterogeneous distributed system, which consists of an arbitrary number of hosts typically within a local area network. The hosts may have different operating systems, but each host has to provide an Agent Server, which acts as run-time environment for both component services and domain components. A distinguished host, the Unit Gateway, is used as the entry point for accessing the unit from the Internet. For security reasons, administration clients may only connect to the Unit Gateway (via the Administration Server). In addition to an Agent Server, the Unit Gateway hosts an Application Server, which contains components for remote administration and access of the domain components within the unit. Domain components are either installed directly on a specific host or they are installed at the Unit Gateway and deploy themselves to the appropriate host depending on environmental conditions.

4 Software Architecture

Figures 2 and 3 give an overview of the main architectural building blocks of the Insight component environment at the target unit. Figure 2 shows the main layers of the component platform. Figure 3 depicts main elements of the Remote Administration and Access Interface (RAAI).

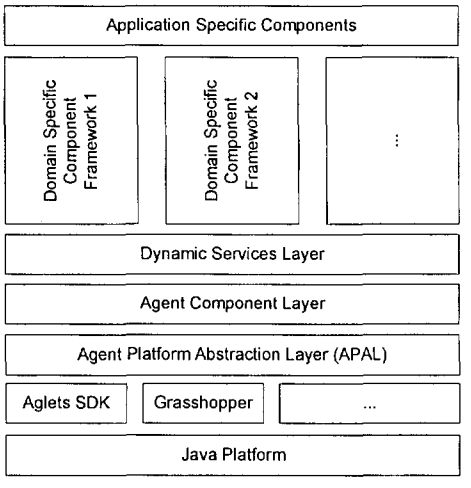


Figure 2: Platform Architecture

Since the target environment is a heterogeneous distributed system, portability among various hosts is important. For this reason, the Insight environment is based on the Java 2 platform, as shown by the most general layer in Figure 2. Java 2 offers a certain degree of portability due to a platform neutral binary format (the Java bytecode) and a large set of standardized core APIs. However, our system is not based on a Java component model like JavaBeans or Enterprise JavaBeans. Instead, domain components and most of the system components are implemented as mobile agents.

Mobile agents [17] are agents that can migrate from host to host in a network. From a run-time point of view they are active objects which may transfer their state, their code and (sometimes) their current execution stack upon migration to another host. From a component-based point of view they may deploy themselves (code + data) to a new target environment. The run-time environment for mobile agents is provided by agent platforms. An agent platform defines facilities for creating and destroying agents, for mobility, for agent communication, and for other platform-specific services also found in many component platforms. The component model for agents is also implicitly defined by the agent platform, though platform independent standards like FIPA (www.fipa.org) are emerging.

We use the facilities provided by existing agent platforms as the basis for our component platform (see Layer 2 in Figure 2). However, our environment is not based on a specific agent platform, but on an *Agent Platform Abstraction Layer* (APAL) defining platform-independent abstractions for agent creation, disposal, communication and migration. Currently, we provide two implementations for the APAL, one for the Aglets SDK (www.aglets.org) and one for Grasshopper (www.grasshopper.de).

The component model of our platform is defined by the *Agent Component Layer*, which is based on the APAL. It extends the primitives of the APAL by a high-level communication API, a mechanism for component aggregation and data types for component identification and component metadata.

The *Dynamic Services Layer* contains various general services, including a trading (directory) service, an event service, and a native-code management service. All services are implemented as portable mobile agents, also termed system components in our environment. Due to their nature these general services can be installed and uninstalled remotely, which enables a dynamic upgrade of the component platform itself.

On top of the Dynamic Services Layer are domain-specific component frameworks for diagnosis and supervision tasks. They contain different coordination and communication models for domain components, i.e., agents performing supervision and data retrieval tasks.

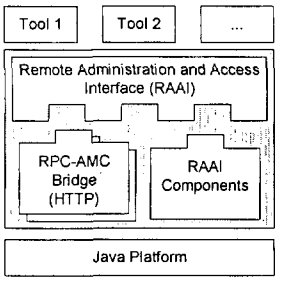


Figure 3: Remote Administration and Access Interface (RAAI)

The agents within a unit use the communication services of the underlying agent platform for communication. The Remote Administration and Access Interface (RAAI) depicted in Figure 3 can be used for accessing agents from outside the unit. The RAAI is installed in an Application Server at the Unit Gateway. It contains components providing information about the whole unit and for bridging access to individual agents. Currently, requests from administration clients are sent using a platform-specific HTTP-based RPC. The RAAI contains a bridge for translating these requests to the native protocol of the agent platform. We have also implemented a bridge for IIOP-based clients. Other components of the RAAI provide information for administration clients, like the number of hosts in the system and the components installed at a specific host.

5 Deployment

Deployment is the process of installing and customizing applications in an operational environment [18]. Hall et. al. [19] describe a software deployment life-cycle and divide it into producer-side and consumer-side processes. The two main activities on the producer's side are *release* and *retire*. The release process encompasses all activities to package, provide, prepare and advertise software components for deployment to consumer sites. Retire is the process of withdrawing support for a particular software system by the producer. Since the retire process is not important in the context of this paper we refer to Hall et. al. [19] for a treatment of this subject.

Release packages include all physical artifacts comprising a component, descriptions of deployment requirements, and they may also contain initial or default configurations for activating a component as part of the install process. According to Hall et. al. [19] installation, activation, reconfiguration, update, and adaptation are consumer-side processes. We will not go as far as to include adaptation and reconfiguration as part of the deployment process in our system, albeit we recognize that this may be valid for the deployment of whole software systems, not individual components, only.

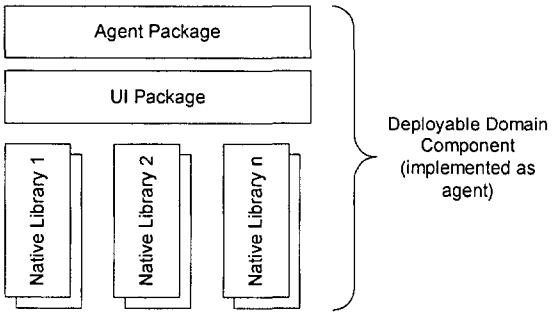


Figure 4: Component Packages

The units of deployment in our environments are mainly software components for supervision and data retrieval at the deployment target. These components are implemented as mobile agents mainly in Java based on the *Agent Platform Abstraction Layer* described in the previous section. As described above, the first step in the deployment process is to release a component for deployment. The three main steps are (1) specifying component properties and external dependencies (2) creating component packages and (3) making these packages available to deployment tools. A component need not be deployed in a single package. In our environment a deployable component may consist of one to several packages. The main packages are depicted in Figure 4.

The Agent Package consists of all Java classes implementing the agent's behavior and structure. The UI Package contains Java classes and sometimes HTML pages for configuring an agent during installation and operation. This package is not part of the Agent package, since the user interface code and resources are not needed during normal operation. The user interface is needed only by configuration and installation tools. Still it is deployed to the target environment for two main reasons. Firstly, agents can be configured from multiple locations with different Administration Servers at the same time. Keeping the user interface of an agent at one central location, i.e., within the agent's operational environment, makes it unnecessary to explicitly synchronize these servers. Secondly, older versions of agents may still be active and need to be configured while only new versions with new configuration interfaces are available at Administration Servers. We should note that the UI Package is optional. We also support automatic generation of (less flexible) configuration user interfaces based on component metadata and presentation annotations, as described in the next section.

Interaction with legacy systems and access to native operating system services is an important requirement for components in our environment. The API provided by the Java 2 platform is often not sufficient to access such systems and services. Thus, each agent may have a number of associated platform-specific (or native-code) libraries (see Figure 4), which have to be deployed with the agent-package and the optional UI Package. The symbols for platform-specific packages or libraries are stacked in Figure 4, illustrating that mobile agents might need different native libraries for the same purpose on different operating systems.

All packages are made available for deployment by putting them into a code repository at an Administration Server (see Figure 1). The repository supports multiple categories for different kinds of agents. The repository contains not only packages but also deployment descriptors for each domain component, specifying the packages that have to be deployed in order to successfully deploy the whole component. More specifically, the deployment descriptor contains a reference to the agent package, to an optional user interface package and to optional platform-specific libraries that are needed by the component at the target environment.

The contents of the repository can be visualized using an installation tool that is loaded from the Administration Server to one of the administration clients depicted in Figure 1. The installation tool is used for handling the consumer-side processes of the deployment life-cycle. An administrator may select a component from the repository, specify an initial configuration and define the target of the installation process. The target may be an individual host or simply the unit. In the latter case, the component will be installed at the Unit Gateway and transfers itself to the final destination within the unit depending on environmental conditions.

The installation process is performed as follows: (1) The installation tool analyses the deployment descriptor from the repository and transfers all specified packages to the destination via a secure Internet connection. (2) In addition to the packages the initial configuration as provided by the administrator is transferred. (3) The packages are stored at different locations at the target environment. (a) The agent package is delegated to the code management system of the agent system used. (b) The optional user interface package is stored on a dedicated UI code server. (c) The platform-specific libraries are managed by a special native-code management service (see Section 7 on mobility and code management). (4) Finally, the agent is activated and parameterized with the initial configuration.

Platform-specific libraries need a special treatment. In order to support mobility of agents within the target system, we have to ensure that an agent finds all required libraries for all hosts it might visit during operation at the target system. Therefore, the installation of these libraries requires additional steps. The installer first checks the available operating systems at the target system. Then, it determines whether the required libraries for these operating systems are already available in the correct version in a code repository at the Unit Gateway. Afterwards, it transfers all libraries that are still missing to the target environment. The installation terminates successfully, if the installer is able to transfer all platform-specific libraries for all operating system an agent might need during operation.

6 Dynamic Configuration

Our environment supports remote and dynamic configuration of both agent properties and dependency relations among agents. Configuration is performed from administration clients over secure Internet connections using graphical user interfaces.

The user interface for configuring the properties of a single agent is either fetched on demand from the agent's operational environment (see code on demand [22]) or it is automatically generated based on agent metadata. In the first case, user interface code for the agent has to be deployed to the target environment (see Section 5). In the second case, no user interface needs to be programmed and transferred, albeit less flexible user interfaces are possible. Both techniques enable configuration of agent properties from arbitrary administration clients, even if the agent itself has been deployed via a different Administration Server.

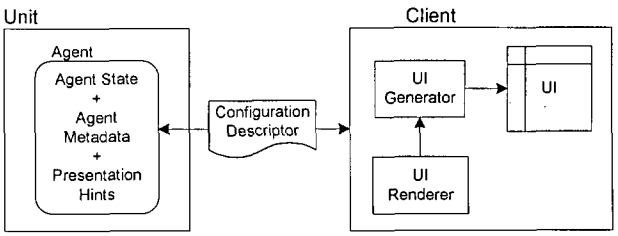


Figure 5: Automatic Generation of User Interfaces

Automatic generation of user interfaces for agent configuration is based on metadata about an agent's configurable properties, which is augmented by UI presentation hints. Examples for UI presentation hints are verbose names of configurable properties, validation ranges, units of measurement, and category qualifiers. Category qualifiers are used for grouping related properties at the user interface. Configuration based on metadata works as follows: (1) The current properties of an agent including structural and type information, current values, and UI presentation hints are stored into an XML-based configuration descriptor, which is transferred to the administration client. (2) At the client the descriptor is analyzed and the user interface is generated. (3) The changed configuration data is returned to the agent, which updates itself properly.

The user interface generator is able to generate different kinds of user interfaces using different renderers. Currently we generate user interfaces based on JFC/Swing (http://java.sun.com/jfc). Renderers for HTML and other kinds of user interfaces may be provided as well. The main elements of automatic user interface generation are depicted in Figure 5.

In addition to changing the properties of single agents at run-time, we provide tools for specifying dependency relations between different agents. Currently these relations are event relationships, which are maintained by an event service at run-time. Remote configuration tools use the Remote Administration and Access Interface (RAAI) for accessing the event service remotely and for viewing, creating, and changing event relationships.

7 Mobility Support and Code Management

Mobile code and especially mobile agents offer a number of benefits for the construction of distributed systems. Benefits of mobile agents that are often described are reduced network load, reduced latency, encapsulation of protocols, asynchronous execution and autonomy, dynamic adaptation, and fault tolerance (see [20][21][22]). Despite these potential advantages code mobility raises new problems, mainly concerning

security, resource management and accounting (see [23][24]). However, these issues are most evident if agents are used in a global setting, like agents roaming the Internet. They are less a problem if mobile agents are used within a rather closed environment, like in an Intranet or a local area network (LAN).

We support mobility within the LAN of the target unit. The most important benefits of using mobility in our system are dynamic adaptation, encapsulation of protocols, and support for distributed data retrieval. Agents performing supervision tasks may dispatch themselves to the appropriate location depending on information provided by the target environment. Agents performing long-term diagnosis tasks may roam the LAN and collect data from multiple hosts. Finally, agents are able to access legacy systems located on specific hosts using proprietary and local communication protocols.

The main elements that have to be transferred during agent migration are the agent's internal state, its code and its execution state. As long as agents are completely implemented in Java, all these issues are handled or at least supported by the underlying agent platform in our system. However, recall that some agents may represent domain components needing access to services that are not supported by the Java API. These agents typically use platform-specific (or native-code) libraries which have to be transferred to an agent's destination, also. Since this is not supported by the underlying agent platform, we provide a native-code management service for managing mobility of native code within the agent's network environment.

The deployment process ensures that all platform-specific libraries an agent might need during operation are available in a central repository at the target unit (see Section 5). The native-code management service is implemented by a collection of system agents, one at each host. The migration process works as follows: (1) The underlying agent platform is used for transferring the portable part of an agent, including its internal state and information about its execution state. (2) After arrival the agent contacts a local system agent responsible for native-code management and specifies its need for particular libraries. This request does not contain platform-specific information and is the same on all target platforms. (3) The system agent checks a local code cache and confirms the request if the needed libraries are available. Otherwise it gets the required libraries from a central repository and stores them locally in the code cache before confirming the request. The system agent is aware of its environment and thus is able to get the correct libraries from the central repository. (4) After confirmation the migrating agent is able to finish the transfer process and continues with its operation.

8 Security

The use of mobile code and especially of mobile agents in a truly open environment such as the Internet raises a number of security problems, which are difficult to solve (see [25][26]). The security requirements in our environment are different. We support remote administration and management of rather closed units with clearly defined access points. General problems in an open setting like protecting hosts from malicious agents and vice versa are not present in our environment. Instead, we have to provide secure connections as well as authentication and authorization for Internet-based access from administration clients to administered units as well as accurate logging of administration tasks. Figure 6 shows the system structure of our environment from a security perspective.

An administration client has to authenticate himself at the Administration Server with a username and password (see (1) in Figure 6). Authentication of the administration server to clients as well as secure transmission is realized using SSL [27]. This means that the Administration Server needs a trusted certificate that can be verified by clients.

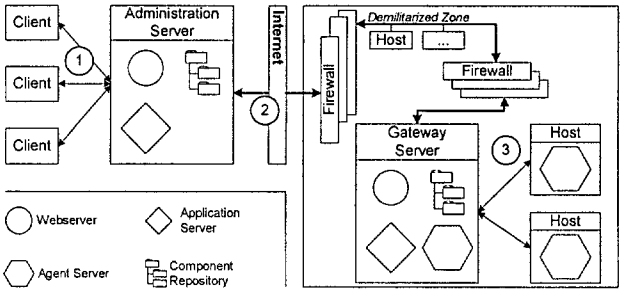


Figure 6: Security Architecture Overview

Access to corporate networks from the Internet is usually secured with firewalls, providing port and protocol filters. Our system architecture has been designed with typical corporate network security architectures (with cascaded firewalls) in mind. Communication between a client and an agent at a remote server is always routed via the Administration Server and the Gateway Server. Communication between the Administration Server and the Gateway Server (see (2) in Figure 6) requires a mutual certificate-based authentication of the Administration Server and the Gateway Server and is always encrypted.

All administration tasks like installation and termination of agents, activation of certain agent tasks (e.g., measurements), and configuration changes are checked by an authorization mechanism, which is based on client-side authentication. In addition, a logging service records all activities of administration clients, which is an important feature for tracking not only security holes but also administration faults.

Within the target unit no special security precautions are taken. Components trust their hosts as well as hosts trust their components. Communication among components within the system (see (3) in Figure 6) need not be

secured, though the underlying agent platform might provide intra-communication security as well.

9 Domain Specific Component Frameworks

Insight provides domain specific component frameworks for system supervision and control as well as system diagnosis tasks (see Section 2). The frameworks assign roles to the agents used for supervision and system diagnosis. Although the primary application domain of these frameworks is process automation, they are general enough to be used in other domains with similar requirements.

Agents for supervision tasks can be grouped into supervision and control agents. Supervision agents and control agents can be connected dynamically using a configuration tool as described in Section 6. Supervision agents continuously observe critical aspects of an automation system (e.g., steel quality attributes in a steel plant). A measurement framework allows customization of measurement activities like frequency of measurements and access to data sources. If a supervision agent detects a problem, it notifies all connected control agents. A control agent may either inform a human operator (e.g., by sending an electronic mail) or autonomously perform necessary operations. Figure 7 depicts typical coordination patterns for supervisor and control agents.

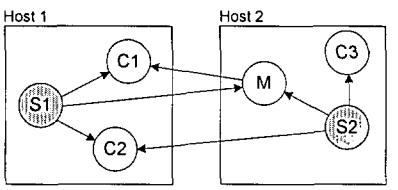


Figure 7: Coordination patterns for supervisor and control agents

In the figure, supervisor *S2* is connected to two control agents (*C2* and *C3*, on different hosts) that may be notified by *S2*. More complicated coordination patterns are also possible. For example, control agent *M* acts as a mediator for several supervision agents (*S1* and *S2* in Figure 7). The mediator analyzes notifications from all connected supervision agents before informing its associated control agents (*C1* in Figure 7). Connections between agents are maintained if an agent moves to another location.

The main components of system diagnosis are diagnosis agents, worker agents and data collectors. Diagnosis agents are used for collecting data about distributed resources over a longer period of time. They create and use worker agents for performing the actual measurement tasks. Usually multiple workers are active in parallel. The diagnosis agent acts as coordinator and supervisor for its worker agents. Measurement results are returned from the worker agents to a diagnosis agent, which subsequently may use other agents for data processing and report generation. Figure 8 depicts measurement strategies that are supported by Insight.

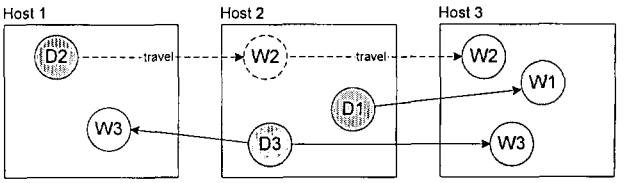


Figure 8: Measurement Strategies

In Figure 8, diagnoses agent *D1* implements a simple strategy. Only one worker agent *W1* is dispatched to a specific host. Diagnoses agent *D3*, however, uses two worker agents for collecting data from multiple hosts in parallel. Agent *D2* uses a worker agent *W2* that visits several hosts in sequential order and performs measurement tasks at each host. The measurement strategy of diagnosis agents can be configured. Worker agents may either collect data once or continuously. In the latter case measurement frequencies can be configured and a trend analysis is possible.

Analysis and preparation of measurement data for operators and managers is performed by data collector agents. Data collectors can be connected to the diagnoses agents using the configuration tool (see Section 6). Currently data collectors usually prepare reports in HTML, which can be viewed with standard web browsers. Data collectors use a data processing framework, which can be parameterized with other components supporting reports in different formats or storing measurement results in a quality database, for example.

10 Related Work

Insight is component platform for dynamically adaptable distributed applications. As a component platform the system supports naming, interoperability, deployment, evolution, customization and composition of software components. The most notable difference to other component models and platforms like Enterprise JavaBeans (http://java.sun.com/ejb), Windows .NET (http://www.microsoft.com/net), and the CORBA Component Model (http://www.omg.org) is that components and services in our system are based on mobile agents. This leads to some of the key features of our environment like dynamic services, support for mobility, dynamic adaptation, and support for legacy systems. A detailed comparison with the above component models and platforms is beyond the scope of this paper.

Also, it is not useful to compare our environment with other agent platforms. We rely on a minimum functionality for agent creation, destruction, communication, and mobility, which is expressed by an

agent platform abstraction layer. The underlying agent system may be replaced by an agent system providing a similar functionality. Research on mobile agent platforms often focuses on security issues in a truly open setting (e.g., [28][29]) and on high-level interaction of agents owned by different principals. The main focus of our environment, however, is a flexible component platform with support for mobility, which utilizes agent technology for increased extensibility and adaptability.

Another focus of our system is on remote administration, including deployment and configuration. Similar approaches for remote management of distributed resources are the Simple Network Management Protocol SNMP [30] and extensions to SNMP like RMON [31]. The focus of these approaches is on remote network management. They may be used not only for the management of computers in a network but also for network devices like switches and routers. They are not based on a component model and usually provide only lower level interfaces for administration. Configuration is sometimes supported based on static predefined tasks like in RMON. SNMPv3 [32] allows to dynamically install and execute scripts. In the telecommunication area network management is, for historical reasons, built upon the OSI model (TMN/OSI [37][38]). TMN/OSI mandates the use of the Common Management Information Protocol (CMIP), which in its nature is comparable to the SNMP model [39].

Java-based approaches for remote management of distributed resources are Jiro [33] and JMX [34]. Jiro is based on Jini [35] and was initially created for the purpose of storage management. We will discuss only JMX, since SUN is currently working on integrating Jiro with JMX [36]. The management components of JMX are MBeans. MBeans may be compared to domain components (i.e., supervisor, control and diagnosis agents) in the Insight environment. JMX supports a notification service for MBeans, which is comparable to the event service provided by our system. MBean objects are hosted by an MBean server, which has to be available at each host. In our environment, the execution environment for agents is provided by the underlying agent platform and is typically an agent server per host. MBeans may also offer services to other MBeans. Examples for MBeans-based services are remote class loading (m-let service), task scheduling (timer service), and monitoring attributes of other MBeans. These services may be installed and removed dynamically and thus are comparable to the dynamic services in our environment.

Mobility of MBeans is not directly supported by JMX, though it could be implemented on basis of the m-let service. Also, the deployment process is not directly supported by JMX. JMX provides only limited support for native code management using the m-let service [34] while Insight supports deployment and mobility of native-code in heterogeneous environments. Tools for remote administration via the Internet are not part of the JMX specification. Also, support for automatic generation of configuration interfaces (see UI presentation metadata in Section 6) is not included in JMX. The JMX specification contains no information about JMX specific security issues. The Insight security model is tailored to the described application requirements and supports not only mutual authentication and authorization but also logging of administrative tasks.

11 Conclusion

Global networking via the Internet and wireless and mobile computing raise additional demands on component platforms and management. Global networking makes remote administration and management possible without leasing dedicated lines. Mobile and wireless computing have initiated a trend towards a spectrum of different end user devices for accessing the Internet. Security threats are increasing. Interaction with and integration of legacy systems becomes more important and raises additional demands on component systems.

We have presented an agent-based component platform and environment, which offers solutions to some of these problems. Key issues of our system are dynamic services, support for remote and dynamic deployment and configuration, support for mobility, management of platform-specific code in heterogeneous environments, and multi-protocol remote access of software components. We also provide a security solution for remote administration tasks, and component frameworks for data retrieval and supervision of hard- and software-resources.

The system is currently used for remote supervision and control of steel-plant process automation systems. Further application domains are investigated. Main issues of future work are better support for agent interaction and increased security and fault tolerance.

12 References

[1] C. Szyperski: Component Software – Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[2] G.T. Heineman, W.T. Councill (eds.): *Component-Based Software Engineering*, Addison-Wesley, 2001.

[3] R. Weinreich, J. Sametinger: "Component Models and Component Services - Concepts and Principles", in *Component-Based Software Engineering* (www.cbseng.com), G.T. Heineman, W.Councill (ed.), Addison-Wesley 2001.

[4] J.M. Bradshaw (ed.): *Software Agents*, MIT Press, 1997.

[5] M.L. Griss: "Software Agents as Next Generation Software Components", in *Component-Based Software Engineering*, G.T. Heineman, W.Councill (ed.), Addison-Wesley 2001.

[6] M.L. Griss, G. Pour: "Accelerating Development with Agent Components", IEEE Computer Vol. 34, No. 5, May 2001.

[7] D. Kafura, J.P. Briot: "Actor Agents", IEEE Concurrency, April/June 1998, pp. 24-29.

[8] M. Wooldridge, N.R. Jennings: "Intelligent Agents: Theory and Practice", Knowledge Engineering Review, Vol 10, No. 2, Cambridge University Press, 1995.

[9] ACM: "Multiagent Systems on the Net and Agents in E-commerce", CACM, Vol. 42, No. 3, March 1999.

[10] C. Petrie: "Agent-Based Software Engineering", Proceedings of the First International Workshop on *Agent-Oriented Software Engineering* (Eds. P. Ciancarini and M. Wooldridge), Lecture Notes in AI, Vol. 1957, Springer, 2001.

[11] B. Brewington, R. Gray, K. Moizumi, D. Kotz, G. Cybenko, D. Rus: "Mobile agents in distributed information retrieval", in *Intelligent Information Agents*, Matthias Klusch (ed.), Springer Verlag, 1999.

[12] A. Bieszczad, B. Pagurek, T. White: "Mobile Agents for Network Management", IEEE Communications Surveys, September 1998.

[13] W.J. Buchanan, M. Naylor, A.V. Scott: "Enhancing Network Management using Mobile Agents", Proceedings of the 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Edinburgh, Scotland, April 2000.

[14] S. Albayrak: "Agent-oriented technology for telecommunications: introduction", CACM, Vol. 44, No. 4, April 2001.

[15] S. Fricke, K. Bsufka, J. Keiser, T. Schmidt, R. Sesseler, S. Albayrak: "Agent-based telematic services and telecom applications", CACM, Vol. 44, No. 4, April 2001.

[16] M. P. Papazoglou: "Agent-oriented technology in support of e-business", CACM, Vol. 44, No. 4, April 2001.

[17] J. White: "Mobile Agents", in *Software Agents*, J. Bradshaw (ed.), MIT Press, 1997, pp. 437-472.

[18] N. Kassem (ed.): *Designing Enterprise Applications with the Java(tm) 2 Platform, Enterprise Edition*, Addison-Wesley, 2000.

[19] R.S. Hall, D. Heimbigner, A.L. Wolf: "A Cooperative Approach to Support Software Deployment Using the Software Dock", Proceedings of the International Conference on Software Engineering 1999 (ISCE '99), Los Angeles, California, USA, 1999.

[20] D.B. Lange, M. Oshima: "Seven Good Reasons for Mobile Agents", CACM, Vol. 42, No. 3, March 1999, p. 88-89.

[21] D. Wong, N. Paciorek, D. Moore: "Java-based Mobile Agents", CACM, Vol. 42, No. 3, March 1999, pp. 92-102.

[22] G.P. Picco: *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility*, doctoral thesis, Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy, 1998.

[23] D. Chess, C. Harrison, A. Kershenbaum: "Mobile Agents: Are they a good idea?", IBM Research Report, T.J. Watson Research Center, Yorktown Heights, New York, 1995.

[24] D. Kotz, R. Gray: "Mobile Agents and the Future of the Internet", ACM Operating Systems Review, August 1999, pp. 7-13.

[25] W. Jansen, T. Karygiannis: "Mobile Agent Security", National Institute of Standards and Technology, Computer Security Resource Center, Special Publication 800-19, available online at http://csrc.nist.gov/mobileagents/, August 1999.

[26] N. Karnik: *Security in Mobile Agent Systems*, PhD dissertation, Computer Science and Engineering, University of Minnesota, 1998.

[27] E. Rescorla: *SSL and TLS – Designing and Building Secure Systems*, Addison-Wesley Professional, 2000.

[28] G. Karjoth, D.B. Lange, M. Oshima: *A Security Model for Aglets*, IEEE Internet Computing, July-August 1997, pp. 68-77.

[29] C. Bryce, J. Vitek: *The JavaSeal Mobile Agent Kernel*, Journal on Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers, 4, 2001, pp. 359-384.

[30] J. Case, M. Fedor, M. Schoffstall, J. Davin (eds.): RFC1157, *A Simple Network Management Protocol (SNMP)*, IETF, May 1990.

[31] W. Stallings: SNMP, SNMPv2, SNMPv3, and RMON 1 and 2, Third edition, Addison-Wesley, Reading, MA, USA 1999.

[32] D. Levi, P. Meyer, B. Steward (eds.): RFC 2573 – SNMPv3 Applications, IETF, April 1999.

[33] P. Monday, W. Connor: The Jiro Technology Programmer's Guide and Federated Management Architecture, Addison-Wesley, 2001.

[34] Sun Microsystems: Java Management Extensions – Instrumentation and Agent Specification, V 1.0, Final Release, July 2000.

[35] K. Arnold, B. O'Sullivan, R.W. Scheifler, J. Waldo, A. Wollrath: The Jini Specification, Addison-Wesley, 1999.

[36] J.P. Martin-Flatin and S. Znaty: "Two Taxonomies of Distributed Network and Systems Management Paradigms", Technical Report DSC/2000/032, DSC, EPFL, Lausanne, Switzerland, July 2000.

[37] ITU-T: "Recommendation M.3010 – Principles for a Telecommunications management network", Geneva, Switzerland, May 1996.

[38] J.K. Shrewsbury: *An Introduction to TMN*, Journal of Network and Systems Management, Vol. 3, No. 1, 1995.

[39] M. Baldi, G.P. Picco: "Evaluating the Tradeofs of Mobile Code Design Paradigms in Network Management Applications", In Proceedings of the 20th International Conference on Software Engineering (ICSE'98), Kyoto (Japan), IEEE CS Press, April 1998

MobiDoc: a mobile agent-based framework for compound documents

Ichiro Satoh
National Institute of Informatics /
Japan Science and Technology Corporation
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430 Japan
Tel: +81-3-4212-2546 Fax: +81-3-3556-1916
E-mail: ichiro@nii.ac.jp

Keywords: compound document, mobile agent, software component, distributed system

Received: May 6, 2001

This paper presents a mobile-agent-based framework for building mobile compound documents, called MobiDoc, where the compound document can be dynamically composed of mobile agent-based components and can migrate itself over a network as a whole, with all its embedded agents. The key idea of this framework is that it builds a hierarchical mobile agent system that enables multiple mobile agents to be combined into a single one. The framework also provides several value-added mechanisms for visually manipulating components embedded in a compound document and for sharing a window on the screen among the components. This paper describes this framework and its prototype implementation, currently using Java as the implementation language as well as a component development language, and then illustrates several interesting applications to demonstrate the utility and flexibility of this framework.

1 Introduction

Building systems from software components has already proven useful in the development of large and complex systems. Several frameworks for software components have been developed, such as COM/OLE [4], OpenDoc [1], CommonPoint [10], and JavaBeans [7]. Among them, the notion of compound documents is a document-centric component framework, where various visible parts, such as text, image, and video, created by different applications can be combined into one document and be independently manipulated in-place in the document. An example of this type of framework is CI Labs' OpenDoc [1] developed by Apple computer and IBM, although their development work on this framework has stopped. However, there have been several problems in the few existing compound document frameworks. A compound component is typically defined by two parts: contents and codes for modifying the contents. Contents are stored inside the component but the codes for accessing them are not always. Thus, a user cannot view or modify a document whose contents need the support of different applications, if the user does not have the applications. Moreover, existing compound documents are inherently designed as passive entities in the sense that they can be transmitted over a network by external network systems such as electronic mail systems and workflow management systems and cannot determine where it should go next. We also need network-wide manipulation for building and assembling various components located in different computers into a document. Therefore, not only a whole compound document but also each of the components of the document must be able to be transmitted to another computer.

The goal of this paper is to propose a new framework for building mobile compound documents. Each document is built as a component that can be a container for components that can migrate over a network. Accessing compound documents over a network requires a powerful infrastructure for building and migrating, such as mobile agents. Mobile agents are autonomous programs that can travel from computer to computer under their own control. When an agent migrates over a network, both the state and the codes can be transferred to the destination. However, traditional mobile agent systems cannot be composed of more than one mobile agent, unlike component technology. Therefore, we built a framework on a unique mobile agent system, called *MobileSpaces*, which was presented in an earlier paper [12]. The system is constructed using Java language [2] and provides mobile agents that can move over a network, like other mobile agent systems. However, it also allows more than one mobile agent to be hierarchically assembled into a single mobile agent. Consequently, in our framework, a compound document is a hierarchical mobile agent that contains its contents and a hierarchy of mobile agents, which correspond to nested components embedded in the document. Furthermore, the framework offers several mechanisms for coordinating visible components so that they can effectively share visual real estate on a screen in a seamless-manner.

This paper is organized as follows. Section 2 surveys related work and Section 3 presents the basic ideas of the compound document framework, called *MobiDoc*. Section 4 details its prototype implementation and Section 5 shows the usability of our framework based on real-world exam-

ples. Section 6 makes some concluding remarks.

2 Background

Among the component technologies developed so far, OpenDoc and JavaBeans are characterized by allowing a component to contain a hierarchy of nested components. Although there are few hierarchical components available on the market today, their advent appears to be necessary and unavoidable in the long run.

OpenDoc is a document-centric component framework and has several advantages over other frameworks, but it has been discontinued. An OpenDoc component is not self-configurable, although it is equipped with scripts to control itself, so a component cannot migrate over a network under its own control. JavaBeans is a general framework for building reusable software components designed for the Java language. The initial release of JavaBeans (version 1.0 specified in [7]) did not contain a hierarchical or logical structure for JavaBean objects, but its latest release specified in [5] allows JavaBean objects to be organized hierarchically. However, the JavaBeans framework does not provide any higher-level document-related functions. Moreover, it is not inherently designed for mobility. Therefore, it is very difficult for a group of JavaBean objects in the containment hierarchy to migrate to another computer.

A number of other mobile agent systems have been released recently, for example Aglets [8], Mole [3], Telescript [17], and Voyager [9]. However, these agent systems unfortunately lack a mechanism for structurally assembling more than one mobile agent, unlike component technologies. This is because each mobile agent is basically designed as an isolated entity that migrates independently. Some of them offer inter-agent communication, but they can only couple mobile agents loosely and thus cannot migrate a group of mobile agents to another computer as a whole. Telescript introduces the concept of places in addition to mobile agents. Places are agents that can contain mobile agents and places inside them, but they are not mobile. Therefore, the notion of places does not support mobile compound documents.

To solve the above problem in existing mobile agent systems, we constructed a new mobile agent system, called MobileSpaces, in a previous paper [12]. The system introduces the notion of agent hierarchy and inter-agent migration. This system allows a group of mobile agents to be dynamically assembled into a single mobile agent. Although the system itself has no mechanism for constructing compound documents, it can provide a powerful infrastructure for implementing compound documents in network computing settings. Also, we presented a compound document framework as just an application of the MobileSpaces system [13]. Therefore, the previous framework lacked many functionalities, which are provided by the framework presented in this paper. For example, it could deliver a compound document as a whole to another computer, but not decompose a document into components or migrate each component to another computer independently. As a result, the previous one could not fetch and assemble components located at different computers into a compound document.

ADK [6] is a framework for building mobile agents from JavaBeans. It provides an extension of Sun's visual builder tool for JavaBeans, called BeanBox, to support the visual construction of mobile agents. In contrast, we intend to construct a new framework for building mobile compound documents in which each component can be a container for components and can migrate over a network under its own control. Our compound document will be able to migrate itself from one computer to another as a whole with all of its embedded components to the new computer and adapt the arrangement of its inner components to the user's requirements and its environments by migrating and replacing corresponding components.

We should explain why our hierarchical mobile agent is essential in the development of compound documents. The reader might think that existing software development methodologies such as JavaBeans and OpenDoc, enable components to be shipped to other computers. Indeed, in the current implementation of our system each mobile agent can be a container of JavaBeans and can get as a whole with its inner Java Beans. However, JavaBean components are not inherently designed to be mobile components, unlike mobile agents. Therefore, it is difficult to migrate each JavaBean component over the network under its own control. On the other hand, our framework introduces a document (or a component) as an active entity that can travel from computer to computer under its own control. Therefore, our document can determine where it should go next, according to its contents. Moreover, it can dynamically adapt the layouts and combinations of its inner components to the user's requirements and the environments.

3 Approach

This section outlines the framework for building compound documents based on mobile agents called *MobiDoc*.

3.1 Mobile Agent-based Components

To create an enriched compound document, a component or document must be able to contain other components, like OpenDoc. On the other hand, each mobile agent resembles a software component in the sense that each entity is a self-contained module holding its code and state, but most existing mobile agent systems do not allow a mobile agent to be composed structurally. Furthermore, each mobile agent is characterized by its mobility. Thus, a composition of mobile agents must be designed to keep their mobility. We intend to provide such a component through a hierarchical mobile agent. Our framework is therefore built on MobileSpaces [12] which can dynamically assemble more than one mobile agent into a single mobile agent. The system supports mobile agents that are computational

and itinerant entities, like other mobile agent systems. It also incorporates the following concepts:

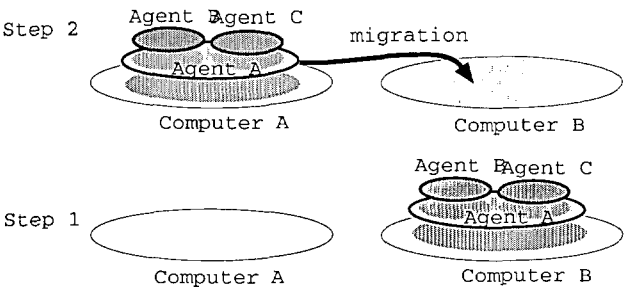


Figure 1: Agent Hierarchy and Group Migration.

- **Agent Hierarchy** The first concept means that each mobile agent can be contained within one mobile agent. It enables us to assemble more than one mobile agent into a single mobile agent in a tree structure.
- **Group Migration** The second concept means that each mobile agent can migrate to another agent or another computer as a whole, with all of its inner agents. It allows a group of mobile agents to be treated as a single mobile agent during their migration.

The first concept is needed in the development of a mobile compound document, because such a document should be able to contain other components, like OpenDoc. The second concept enables a compound document to migrate itself and its components as a whole. Accordingly, a compound document is given as a collection of mobile components and can be treated as a mobile component. Figure 1 shows an example of an inter-agent migration in an agent hierarchy. In an agent hierarchy, each agent is still mobile and can freely move into any computer or any agent in the same agent hierarchy except into itself or its inner agents, as long as the destination accepts the moving agent.

3.2 Compound Document Framework

MobileSpaces is a suitable infrastructure for mobile compound documents, but it does not provide any document-centric mechanisms for managing components in a compound document. We offer a compound document framework for supporting mobile agent-based components, including graphical user interfaces for manipulating visible components. This framework, called *MobiDoc*, is given as a collection of Java objects that belong to one of about 50 classes. It defines the protocols that let components embedded in a document communicate with each other. It also deals with in-place editing services similar to those provided by OpenDoc and OLE. The framework offers several mechanisms for effectively sharing the visual estate of a container among embedded components and for coordinating their use of shared resources, such as keyboard, mouse, and window.

4 Implementation

Next, we will describe our method for using MobileSpaces to construct mobile compound documents.[1] It has been incorporated in Java Development Kit version 1.2 and can run on any computer that has a runtime system compatible with this version.

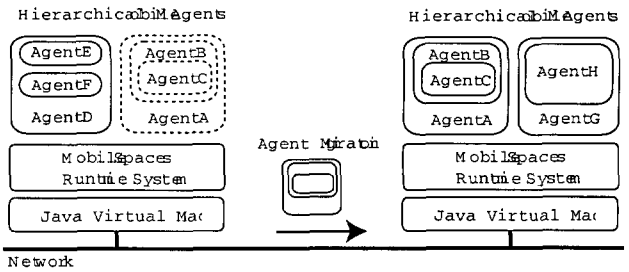


Figure 2: Agent Migration between Two MobileSpaces Runtime Systems.

4.1 MobileSpaces Runtime System

The MobileSpaces runtime system is a platform for executing and migrating mobile agents. It is built on a Java virtual machine and mobile agents are given as Java objects [2]. Each component is given as a mobile agent in the system and the containment hierarchy of components in a document is given as an agent hierarchy managed by the system. The runtime system has the following functions:

Agent Hierarchy Management

The agent hierarchy is given as a tree structure in which each node contains a mobile agent and its attributes. The runtime system is assumed to be at the root node of the agent hierarchy. Agent migration in an agent hierarchy is performed just as a transformation of the tree structure of the hierarchy. In the runtime system, each agent has direct control of its internal agents. That is, a container agent can instruct its embedded agents to move to other agents or computers, serialize them and destroy them. In contrast, an embedded agent has no direct control over its container agent. It can only access the collection of service methods offered by its container agents.

Agent Life-cycle Management

The runtime system is at the root node of the agent hierarchy and can control all the agents in the agent hierarchy. Furthermore, it maintains the life-cycle of agents: initialization, execution, suspension, and termination. When the life-cycle state of an agent is changed, the runtime system issues events to invoke certain methods in the agent

[1] Details of the MobileSpaces mobile agent system can be found in our previous paper [12].

and its containing agents. Moreover, the runtime system enforces interoperation among mobile agent-based components. The runtime system monitors changes in components and propagates certain events to the right components. For example, when a component is added to or removed from its container component, the system dispatches certain events to the component and the container.

Agent Migration Mechanism

Each document is saved and transmitted as a group of mobile agents. When a component is moved inside a computer, the component and its inner components can still be running. When a component is transferred over a network, the runtime system stores the state and the codes of the component, including the components embedded in it, into a bit-stream formed in Java's JAR file format that can support digital signatures for authentication. The system provides a built-in mechanism for transmitting the bit-stream over the network by using an extension of the HTTP protocol. The current system basically uses the Java object serialization package for marshaling components. The package does not support the capturing of stack frames of threads. Instead, when a component is serialized, the system propagates certain events to its embedded components to instruct the agent to stop its active threads.

4.2 Mobile Agent Program

In our compound document framework, each component is a group of mobile agents in MobileSpaces. They consist of a body program and a set of services implemented in Java language. The body program defines the behavior of the component and the set of services defines various APIs for components embedded within the component. Every agent program has to be an instance of a subclass of the abstract class `ComponentAgent`, which consists of some fundamental methods to control the mobility and life-cycle of a mobile agent-based component as shown in Figure 3.

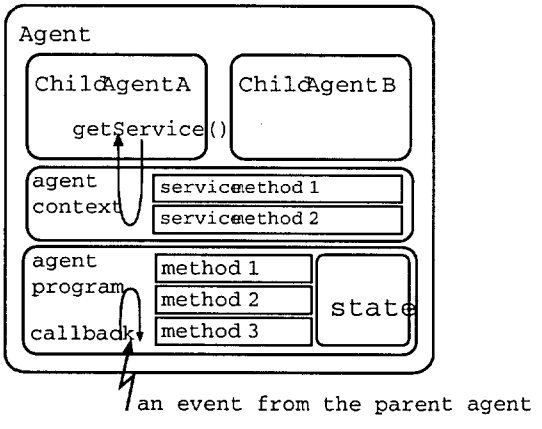


Figure 3: Structure of a Hierarchical Mobile Agent.

```
1: public class ComponentAgent extends Agent {
2:   // (un)registering services for inner agents
3:   void addContextService(
4:     ContextService service) { ... }
5:   void removeContextService(
6:     ContextService service) { ... }
7:   ....
8:   // (un)registering listener objects
9:   // to hook events
10:  void addListener(
11:    AgentEventListener listener) { ... }
12:  void removeListener(
13:    AgentEventListener listener) { ... }
14:  ....
15:  void getService(Service service)
16:    throws ... { ... }
17:  void go(AgentURL url)
18:    throws ... { ... }
19:  void go(AgentURL url1, AgentURL url2)
20:    throws ... { ... }
21:  byte[] create(byte[] data) throws ... {...}
22:  byte[] serialize(AgentURL url) throws ... {...}
23:  AgentURL deserialize(byte[] data)
24:    throws ... {...}
25:  void destroy(AgentURL url) throws ... {...}
26:  ....
27:  ComponentFrame getFrame() { ... }
28:  ComponentFrame getFrame(
29:    AgentURL url) {...}
30:  ....
32: }
```

The methods used to control mobility and life-cycle defined in the `ComponentAgent` class are as follows:

- An agent can invoke public methods defined in a set of service methods offered by its container by invoking the `getService()` method with an instance of the `Service` class. The instance can specify the kind of service methods, arbitrary objects as arguments, and deadline for timeout exception.
- When an agent performs the `go(AgentURL url)` method, it migrates itself to the destination agent specified by `url`. The `go(AgentURL url1, AgentURL url2)` method instructs the descendant specified as `url1` to move to the destination agent specified as `url2`.
- Each container agent can dispatch certain events to its inner agents and notify them when certain actions happen within their surroundings.

Our framework provides an event mechanism based on the delegation-based event model introduced in the Abstract Window Toolkit of JDK 1.1 or later, like Aglets [8]. When an agent is migrated, marshaled, or destroyed, our runtime system does not automatically release all the resources, such as files, windows, and sockets, which are acquired by the agent. Instead, the runtime system can issue certain events in the changes of life-cycle states. Also, a container agent can dispatch certain events to its inner mobile agent-based components at the occurrence of user-interface level actions, such as mouse clicks, keystrokes, and window activation, as well as at the occurrence of application level actions, such as the opening and closing of documents. To hook these events, each mobile agent-based component can have one or more listener objects which implement certain methods invoked by the runtime system and its container component. For example, each component can have one

or more activities that are performed using the Java thread library, but it needs to capture certain events issued before it migrates over a network and stop its own activities.

4.3 MobiDoc Compound Document Framework

The *MobiDoc* framework is implemented as a collection of Java classes to embody some of the principles of component-interoperation and graphical user interface.

Visual Layout Management

Each mobile agent-based component can be displayed within the estate of its container or a window on the screen, but it must be accessed through an indirection: *frame* objects derived from the `ComponentFrame` class.[2] as shown in Fig. 4. Each frame object is the area of the display that represents the contents of components and is used for negotiating the use of geometric space between the frame of its container component and the frame of its component.

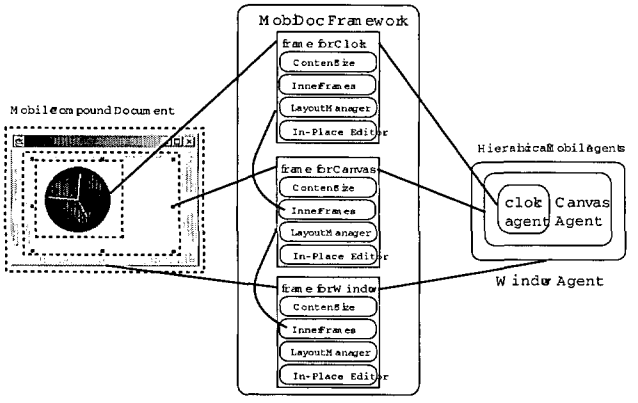


Figure 4: Components for Compound Document in Agent Hierarchy.

The frame object of each container component manages the display of the frames of the components it contains. That is, it can control the sizes, positions, and offsets of all the frames embedded within itself, while the frame object of each contained component is responsible for drawing its own contents. For example, if a component needs to change the size of its frame by calling the `setFrameSize()` method, its frame must negotiate with the frame object of its container for its size and shape and redraw its contents within the frame.

```
1:  public class ComponentFrame
2:  extends java.awt.Panel {
3:     // sets the size of the frame
4:     void setFrameSize(java.awt.Point p);
5:     // gets the size of the frame
6:     java.awt.Point getFrameSize();
7:     // sets the layout manager for
8:     // the embedded frames
9:     void setLayout(CompoundLayoutManager mgr) {
10:    // views the type of the component, . . .
11:    // e.g. iconic, thumbnail, or framed,
12:    int getViewType();
13:    // gets the reference of the container's frame
14:    ComponentFrame getContainerFrame();
15:    // adds an embedded component specified as frame
16:    void addFrame(ComponentFrame frame);
17:    // removes an embedded component
18:    // specified as frame
19:    void removeFrame(ComponentFrame frame);
20:    // gets all the references of embedded frames
21:    ComponentFrame[] getEmbeddedFrames();
22:    // gets the offset and size of the inner frame
23:    // specified as cf
24:    java.awt.Rectangle getEmbeddedFramePosition(
25:                                  ComponentFrame cf);
26:    // sets the offset and size of the inner frame
27:    // specified as cf
28:    void setEmbeddedFramePosition(ComponentFrame cf,
29:                                  java.awt.Rectangle);
30:    ....
31: }
```

[2]Although the `ComponentFrame` class is a subclass of the `java.awt.Panel` class, we call them *frame* objects because many existing compound document frameworks often call the visual space of an embedded component *frame*.

When one component is activated, another component is usually deactivated but does not necessarily become idle. To create a seamless application look, components embedded in a container component need to share, in a coordinated manner, several resources, such as keyboard, mouse, and window. Each component is restricted from directly accessing such shared resources. Instead, the frame object of one activated component is responsible for handling and dispatching user interface actions issued from most resources, and can reserve these resources until it sends a request to relinquish them.

In-Place Editing

Our framework provides for document-wide operations, such as mouse click and keystrokes. It can dispatch certain events to its components to notify them when certain actions happen within their surroundings. Moreover, the framework provides each container component with a set of built-in services for switching among multiple components embedded in the container and for manipulating the borders of the frame objects of its inner components. One of these services offers graphical user interfaces for in-place editing. This mechanism allows different components in a document to share the same window. Consequently, components can be immediately manipulated in-place, without the need for opening a separate window for each component.

To directly interact with a component, we need to make the component *active* by clicking the mouse within its frame. When a component is active, we can directly manipulate its contents. When the boundary of the frame is clicked, the frame becomes *selected* and displays eight rectangle control points for moving it around and resizing it, as shown in Fig. 5. The user can easily resize and move the selected component by dragging its handles.

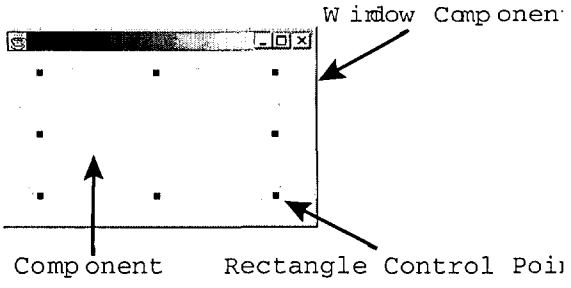


Figure 5: Selected Component and its Rectangle Control Points.

Structured Storage and Migration

While migrating over a network and being stored on a disk, each component must be responsible for transforming its own contents and codes into a stream of bytes by using the serialization facility of the runtime system. However, the frame object of each component is not stored in the component. Instead, it is dynamically created and allocated in its container's frame, when it becomes visible and restored. The framework automatically removes frame objects of each component from the screen and stores specified attributes of the frame object in a list of values corresponding to the attributes, because other frame objects may refer to objects that are not serializable, such as several visible objects in the Java Foundation Class package. After restoring such serialized streams as components at the destination, the framework appropriately redraws the frames of the components, as accurately as possible.

Network-Wide Component Assembly

Nowadays, cut-and-paste is one of the most common manipulations for assembling visible components. However, while a cut-and-paste on a single computer is easy, the system often forces users to transfer information between computers in a very different way. Therefore, our framework offers a mechanism for cutting and pasting between different computers. When a cut operation occurs at a component in one (source) container, the mechanism marshals the component and transmits the resulting byte sequence to another (destination) container at a local or remote computer by using the agent migration management of MobileSpaces. It becomes an infrastructure for providing a network-wide and direct manipulation technique, such as Pick-and-Drop that is a kind of network-wide drag-and-drop manipulations studied in [11].

4.4 Current Status

The MobiDoc framework has been implemented in the MobileSpaces system using the Java language (JDK1.2 or later version), and we have developed various components for compound documents, including the examples presented in this paper. The MobiDoc framework and the MobileSpaces System are constructed independently of the underlying system and can run on any computer with a JDK 1.2-compatible Java runtime system.

MobileSpaces is a general-purpose mobile agent system. Therefore, mobile agents in the system may be unwieldy as components of compound documents, but our components can inherit the powerful properties of mobile agents, including their activity and mobility. Security is essential in compound documents as well as mobile agents. The current system relies on the Java security manager and provides a simple mechanism for authentication of components. A container component can judge whether to accept a new inner component or not beforehand, while the inner components can know the available methods embedded in their containers by using the class introspector mechanism of the Java language. Furthermore, since a container agent plays a role in providing resources for its inner agent, it can limit the accessibility of its inner components to resources such as window, mouse, and keyboard, by hiding events issued from these resources.

Even though our implementation was not built for performance, we have conducted a basic experiment on component migration with computers (Pentium III-800MHz with Windows2000 and SUN JDK 1.2). The time of a component migration from a container to another container in the same hierarchy was measured to be 30 ms, including the cost to draw the visible content of the moving component and to check whether the component is permitted to enter the destination agent. The cost of component migration between two computers connected by Fast-Ethernet was measured to be 120 ms. The cost is the sum of the marshaling, compression, opening a TCP connection, transmission, acknowledgment, decompression, security and consistency verifications, unmarshaling, layout of the visual space, and drawing of the contents. The moving component is a simple text viewer and its size (the sum of code and data) is about 4 Kbytes (zip-compressed). We believe that the latency of component migration in our framework is reasonable for a Java-based visual environment for building documents.

5 Examples

The MobiDoc compound document framework is powerful and flexible enough to support radically different applications. This section shows some examples of compound documents based on the MobiDoc framework.

5.1 Electronic Mail System

One of the most illustrative examples of the MobiDoc framework is for the provision of mobile documents for communication and workflow management. We have constructed an electronic mail system based on the framework. The system consists of an inbox document and letter documents as shown in Fig. 6. The inbox document provides a

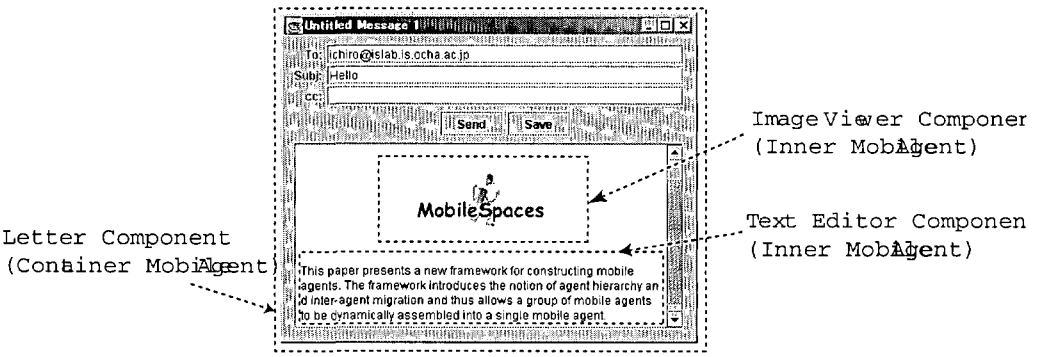


Figure 6: Structure of a Letter Document.

window that can contain two components. One of the components is a history of received mails and the other component offers a visual space for displaying the contents of mail selected from the history. The letter document corresponds to a mobile agent-based letter and can contain various components for accessing text, graphics, and animation. It also has a window for displaying its contents. It can migrate itself to its destination, but it is not a complete GUI application because it cannot display its contents without the collaboration of its container, i.e., the inbox document.

For example, to edit the text in a letter component, one simply clicks on it, and an editor program is invoked by the in-place editing mechanism of the MobiDoc framework. The component can deliver itself and its inner components to an inbox document at the receiver. After a moving letter has been accepted by the inbox document, if a user clicks a letter in the list of received mail, the selected letter creates a frame object of itself and requests the document to display the frame object within the frame of the document. The key idea of this mail system is that it combines different mobile agent-based components into a seamless-looking compound document and allows us to immediately display and access the contents of the components in-place. Since the inbox document is the root of the letter component, when the document is stored and moved, all the components embedded in the document are stored and moved with the document.

5.2 Desktop Teleporting

We constructed a mobile agent-based desktop system similar to the Teleporting System and the Virtual Network Computing system. These systems are based on the X Window System and allow the running applications in the computer display to be redirected to a different computer display.

In contrast, our desktop system consists of mobile agent-based applications and thus can migrate not only the appearance of applications but also the applications themselves to another computer (Fig. 7). The system consists of a window manager document and its inner applications. The manager corresponds to a desktop document at the top of the component hierarchy of applications separately displayed in their own windows on the desktop on the screen. It can be used to control the sizes, positions, and overlaps of the windows of its inner applications. When the desktop document is moved to another computer, all the components, including their windows, move to the new computer. The framework tries to keep the moving desktop and applications the same as when the user last accessed them on the previous computer, even when the previous computer and network have stopped. For example, the framework can migrate a user's custom desktop and applications to another computer that the user is accessing.

6 Conclusion

We have presented an approach for building compound documents. The key idea of the approach is to build compound documents from hierarchical mobile agents in the MobileSpaces system, which allows more than one mobile agent to be dynamically assembled into a single mobile agent. Our approach allows a compound document to be dynamically composed of mobile components and to be migrated over a network as a whole with its inner components under its own control. We designed and built a framework, called MobiDoc, to demonstrate the usability and flexibility of this approach. The framework provides value-added services for coordinating mobile agent-based components embedded in a document.

Finally, we would like to point out further issues to be resolved. To develop compound documents more effectively, we need a visual builder for our mobile components. We plan to extend a visual builder tool for JavaBeans, such as the BeanBox system included in the Bean Development Kit (BDK) [15], so that can support mobile agent-based compound documents. In the current system, resource management and security mechanisms are incorporated relatively straightforwardly. These should now be designed for mobile compound documents. Additionally, the programming interface of the current system is not yet satisfactory. We plan to design a more elegant and flexible interface incorporating existing compound document technologies.

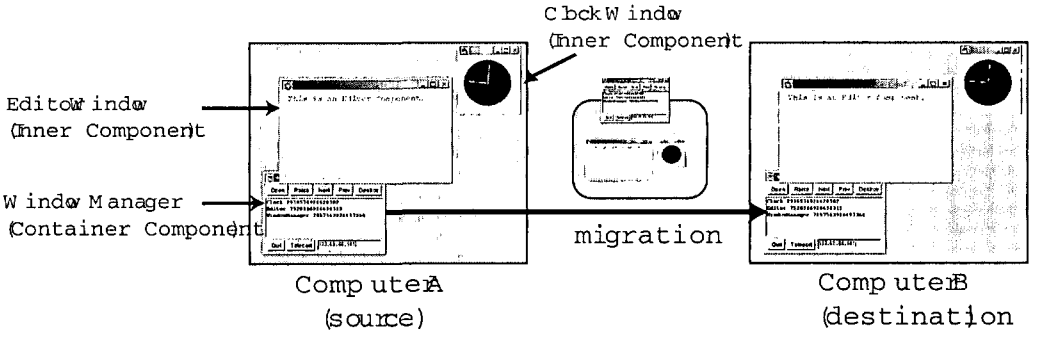


Figure 7: Desktop Teleporting to another Computer.

References

[1] Apple Computer Inc. (1994) *OpenDoc: White Paper*, Apple Computer Inc.

[2] Arnold, K. & Gosling, J. (1998) *The Java Programming Language*, Addison-Wesley.

[3] Baumann, J. Hole, F., Rothermel, K., & Strasser, M., (1999) Mole - Concepts of A Mobile Agent System, *Mobility: Processes, Computers, and Agents*, pp.536-554, Addison-Wesley.

[4] Brockschmidt, K. (1995) *Inside OLE 2*, Microsoft Press.

[5] Cable, L. (1997) *Extensible Runtime Containment and Server Protocol for JavaBeans*, Sun Microsfystems, http://java.sun.com/beans.

[6] Gschwind, T., Feridun, M., & Pleisch, S. (1999) *ADK: Building Mobile Agents for Network and System Management from Resuable Components*, Technical University of Vienna, TUV-1841-99-10.

[7] Hamilton G. (1997) *The JavaBeans Specification*, Sun Microsfystems, http://java.sun.com/beans.

[8] Lange, B. D., & Oshima, M. (1998) *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley.

[9] ObjectSpace Inc. (1997) *ObjectSpace Voyager Technical Overview*, ObjectSpace Inc.

[10] Potel, M., & Cotter, S. (1995) *Inside Taligent Technology*, Addison-Wesley.

[11] Rekimoto, J. (1997) Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments, *ACM Symposium on User Interface Software and Technology (UIST'97)*, pp.31-39.

[12] Satoh, I. (2000) MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System, *Proceedings of International Conference on Distributed Computing Systems (ICDCS'2000)*, pp.161-168, IEEE Computer Society.

[13] Satoh, I. (2000) MobiDoc: A Framework for Building Mobile Compound Documents from Hierarchical Mobile Agents, *Proceedings of Symposium on Agent Systems and Applications / Symposium on Mobile Agents (ASA/MA'2000)*, Lecture Notes in Computer Science, Vol.1882, pp.113-125, Springer.

[14] Satoh, I. (2001) Adaptive Protocols for Agent Migration, *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'2001)*, pp.711-714, IEEE Computer Society.

[15] Sun Microsystems (1998) *The Bean Development Kit*, http://java.sun.com/beans, Sun Microsystems.

[16] Szyperski, C. (1998) *Component Software*, Addison-Wesley.

[17] White, J. E. (1995) *Telescript Technology: Mobile Agents*, General Magic.

Blocks, a component framework with checking facilities for knowledge-based systems

Sabine Moisan[1], Annie Ressouche[1] and Jean-Paul Rigault[1,2]
[1] INRIA Sophia Antipolis, France
Phone: +33 4 92 38 78 47, Fax: +33 4 92 38 79 39
E-mail: {Sabine.Moisan, Annie.Ressouche}@sophia.inria.fr
[2] I3S Laboratory, University of Nice Sophia Antipolis, France
Phone: +33 4 92 96 51 33, Fax: +33 4 92 96 51 55
E-mail: jpr@essi.fr

Keywords: component framework, behavioral model, model checking, artificial intelligence

Received: May 12, 2001

BLOCKS *is an answer to the software engineering needs of the design of knowledge-based system engines. It is a framework composed of reusable and adaptable software components. However, its safe and correct use is complex and we supply formal models and associated tools to assist using it. These models and tools are based on behavioral description of components and on model checking techniques. They ensure a safe reuse of the components, especially when extending them through inheritance, owing to the notion of behavioral refinement.*

1 Introduction

In the design of Knowledge-Based Systems (KBS) more attention has been paid to cognitive issues than to software engineering ones. Yet, *software* quality (reusability, maintenance, evolution, and safety) is also an important issue for such systems. That is why we have developed a generic multi-level approach to KBS development relying on best software engineering practices. A major outcome is a component framework enriched with models and tools enforcing the correct use of the framework.

A Knowledge-Based System basically consists of an inference engine, a knowledge repository (aka Knowledge Base), and a fact base. Each of these three parts is the realm of one particular type (or role) of actor. In this paper we focus on the role of the *designer*, the one who develops KBS engines.

The notion of *KBS generators* (or shells) emerged in the late 80's [17]. A KBS generator addresses a given activity (e.g., diagnosis, classification) but it is domain-independent: its KBS instances apply to various domains (e.g., classification of cardiologic diseases, of astronomic objects, of biological organisms). KBS generators take advantage of the cross-domains similarities by abstracting the common artificial intelligence concepts and by gathering representation techniques within a unique environment.

Whereas generators aim to meet experts or end-users needs (e.g., they help them manage knowledge base evolution and maintenance), they provide little help for the designer as far as Software Engineering is concerned. Therefore we promote generic tools for producing KBS generators. Adding such a level improves versatility but increases complexity. This paper proposes methods and tools to assist the designer in implementing Artificial Intelligence (AI) techniques in an efficient, versatile, reusable, and maintenable way.

To face the corresponding software engineering challenge (essentially a reusability problem), a collection of software engineering best practices have been prescribed: object-oriented modeling (UML) and programming (C++ and Java), component-oriented framework [3, 10], behavioral modeling with associated proofs and simulations. In a KBS, the primary element that is likely to evolve is the inference engine. That is why this paper focuses on the design, simulation, and validation of engines.

In the sequel we first describe our engine design framework, named BLOCKS[1] (section 2). Then we present the static model and the notion of a component in BLOCKS (section 3). Section 4 is devoted to the component behavioral model and the associated verification techniques. We finally discuss the scope and the benefits of our approach (section 5).

2 General Description of BLOCKS

This paper concentrates on BLOCKS which is part of a wider software platform providing designers with a set of generic toolkits. In addition to BLOCKS (components for engine design) the platform offers compiler generators for knowledge description languages, and several libraries (for graphic user interfaces, for knowledge base simulation and verification). The task of the designer is to select, adapt,

[1] Basic Library Of Components for Knowledge-based Systems

and assemble components from these toolkits into a customized KBS generator, which can then be used to develop KBS applications.

The objective of BLOCKS is to help designers create new engines and reuse or modify existing ones without extensive code rewriting. Thus the components of BLOCKS stand at a higher level of abstraction than programming language usual constructs.

The framework consists of around 60 (C++) classes. About a dozen of them implement basic data structures (lists, sets, maps...). The remaining classes are dedicated to knowledge representation artefacts such as the classical AI notions of *frame* and of *rule* [8]. As a matter of example, class `Rule` is composed of a set of conditions and a set of actions that are to be executed when the conditions are true (see figure 1).

The methods of BLOCKS classes are used by the designer to construct new KBS engines. To continue with the same example, class `Rule` sports two fundamental methods: one to test the conditions, the other to execute the actions. Calls to these methods will appear in the code of rule engines. For instance, a classical forward-chaining engine loops over three phases: finding applicable rules (call `Rule::test_conditions`); selecting a rule for execution (conflict resolution specific strategy, written by the designer); execution of the chosen rule (call `Rule::execute_actions`).

The framework is rooted in our extensive experience with designing various KBS engines, for activities as diverse as computer aided design, classification, or planning and in domains as different as civil engineering, astronomy, medicine, finance, etc. This has been the basis for a *domain analysis* that allowed the major concepts of BLOCKS to emerge. A crucial design decision was to determine the proper generality level of the framework components. Too much generality is not suitable for efficiency, whereas too specific components, though easily applicable, are hardly reusable. Our solution was to restrict the range of targeted activities: we choose planning and classification, merely because they are useful in our current applications.

The analysis has been an iterative process with three main steps:

- abstract modeling of existing engines using formalisms such as UML [18]; this led to the definition of the knowledge representation classes;
- completing classes and detailing their behavior; this has been a major step for identifying common concepts and methods behavior, their roles in problem solving, and their organization;
- modeling control to define sequencing of method calls in engines.

BLOCKS is divided into several layers: the *support* layer contains generic and abstract features (abstract classes and methods, and generic functions) useful for any kind of engine. By specializing the classes in the support layer, the designer may define new layers dedicated to specific activities. These layers contain concrete classes, the instances of which will populate the knowledge bases.

3 A Component View of BLOCKS

In BLOCKS we define a component as the realization of a sub-tree of the class hierarchy: this complies to one of Szyperski's definitions for components [20]. At the framework top level, there are presently three such components that the designer may compose or extend. For this to be possible, the designer needs information about component properties. For it to be safe, he or she should commit to some protocol. For forcing it to be safe, we offer automatic proof and validity checking tools.

3.1 Components in BLOCKS

The three high level components are associated with the initial sub-trees of classes `Frame`, `Rule`, and `State`, corresponding to major KBS concepts. Frames describe pieces of knowledge as static structures, composed of attributes which in turn are composed of sub-attributes or "facets" (declarative or procedural). Rules describe pieces of knowledge as dynamic inferences in the form of conditions/actions patterns. States store the history of the problem solving process.

The designer both adapt the components and writes the glue code of engines. To achieve a given strategy he/she will (non-exclusively) use these components directly, or extend the classes they contain by inheritance, or compose the classes together, or instantiate new classes from predefined generic[2] ones. Among all these possibilities, class derivation is certainly the most frequent one. It is also the one that raises the trickiest problems. In the sequel we shall mainly concentrate on it.

Let us continue with our example: the `Rule` class in BLOCKS (figure 1) is composed of conditions and actions which originally do not take into account fuzzy values. Thus, as mentioned in section 2, it can be used by a simple rule engine. To cope with activities requiring fuzziness, the designer must introduce a `FuzzyRule` class as a derivative of `Rule`. Relying on the static information of the class diagram of `Rule` (signatures of methods and associations among classes), the designer obtains the inheritance graph shown on figure 1. But this static information is not sufficient to ensure a safe use of the framework. Indeed, in the example, the designer must also redefine–in a "semantically acceptable" way–methods `test_conditions` and `execute_actions`.

3.2 Protocol to Use the Framework

As previously mentioned, safe use of the framework requires that a *protocol* be specified. This protocol of

[2] "template classes" in C++

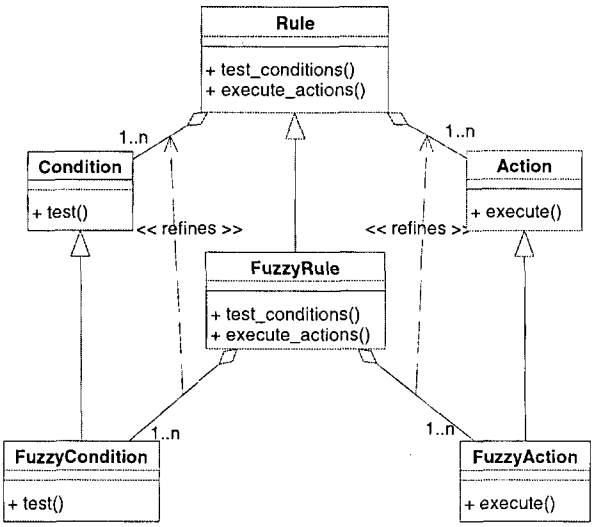


Figure 1: `Rule` and `FuzzyRule` classes: above the original classes, below the derived ones.

use is defined by two sets of constraints. First, a static set enforces the internal consistency of class structures; for instance, in C++, class derivation and composition demand a scaffolding of structure-dependent construction/destruction operations. The static nature makes it easy to generate the necessary information at compile-time.

A second set of constraints describes dynamic method requirements:

1. legal sequences of method calls; for instance, `Rule` requires that `test_conditions` be invoked before calling `execute_actions`;

2. constraints on the operations that a component expects from other components; in the example, the `execute_actions` method expects actions to sport an `execute` method; this is hardly obvious on the class diagram(s);

3. specification of internal behavior of methods;

4. specification of the valid ways to redefine method behavior in derived classes.

These dynamic aspects are more complicated to express than static ones, they are error-prone, and there is no tool (as natural as a compiler for the static case) to handle and check them. While items 1 and 2 can be partially addressed by classical UML models (class diagrams and Statecharts), the last two items are more challenging. We shall propose a solution in section 4.

3.3 Realizing the Component Protocol

To implement the protocol of the previous section BLOCKS applies three non-exclusive techniques.

First, well-known design patterns [9] make it possible to create polymorphic objects (abstract factory, virtual constructor, singleton, prototype), to traverse complex data structures (iterator, visitor), and to implement polymorphic algorithms (strategy). This helps clarify the software architecture, but it seldom is a complete solution.

Second, we use meta-programming [12], namely the OpenC++ meta-object protocol [4, 5]. This helps generate the language-dependent "scaffolding" of constructors requested for frame derivation. It also allows to implement some specific "aspects" [13] of frames such as introspection or persistence. However meta-programmation is complex. Moreover the knowledge about components is external to the components, a risk of inconsistent evolution.

Therefore, third, the knowledge for using, deriving, and composing is embedded into the components themselves. This allows static as well as dynamic verifications relying on this knowledge.

The first two techniques are out of the scope of this paper. We focus on representing and embedding information about behavior of components and methods. There is no complete and consensual technique for this: for instance, in JavaBeans, the embedded knowledge is rather poor; in CORBA, the IDL is external to the components and is not much richer. The next section presents our solution.

4 Behavior Description and Behavior Refinement

In order to reuse BLOCKS components in a safe way, we define a mathematical model providing consistent description of *behavioral entities*. Behavioral entities are whole components, sub-components, or single methods. Such a model complements the UML approach and allows to specify the class and method behavior with respect to class derivation. We also propose a *hierarchical* specification language to describe the dynamic aspect of components both at the class and method levels. Finally we define a *semantic* mapping to bridge the gap between the specification language and its meaning in the mathematical model.

In this paper we just intend to give the flavor of the formal models.

4.1 Mathematical Model of Behavior

We have chosen input/output labeled transition systems [15] as a basis for our mathematical model. Since these systems are a special kind of finite state machines (automata), we shall denote them LFSM for short in the rest of the paper. In our model a LFSM is associated with a behavioral entity; each transition has a label representing an elementary step of the entity, consisting of a *trigger* event (input condition) and the *action* to be executed when the transition is fired.

LFSMs are particularly well suited to check temporal logic properties. Temporal logic easily expresses asser-

tions about behavior. Formulae of this logic concern either the states of the model or its executions[3]. Moreover, tools and proof environments are available to perform temporal logic checking on LFSM [11]. The major drawback of model checking is a possible explosion of the state space. Although some tools use symbolic model checking methods to cope with it, an obvious method to push back the bounds of possibility is to use the natural decomposition of the system. Hence, our specification language provides a hierarchical description of behaviors that allows to merge symbolic and compositional approaches.

We substitute LFSM for regular UML Statecharts to represent the state behavior of a class as well as of a method.

In the object-oriented approach, the static semantics of specialization (aka class derivation, or subtyping, or extension) usually obeys the classical Substituability Principle [14]. To enforce behaviorwise *safe* derivation, the same principle should apply to the dynamic semantics of a behavioral entity–such as either a whole class, or one of its (redefined) methods.

If P and Q are LFSMs denoting respectively some behavior in a base class and its redefinition in a derivative, we seek for a relation $Q \preceq P$ stating that "Q extends P in a safe way". To comply with inheritance, this relation must be a preorder sufficient to capture the notion of "correct extention of behavior".

Q simulates P iff we can build a relation H that relates each state of P to a state of Q so that for two related states p an q, every successor of p is related to some successor of q with a transition bearing compatible[4] labels (trigger/action). The definition of simulation is local since the relation between two states is based only on their successor states. As a result, it can be checked in polynomial time. Intuitively, if Q simulates P then any valid input/output sequence (trace) of P is also a trace of Q. Thus Q can be substituted for P, for all purposes of P. Therefore, the extensions in Q do not jeopardize the behavior of P.

For $\preceq$ we choose the notion of "simulation preorder", i.e., $Q \preceq P$ iff there is a simulation relation H such that $H(q_0, p_0)$, where q_0 and p_0 are respectively the initial states of Q and P. Relation $\preceq$ is a preorder over LFSMs and it preserves satisfaction of the formulae of a subset of temporal logic, expressive enough for most verification tasks ($\forall$ *CTL* [11]). Moreover, this subset has a practicable model checking algorithm.

To capture the notion of safe extensibility for components, we define a relation ($\sqsubseteq$): if A and B are two classes, B $\sqsubseteq$ A iff B derives from A and the LFSM associated with B simulates the one associated with A. The relation is also defined for method behavior: $m \sqsubseteq m\prime$ iff the LFSM associated with m simulates the one of $m\prime$.

[3]Temporal logic is based on first order logic and has specific temporal operators to express properties holding for a given state, for the next state, eventually for a future state, or for all future states. We can also express that a property holds for all the executions starting in a given state or that it exists an execution satisfying a given condition.

[4]Two labels are compatible if they are equal once restricted to the intersection of the LFSM alphabets.

With such a model, the description of behavior matches the class hierarchy. Hence, class and method refinements are compatible and consistent with the static description: checking dynamic behavior may benefit from the hierarchical organization.

4.2 Behavior Description Language

In addition to the previous mathematical model, we propose a specification language. This language, very similar to the *Argos* graphical language [15], is also automata-based. It is easily compiled into finite state machines and it supports existing verification methods and tools. Programs written in this language operationally describe behavioral entities, we call them *behavioral programs*.

Behavioral programs use simple automata as a primitive construct. Labels correspond to input/output *events* which determine how the entity changes its state. The notion of event is abstract; in the language it is just represented by a name and, thus, it may receive various interpretations. For instance, it may be associated with the code of a method or with another behavioral program.

The language defines three main constructs. The first one is *parallel composition* (noted $P \parallel Q$). It is a symmetric operator which behaves as the direct product of its automata operands: transitions triggered by the same input are fired simultaneously and their outputs are unioned. Second, *local event declarations* allow to declare events local to a (behavioral) entity (when a local event is emitted, it can trigger transitions only in its own entity). Parallel composition combined with local event declarations makes it possible to represent communication between subprograms. Third, the *refinement* operator is similar to its Statecharts counterpart (definition of hierarchical states), except that it cannot break the hierarchical structure of programs and states. The states of an entity may be decomposed into behavioral sub-entities. This operator makes it possible to express interrupts, exceptions, and normal termination of (sub)programs.

This language offers a syntactic means to build programs that reflect the behavior of BLOCKS components. Nevertheless, the soundness of this approach implies a clear definition of the relationship between behavioral programs and their mathematical representation as LFSM (section 4.1). Let $\mathcal{P}$ denote the set of behavioral programs and $\mathcal{L}$ the set of LFSMs. We define a *semantic* function $\mathcal{S} : \mathcal{P} \longrightarrow \mathcal{L}$ that is stable with respect to the previously defined operators (local events, parallel, and refinement).

As a consequence, the language exhibits a fundamental *composition property*. This property is the key to simplify model checking. For instance if we have proved that $P_1 \sqsubseteq P_2$, then we can infer that $P_1 \parallel Q \sqsubseteq P_2 \parallel Q$, for any possible Q. Thus, compositionality provides a hierarchical means to verify properties.

4.3 Example: Adding Fuzziness to a Rule Engine

Let us apply the previous model to a simple rule engine, involving classes `Rule` and `FuzzyRule` (figure 1).

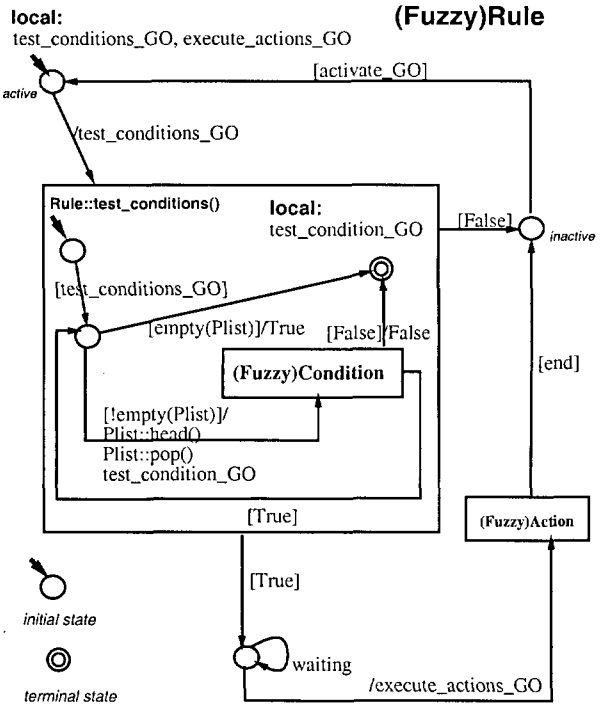


Figure 2: Rule and FuzzyRule behavior description. *Rectangular boxes represent refinement and the keyword* ***local*** *denotes local events. Note that we had to introduce events to trigger method calls (e.g.,* `test_condition_GO`*).*

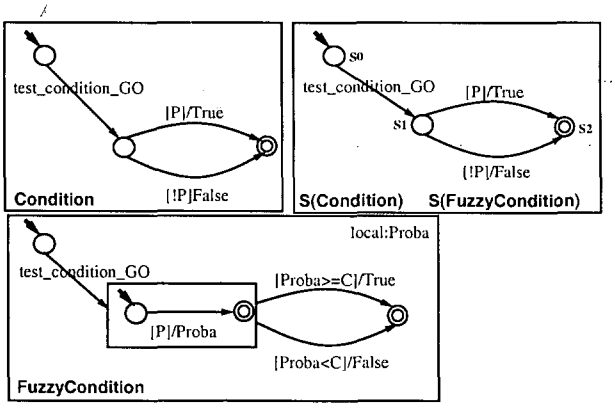


Figure 3: `Condition` and `FuzzyCondition` behavioral programs and semantics. *According to the semantics of refinement and encapsulation, it turns out that* `Condition` *and* `FuzzyCondition` *are associated with identical LFSMs. We recall that $\mathcal{S}$ is the semantic mapping of section 4.2.*

We can show that the behavioral program of `FuzzyRule` is a safe extension of the one of `Rule` (`FuzzyRule` $\sqsubseteq$ `Rule`). First, as can be seen on figure 2, the `FuzzyRule` behavior diagram is identical to the `Rule` diagram except that `FuzzyCondition` is substituted for `Condition` and `FuzzyAction` for `Action`. This diagram expresses the dynamic behavior of class `Rule` with respect to the correct sequence of method calls: `test_conditions` must be called before `execute_actions`.

Second, since `Rule` and `FuzzyRule` are composite classes, we must check the behavior of their parts. Thus we consider classes `Condition` and `FuzzyCondition`. Their behavorial programs are displayed in figure 3. In this simple example, it is easy to see that `FuzzyCondition` $\sqsubseteq$ `Condition`: `FuzzyCondition` derives from `Condition` and `FuzzyCondition` trivially simulates `Condition` (they are associated with identical LFSMs). The same holds for `Action` and `FuzzyAction`.

Hence, according to the composition property, we can deduce that `FuzzyRule` is substituable to `Rule` (`FuzzyRule` $\sqsubseteq$ `Rule`). Compositionality is indeed the way to avoid state explosion in this kind of models.

In this example, the proof is straightforward. In more complicated cases though, the proof may be less obvious, but tools are available to run it automatically.

Relying on this proof, the designer can safely implement the methods; he/she also has to modify the glue code of the engine, especially the conflict resolution strategy (section 2), e.g., to select the rule with the highest likelihood. The resulting rule engine will now accept fuzzy rules[5].

5 Discussion

5.1 Components and Frameworks

Both Software Engineering (SE) and Artificial Intelligence (AI) have an interest in component models. However they have different views on components and, hence, on reusability. SE tools focus on reusing code, analysis and design patterns, or software architectures. Few, if any, existing component frameworks go as far as ensuring correct use through a proof system. On the other hand, several AI approaches have been proposed to reuse *knowledge* components such as abstract problem-solving methods or ontologies [16, 19, 1]. They often manipulate formal descriptions but they usually remain at the knowledge level, thus they do not help producing code.

AI research has already proposed generic tools that cover all steps of KBS design (from cognitive model to implementation or simulation). We can cite DSTM [22] or TASK [21] that are dedicated to KBS design, although with different techniques and approaches. DSTM aims at prototyping a cognitive model before implementing it and, thus,

[5]Of course the other elements of the KBS generator (such as knowledge description language and expert interfaces) must be adapted accordingly: our platform provides the necessary toolkits. By assembling all these elements, the designer produces a new generator. Afterwards, experts can fill in different knowledge bases, in order to produce new KBS instances.

it is more expert-oriented. TASK proposes different languages for the various steps of KBS design, and in particular a formal specification language. Such generic tools are very powerful since they are applicable across domains and activities, but their use may be difficult. Our work follows a similar line, with a stronger software engineering flavor.

5.2 Verification of KBS

In AI, the most common verification adresses the internal consistency of knowledge bases and, of course, our platform provides tools for such verification. Usually, it is on the final KBS that verification is performed. It is too late since, at this time, all the KBS elements (domain knowledge, engine strategy, or even implementation artefacts) have been blended together. Hence, each verification process has to sort out its elements of interest. On the contrary, we promote high separation of concerns, i.e., we separate the engine design phase from the KBS one. The corresponding tools are also separated.

Some systems verify the KBS consistency against its domain and activity models. This verification generally relies on theorem proving techniques, using either an embedded theorem prover as in TASK or applying an external tool like KIV [7]. We have not yet investigated such verifications, but we expect that model checking could also be applied.

The Software Engineering issue of verifying that a KBS properly uses its generator features is often assumed and seldom performed. Our generic approach introduces such a verification. It corresponds to *usage verification* of a complete protocol of use (both static and dynamic properties). For this purpose, we use model checking instead of theorem proving, since it is adapted to our finite state machine model, it can be made automatic, and it can also automatically produce code for refined entities (furthermore this code will be correct, by construction).

5.3 Run-time Verification and Simulation

The designer can use our specification language to describe classes and methods behavior through a dedicated interface. The corresponding programs can serve both formal and practical aims.

On the formal side, the composition property makes it possible to apply model checking techniques in an incremental way. We have experimented with several tools. *EsterelStudio*[6] is a powerful environment to describe, simulate and verify reactive systems. However, its underlying paradigm (the synchrony hypothesis [2]) restricts the type of communication. By contrast, *Ptolemy*[7] is an open (meta-)tool for heterogeneous modeling and simulation. In particular, the user can introduce new models of communication. For this reason, we are going to customize Ptolemy; this will provide a simulation tool and a front-end for model checkers.

On the practical side, as we already mentioned, our specification language can be used to generate (correct) code. The generated code can provide either skeletal implementations of methods, simulation code, and run-time trace facilities. Moreover, by embedding the code of behavioral programs in their components, we can achieve run-time verification.

[6] from Esterel Technologies Company, http://www.esterel-technologies.com

[7] available at: http://ptolemy.eecs.berkeley.edu

6 Conclusion

We have experienced that framework technology can be adapted to the design of knowledge-based system engines. Such an approach allows a significant gain in development time. For instance, two years ago, we had to design a new planning engine [6]. Once the analysis completed, the implementation only took two months (instead of about two years for a similar former project started from scratch) and more than 90 % of the code was composed of existing components. Another experiment (for the classification activity) led to almost the same measurement.

However, the protocol to use the framework is complex and the static modeling (*á la* UML) is not sufficient to prevent the designer from fatal misuse. To this end, we assist the designer by modeling the behavior of components, thus permitting automatic verification during class derivation and composition. The model has also a pragmatic outcome: it allows the simulation of resulting KBS engines and the generation of code, of run-time traces, and of run-time assertions.

This behavioral formalism relies on a mathematical model, a specification language, and a semantic mapping from the language to the model. This lays the foundation for model checking and simulation tools. The model supports multiple levels of abstraction, from highly symbolic (just labels) to merely operational (pieces of code). Moreover this model is original in the sense that it covers both static and dynamic properties of components. To use our formalism, the designer has only to draw simple graphs with a (yet to be) provided graphic interface, oblivious of the underlying models and their complexity.

The same idea could be applied to other component frameworks, outside AI. Our approach gathers techniques from several Computer Science domains seldom intersecting each other: real-time and reactive systems, object-oriented paradigm, and knowledge-based systems. This work can be considered as a successful example of multidisciplinary integration.

References

[1] V.R. Benjamins, B. Wielinga, J. Wielmaker, and D. Fensel. Brokering Problem Solving Knowledge at the Internet. In *EKAW'99, European Knowledge Ac-*

quisition Workshop, volume 1621 of *LNAI*. Springer-Verlag, 1999.

[2] G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stearling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honor of Robin Milner*. MIT Press, 2000.

[3] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson, and M. E. Fayad. Object-Oriented Frameworks: Problems & Experiences. In R. Johnson M. Fayad, D. Schmidt, editor, *Building Application Frameworks: Object Oriented Foundations of Framework Design*. John Wiley, 1999.

[4] J. Cavarroc, S. Moisan, and J-P. Rigault. Simplifying an Extensible Class Library Interface with OpenC++. In *OOPSLA'98, Worshop on Reflective Programming in C++ and Java*, 1998.

[5] S. Chiba. A Metaobject Protocol for C++. In *OOPSLA'95*, volume 30 of *SIGPLAN Notices*, pages 285–299. ACM Press, 1995.

[6] M. Crubézy. *Pilotage de programmes pour le traitement d'images médicales*. PhD thesis, Université de Nice Sophia Antipolis, 1999.

[7] D. Fensel, A. Schönegge, R. Groenboom, and B. Wielinga. Specification and Verification of Knowledge-Based Systems. In *Workshop on Validation, Verification and Refinement of Knowledge-Based Systems, ECAI*, 1996.

[8] R. Forsyth. *Expert Systems : Principles and Case Studies*. Chapman and Hall, 2nd edition, 1989.

[9] E. Gamma, R. Helm, R. Johson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] R. E. Johnson. Frameworks = (Components + Patterns). *CACM*, 10(40):39–42, 1997.

[11] E. M. Clarke Jr., O.Grumberg, and D.Peled. *Model Checking*. MIT Press, 2000.

[12] G. Kiczales, J. de Rivičre, and D. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97*, volume 1241. Springer-Verlag, 1997.

[14] B. Liskov and J. L. Wing. A New Definition of the Subtype Relation. In *ECOOP'93*, volume 707 of *LNCS*, pages 119–141. Springer-Verlag, 1993.

[15] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Composition. *LNCS: Concur*, 630, 1992.

[16] M. A. Musen, S. W. Tu, H. Eriksson, J. H. Gennari, and A. R. Puerta. PROTEGE-II: An Environment for Reusable Problem-Solving Methods and Domain Ontologies. In *IJCAI*, Chambéry, August 1993.

[17] M. Richer. An evaluation of expert system development tools. *Expert Systems*, 3(3):166–182, July 1986.

[18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley 1999.

[19] G. Schreiber, B. Wielinga, R. de Hoog, H. Akkermans, and W. v. de Velde. CommonKADS: A Comprehensive Methodology for KBS Development. *IEEE Expert*, 9(6):28–37, 1994.

[20] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[21] X. Talon and C. Pierret-Golbreich. TASK: from the specification to the implementation. In *8th IEEE Int. Conf. on Tools with Artificial Intelligence*, pages 80–88. IEEE Computer Society Press, 1996.

[22] F. Trichet and P. Tchounikine. DSTM: a Framework to Operationalize and Refine a Problem-Solving Method Modeled in Terms of Tasks and Methods. *Int. J. of Expert Systems With Applications (ESWA)*, 16(2):105–120, February 1999.

A security assurance framework for component based software development

Ashwin Kumar M. V. N.[1], Arun K. Singh and Ramesh Babu S.
Education and Research Department, Infosys Technologies Limited,
Electronic City, Hosur Road, Bangalore – 561 229, INDIA
Phone: +91 80 8520261, Fax: +91 80 8520741
arunks, rameshbabu@infy .com

Keywords: Components, Security, Proof Carrying Code, Aspect Oriented Programming.

Received: June 1, 2000

Commercial-off-the-shelf (COTS) components are black box software products. The absence of their code precludes them from any kind of inspection to certify that the code is safe. This increases the security risk for safety-sensitive applications. The application, before interfacing with COTS component, needs an assurance that it is secure. This paper presents a framework to assure security of components for such applications. This framework uses Aspect Oriented Programming (AOP) paradigm to capture security characteristics of the components and weaves the corresponding security checks into them. It also introduces a novel verification mechanism to ensure that the COTS components are developed as per security contract.

1 Introduction

Software development today is increasingly dominated by the use of generic software components, also known as Commercial-Off-The-Shelf (COTS) components, with some fixed functionalities. Use of such components in the application development significantly reduces time and effort, as there is no need to reinvent the wheel. Notwithstanding these advantages, there lies a great risk in using COTS components in a safety sensitive application. The security risk arises because these components are typically black-box products developed by third parties. For instance, a maliciously written component could silently leak information to the interested parties or write into the local resources. While using these components, the user of the safety-sensitive application needs to address two main aspects – security characterization and security verification.

While security characterisation deals with the properties that a component should possess to be called secure, the security verification deals with the issues regarding how exactly one can implement these properties in that component and be able to reliably check whether it is secure. This paper presents a framework to capture the security characteristics of the components and their verification.

2 Issues in security assurance

In a situation where the components are written by someone and used by someone else, there are many issues related to security assurance. Some examples, which fit this scenario are Applets, ActiveX controls, Java Beans, CORBA objects and so on. While using a component, there is a need to establish trust in some manner.

Security assurance starts with identifying certain security characteristics of the components that can be used to assure trust. Security characterization throws up many issues. At the outset, the granularity level for the security characterization has to be decided. This means taking a decision that the security characteristics need to be inferred from (i) the properties of the component as a whole, or (ii) the properties of the objects in the component, or (iii) the properties of each statement in the object of the component. At each of these levels there can be many properties related to security e.g. read/write/execute access to local resources. Once the level of granularity is decided, the next issue is to identify the type of associated properties. Furthermore, it needs to be decided whether to adopt a Black and White security scheme or shades of Grey security scheme of components. While a Black and White scheme would characterize a component as either fully secure or fully insecure, whereas, shades of Grey scheme will help the designer to support a fine grained view of the security of the component.

Having identified the security characteristics of the components, there can be issues related to developing such components where the desired security characteristics have to be incorporated. For instance, the security checks can be directly embedded into the code. In such a case, there are issues related to identifying the checkpoints in the code, and pondering about the performance measures due to the additional checks.
Having developed a secure component, the next step is to verify the secure behaviour of the component. The first

[1] Summer Intern 2001 from Dept. of Computer Science & Engg., Indian Institute of Technology, Chennai, INDIA.

issue in verification is to identify the source of trust. There can be many scenarios for this. For instance, the verification can be done based on (i) the security policy defined by the code consumer, or (ii) the trust established by the code producer through some mechanism, for example using digital signature, or (iii) the trust established through a third party by using trust certificates. Each scenario brings its own pros and cons.

3 Approaches to assure security of components

A number of solutions have been proposed in the literature to address the above problem and each one has its own advantages and disadvantages. Three widely used solutions are discussed below.

3.1 Running the code in a restricted environment

Here, a restricted environment is created for executing the external/un-trusted code, and the operation carried out by the code is monitored for any "dangerous" operations. An operation is dangerous if, for example, it opens a new network socket or accesses a sensitive part of the file system. Whether an operation is dangerous or not is very subjective in nature and depends on the security needs of the code consumer. This solution approach is also known as the sand-boxing technique [4] and, for example, used by Sun Microsystems to address security concerns of downloaded Applets.

In sand-boxing technique, there is no guarantee with respect to the "security-worthiness" of the code. Since the component is just a black box, the consumer cannot make reasonable estimates about the security-worthiness of the component. The security characterization has to be done by the code consumer to design the sand box.

The disadvantage of the sandbox model is that the security check is not static, i.e. there is no security check done on the code before executing. Due to this, every time the code is used, it should be run through the sandbox. Also, the sandbox technique cannot inherently support a fine-grained security policy. A sandbox technique disallows dangerous operations, as the checking is not static. But there might be cases when some trusted code need to be allowed to perform so called dangerous operations in a controlled way. Trusting some selected code is possible using code signing approach.

3.2 Code signing

The code signing technique [4] involves a trusted third party, also called Certifying Authority (CA), to assure security of the code. In this technique, the code producer puts a digital signature on the code and sends it along with a certificate issued by CA to the consumer. The CA certificate contains information to verify the digital signature and assures the identity of the code producer to the verifier. The whole system centers on the assumption that both the code producer and the code consumer trust a third party or CA. The CA checks the identity of code producer carefully and issues the certificate only if it trusts the code producer.

In this approach, though the certification of the component is done before run time, the certificate gives the assurance about the code producer and not about the code. Hence, the main drawback of this solution is the very foundation it stands on – the notion of trust. The technique requires the code consumer to place complete trust in the CA, and complete trust cannot be placed on CA as after all even the CA can commit mistake. There have been cases when hackers have cheated the CA and obtained the certificates using fake identity (Source www.securitynewsportal.com).

The problems discussed above suggest that the code itself should carry the trust agreed upon by both the consumer and producer without involving a third party. This is the basis of proof carrying code concept.

3.3 Proof carrying code

Peter Lee and George Necula introduced the concept of Proof Carrying Code (PCC) [1]. This concept can be stated in three steps: 1) The code consumer designs a security policy and sends it to the code producer. 2) The code producer develops the code based on the supplied security policy and generates a "proof" for compliance. This proof is sent along with the code to the consumer. 3) The code consumer verifies the correctness of the proof and checks that it is compliant to the supplied security policy. If the proof is correct, the code is deemed security-worthy or else it is rejected.

The "proof" that the code is safe can be considered as some sort of an embedding in the code that helps the code consumer in compliance verification. This technique has both the advantages of being static and also not being dependent on any complex trust relationships as the code itself carries the proof. The verification of the "proof" is done before the code execution. Here the code consumer is not trusting the competence of a CA, but is trusting the "proof" generated using the actual code which qualifies for the competence and the security of the code. The lack of a CA and its related issues with trust makes this technique very attractive.

Peter Lee and George Necula implemented PCC [1] using formal theory. Here, the security characterization was done at a statement level and "proofs" also had been given at the statement level. Defining security policies and "proofs" at statement level for complex components like COTS can be very complex. Next section proposes a framework to address the security assurance problem at the component level.

4 Proposed framework to assure security of components

Clearly, the sandbox and code-signing approaches have limitations. The PCC concept though alleviates these limitations; it works at the statement level. Extending the PCC concept to define and verify the security policy for components poses many challenges. Also, since the security concerns, in general, are crosscutting in nature, it is required to enforce security policy checks throughout the component.

The proposed security assurance framework uses Aspect Oriented Programming (AOP) [2] paradigm to extend PCC concept at the component level. Using AOP, the crosscutting security characterstics can be modularised and specified separately. These characterstics are then weaved in the code. This weaved code serves the purpose of "proof" which can be used for security verification. Before describing the framework, here are a few definitions related to AOP.

4.1 Aspect Oriented Programming (AOP) definitions

Aspect-oriented programming allows capturing of certain global properties of a program and then interleaves them into its executable.

An Aspect, in AOP, is a subprogram to specify some action to be performed at "strategic points" in the code. These strategic points could be before, around or after executing some specific regions in the code. When the code is compiled along with the aspect, the action specified in the aspect is weaved into the code at the corresponding strategic points. An Aspect contains Join Points, Point Cuts and Advices.

A Join Point, in AOP, is a node in the program's runtime object call graph. The various join points are constructor or method calls, constructor or method receptions, constructor or method executions, field gets, field sets and exception handler executions.

A Point Cut, in AOP, is a collection of these Join Points and the values associated with them. Each Point Cut has certain designators to match join points in the execution of the code. Primitive Point Cut Designators are:

```
calls(<return type> classname.funcname(params))
receptions(<return type>
classname.funcname(params))
gets(<return type> classname.funcname(params))
sets(<return type> classname.funcname(params))
handles(ThrowableTypeName)
instanceof(CurrentlyExecutingObjectTypeName)
within(ClassName)
withincode(<return type>
classname.funcname(params))
```

User defined point cuts can be constructed using && (and), || (or) and ! (not) Boolean operations. An example of User Defined Point Cut depicting any line movement function call reception is as follows:

```
pointcut moves():
 receptions(void FigureElement.slide(int, int)) ||
 receptions(void Line.setP1(Point)) ||
 receptions(void Line.setP2(Point)) ||
 receptions(void Point.setX(int)) ||
 receptions(void Point.setY(int));
```

These user-defined pointcuts define the strategic points in a program.

An Advice, in AOP, defines the action to be performed before or around or after a strategic point. An example of advice to set a flag after a move is shown here:

```
static after(): moves() { flag = true; }
```

All these definitions are grouped in a class like structure called the Aspect. An example of aspect to track any moves is the code looks like this:

```
aspect MoveTracking
{
  static boolean flag = false;
  static boolean testAndClear()
  {
    boolean result = flag;
    flag = false;
    return result;
  }
 pointcut moves():
   receptions(void FigureElement.slide(int, int)) ||
   receptions(void Line.setP1(Point)) ||
   receptions(void Line.setP2(Point)) ||
   receptions(void Point.setX(int)) ||
   receptions(void Point.setY(int));
   static after(): moves()
    {
        flag = true;
    }
```

4.2 The framework

The framework comprises of two main parts – a) Security characterization and b) Security verification.

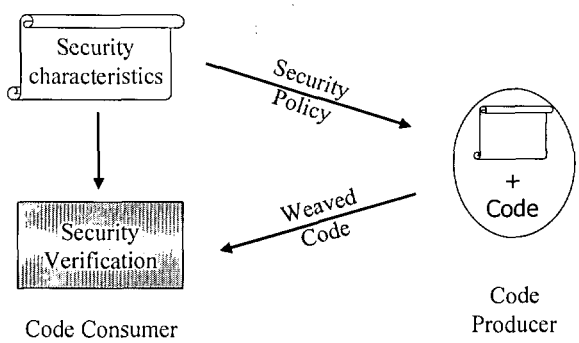


Fig. 4.2: The Proposed Framework

Security characterization of components can be done either by the code consumer or by some standard

organization. In case of standard organization, various levels of security for different applications can be pre-defined and published.

For security characterization, the framework makes use of Aspect as a template to capture the requirements, and makes use of Aspect Compiler to interleave security checks in the component at appropriate points.

For security verification, the framework uses a verification engine (VE). The VE does following things e.g. it checks that the code producer had used the same security policy, which was used to capture security requirements. It checks whether any modifications were done into the component to bypass security checks. It also checks whether any malicious tinkering in security policy itself was done and so forth. If any of these checks fail, the VE stops and prompts an appropriate error message. Once VE verifies the code, it gets executed in the normal fashion.

In order to explain the working of this framework, a prototype implementation has been done. The next section explains the steps involved in the framework by describing through an implementation of this framework.

4.3 A prototype implementation of the framework

For the prototype implementation, Java code and AspectJ, (developed by Xerox Parc) [2], have been chosen. AspectJ is an extension of the Java language incorporating the Aspect Oriented Programming (AOP) principles. There are three steps in the implementation:

Step 1: Define a security policy
Step 2: Embed the security policy into the code
Step 3: Verification engine

Step 1: Define a security policy

Here, the code consumer or the security standard organization identifies those methods that might do dangerous operations. The security policy is described at a method level. The security policy is captured as an aspect, which defines dangerous operations as point cuts and associates an advice for these point cuts.

The security policies, in general, control the dangerous operations done by the program that are normally related to accessing the local resources, which involves use of certain library functions. Since the source codes of these library functions are not accessible, only the "calls" to these functions can be controlled. As a result, mostly the 'calls () pointcut' is used in Aspects.

To define the action to be performed at the calls() pointcut, this prototype implementation uses 'before() advice' only, i.e. all actions defined in the aspect are carried out before invoking the function.

Step 2: Embed the security policy into the code

The code producer obtains the security policy either from code consumer or downloads it from some central repository and writes the code corresponding to the components. After this, the component code and Aspect are compiled using AspectJ compiler (ajc). The compiler does the following:

1 Compiler converts the aspect file into a java class. In this newly created java class, all before() advices are converted into methods and an object aspect$ of the same class type is declared as a member (see Appendix A for a sample code). Since there is no pointcut information preserved in the final class generated from the aspect, the aspect need to be preserved for verification.

2 Next, the compiler weaves the aspect in the code files. While weaving, the function calls that are part of the 'calls() pointcut' are replaced by some new member functions. For instance, if the original function call was f.getAbsolutePath() in class code, and it falls in the jurisdiction of a calls() pointcut, then the function call is replaced by code.getAbsolutePath$call4(f) (See Appendix A). In the new member function, the number indicates the serial number of that function call in the actual code. We call such a function a "tinkered" function. The new member function defined in code class will be having the same name. Within the new member function, all the required 'before advices' will be called.

3 The AspectJ compiler then invokes the javac compiler to generate the aspect and code class files.

The code consumer will use these files to verify that the code is indeed security-worthy. Hence the code producer shouldn't fiddle with the code that the AspectJ compiler provides after aspect compilation and before javac compilation.

Step 3:Verification engine (VE)

This is the final and most crucial step in the implementation. As compared to previous two steps where Aspect and ajc were used to execute the steps, there are no tools available to perform this step. For this, a verification engine has been developed. As shown in Figure 4.3.1,the VE takes the class files corresponding to component and aspect as well as Aspect source file as inputs and provides the verification results for security assurance.

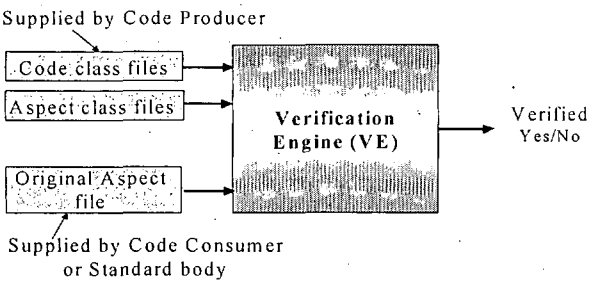


Figure 4.3: Overview of Verification Engine

The verification process can be divided into two parts:

Part I: In the first part, VE checks if the code producer has indeed used the same aspect, which was supplied by the code consumer or standard body. Since the Aspects are defined using before() advice only and these advices are converted into functions, actions performed at pointcuts are actually function calls to these advice functions. To check whether the code producer has used the same advice, it is enough to check whether the aspect returned by the code producer is identical to the original aspect file defined for the code production.

Part II: In the second part, VE checks if the code producer has not bypassed any security check specified by the security policy. After weaving and before the usual compilation with javac compiler, it is possible to introduce a malicious code that will not be subject to the aspect's advice or might remove a call to the advice. Hence at every function call, which is in a pointcut, the verifier has to check if all the advice functions have been called or not. In order to do this, the VE extracts the following information about each method from the class file of the component code:

- Whether it is a "tinkered" method.
- Its signature i.e. if it is an original method or the signature of the method, which was "tinkered" by AspectJ compiler to get this method.
- A list of functions called by in the method.
- The byte code representation of the method.

After extracting this information, the VE does following checks:

Check1: If the method is not a "tinkered" method, check if it is a call to an advice, flag the error and halt.

Check2: If there exist two function calls such that one call is to a function, which is a "tinkered" version of the other, then the verifier halts flagging an error. Such a scenario would mean that there is a function call that is in a pointcut but the corresponding action has been bypassed deliberately.

Check3: If the method is a "tinkered" method, check that two tinkered methods having the same "original" signature (signature of the method which was "tinkered" is the "original" signature) should have the same code. "Tinkered" method will be created only if the original function call was part of some pointcut. Hence for the two "tinkered" methods specified above, the list of advices to be called would be the same.

Check4: Any function call that appears in the non-tinkered form should not be a part of any pointcut. For conducting this check, the original aspect is parsed and corresponding pointcut information is obtained.

Check5: The list of function calls in any "tinkered" method should be the list of advice functions to be called before calling the original method and the original function call.

Check6: Finally the verifier ascertains that apart from the function calls listed above no other instructions are there in a "tinkered" method. Of the above checks made, only Check1, Check4, Check5 and Check6 are truly essential. Other checks have been added to handle the performance issues. If there is malicious intent on the part of the code producer, these checks will not allow code execution.

5 Conclusion and future work

This paper proposes a novel framework for implementing the PCC concept using AOP to assure the security of components. The crucial part in this framework is the verification of the weaved code with respect to initial security policy. A prototype of Verification Engine has been implemented. The VE does the verification based on byte code analysis.

There are some threads that merit future research. Our framework requires the code consumer or a standard body to define the security policy that the code producer has to adhere to. For the latter, it is required to characterize some "standard" security policies and publish them such that any code producer can use them. Defining these standard security policies to represent various levels of security is a very challenging task.

The other thread is to explore on extending to the AspectJ compiler itself to do the VE operations. The VE does operations like pointcut recognition; advice generation etc. These operations could be performed by modifying AspectJ compiler itself. If the aspect-oriented compiler provides a tool, which does the verification, a new verifier need not be designed separately. This can save lots of maintenance problems related to upgrading VE when there is a change in the compiler version or when an aspect-oriented compiler is built for a new language. This could be an important issue to extend this framework to other languages apart from Java.

6 References

[1] Peter Lee and George Necula (1996), *Proof Carrying Code – A Term Report* submitted at CMU-CS-96-165

[2] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, *An Overview of AspectJ*, http://aspectj.org

[3] Qun Zhong and Nigel Edwards (1998), *Security Control for COTS components*, IEEE Computer, Jun 1998

[4] Gary McGraw and Edward W.Felten (1999), Security *Java,* Wiley Computer Publishing.

[5] Tim Lidholm and Frank Yellin (1999), *The Java Virtual Machine Specification, Second Edition*, Copyright @ 1999 Sun Microsystems, Inc.

7 Appendices

Appendix A – A simple file system security using proposed framework

As required by the AspectJ compiler, create a package containing the aspect code. To facilitate specification of pointcuts, the code producer has to put the actual code inside a sub-package. Let the package be 'fileio' then the sub-package holding the code should be 'fileio.Code'

Consider this file system safety policy: "No file operations should be allowed on files existing in the c:\WINNT directory." Here, any file operation would be dangerous if the file buffer is pointing to a valid file in the c:\WINNT directory. Hence, the pointcut defined would be:

```
pointcut Fileio (java.io.File f):
        within(fileio.Code..*)&&calls (* f.*(..));
```

The above definition denotes that the pointcut is a collection of all those join points which denote function calls to the methods of a File object from the code within fileio.Code package (apart from the constructor because the return type is specified as a wildcard).

Note that the calls pointcut descriptor has been used, and not executions or gets. This is because, for the calls pointcut descriptor, code need to be weaved only in the class where the function is called for which we shall have the source code. In executions, gets or sets, code has to be weaved in the function that is called. Here the function called is a library function and usually source code of library functions is not available. Hence, only calls can be used. For our purposes this is enough. Now, we define an advice for this pointcut.

```
before(java.io.File f) : Fileio(f)
{
  if(f.getAbsolutePath().compareTo("c:\\WINNT")==0
  && f.exists() )
        throw new UnknownException;
}
```

All function calls in the pointcut are "dangerous" functions calls. Hence, the code consumer specifies in the security policy (here the aspect) to abort the program by throwing an UnknownException. This advice has been written only for illustration purposes.

As in the pointcut case, it is enough to use only 'before' advice. To simplify the final verification process, 'around' advice is not used.

The aspect that the code consumer thus defines is sent to the code producer.

The aspect defining the security policy would look as follows:

```
package fileio;
public aspect aspectcode
{ {
  pointcut Fileio(java.io.File f):
    within(fileio.Code..*) &&
    calls(public * f.*(..));

    before(java.io.File f) : Fileio(f)
    {
     if((f.getAbsolutePath().substring(0,8).compareTo(
       "c:\\WINNT") == 0) && (f.exists()))
         {
           System.out.println("Forbidden operation.. exiting");
           throw new UnknownError();
         }
     }
  }
```

When run through the AspectJ compiler, the class that is generated using this aspect will be as follows.

```
/*  Generated by AspectJ version 0.8beta4 */
package fileio;
//aspectcode.java:1
public class aspectcode {
//aspectcode.java:4
  public final void before0$ajc(java.io.File f) {
//aspectcode.java:11
    if ((f.getAbsolutePath().substring(0,
       8).compareTo("c:\\WINNT") == 0) && (f.exists())) {
//aspectcode.java:14
      System.out.println("Forbidden operation.. exiting");
//aspectcode.java:15
      throw new UnknownError();
//aspectcode.java:16
    }
  }

  public aspectcode() {
    super();
  }
```

```
public static aspectcode aspect$;
 public static aspectcode aspectOf() {
   return aspectcode.aspect$;
 }

 public static boolean hasAspect() {
   return aspectcode.aspect$ != null;
 }

 static {
   aspectcode.aspect$ = new aspectcode();
 }
}
```

Notice how the before advice has become a function before0$ajc. (0 because it is the first advice in the aspect) Also note that there is no pointcut information existing in the class any more. Hence the code consumer will have to preserve the aspect that had been created for verification.

Consider the following code in the fileio.Code package.

```
package fileio.Code;
import java.lang.*;
import java.io.*;

public class code
{
 public static void main(String[] args)
 {
   File foo = new File("simple");
   File f = new File("c:\\WINNT\\tsoc.log");
   System.out.println(foo.getAbsolutePath());
   System.out.println(foo.getAbsolutePath());
   System.out.println(f.getAbsolutePath());
 }
}
```

The weaved code is as follows:

```
/*  Generated by AspectJ version 0.8beta4 */
package fileio.Code;                                    //code.java:1
import java.lang.*;                                     //code.java:3
import java.io.*;                                       //code.java:4

public class code {                                     //code.java:6
 public static void main(String[] args) {
//code.java:9
   File foo = new File("simple");
//code.java:10
   File f = new File("c:\\WINNT\\tsoc.log");
//code.java:11
   System.out.println(code.getAbsolutePath$call4(foo));
//code.java:12
   System.out.println(code.getAbsolutePath$call5(foo));
//code.java:13
   System.out.println(code.getAbsolutePath$call6(f));
//code.java:14
 }

 public code() {
   super();
 }


 private static String getAbsolutePath$call4(File arg$callThis)
{
   fileio.aspectcode.aspect$.before0$ajc(arg$callThis);
   return arg$callThis.getAbsolutePath();
//code.java:12
 }

 private static String getAbsolutePath$call5(File arg$callThis)
{
   fileio.aspectcode.aspect$.before0$ajc(arg$callThis);
   return arg$callThis.getAbsolutePath();
//code.java:13
 }

 private static String getAbsolutePath$call6(File arg$callThis)
{
   fileio.aspectcode.aspect$.before0$ajc(arg$callThis);
   return arg$callThis.getAbsolutePath();
//code.java:14
 }
}
```

Note how the function calls have changed. Also note that if the function signatures are the same then the bodies of the two new functions made to perform the before advice on those function calls are identical.

The ajc then runs the javac compiler on these classes to get the aspect and code class files. This is then sent to the code consumer.

Now the code consumer gets these files. These files and the original aspect file are fed as input into the verifier. If the verifier passes the code, then the code conforms to the security policy and can be used.

The ABCs of specification: asml, behavior, and components

Mike Barnett and Wolfram Schulte
Microsoft Research
One Microsoft Way
Redmond WA, 98052-6399, USA
{mbarnett, schulte}@microsoft.com

Keywords: Specification, Component, Subtyping

Received: May 25, 2001

We show how to use AsmL, an executable specification language, to provide behavioral interfaces for components. This allows clients to fully understand the meaning of an implementation without access to the source code. AsmL implements the concept of behavioral subtyping to ensure the substitutability of components and provides many advanced specification features such as generic types, transactional semantics, invariants and history constraints.

1 Introduction

There is a broad consensus that a specification of a component's interface must include some way of describing its behavior [26, 32, 36, 43]. Current practice tends towards formal specification of the syntax of the interfaces while using informal natural-langugage descriptions for the semantics. Current theory is based on the idea of design by contract [35], generally using pre- and post-conditions. Previous attempts at describing software also have used algebraic specifications [24].

Interfaces, as they are standardized today, for example using IDL [11], are clearly inadequate for the task of specifying components. It is not enough to provide merely the syntax — signature — for each method contained in an interface. A client who wishes to use a component needs to know the semantics — behavior — of each method. In addition, understanding the relationships between the methods contained in an interface is crucial for the effective use of a component supporting that interface.

We follow the specification taxonomy of Beugnard et al. [9]. A specification is a contract for a software component that describes properties on four levels:

1. *basic*: the syntactic properties of method names, number and type of parameters and very simple semantic properties (e.g., in IDL one can specify whether a parameter that is a pointer can ever be null or not).

2. *behavioral*: the properties that can be specified with pre-conditions, post-conditions, and invariants, including history constraints [34].

3. *synchronization*: properties of component interaction.

4. *quantitative*: all non-functional properties, such as quality of service, response times, throughput guarantees, etc.

Our method for specifications covers only the first three levels; however, we use the term *behavioral* to refer also to the synchronization class of specification.

Our group at Microsoft Research, the Foundations of Software Engineering [16], has developed an executable specification language, AsmL, which is based on the theory of Abstract State Machines (ASMs) (see [22] for an introduction to the notion of ASMs). ASMs allow precise, formal, operational specifications of software systems. AsmL has many important features, among which are generic interfaces and classes, and a transaction-based semantics.

In this paper, we use AsmL to specify the behavioral and synchronization properties of component interfaces, in particular method behavior, interface-wide invariants, history properties, and component composition.

In previous work we used AsmL to write component models by reverse-engineering already existing components [6]. The resulting models provided essentially the identical functionality as the components they were models of. In other words, they modeled the classes that implemented the components. Our concern here is the use of AsmL at the design stage by providing models of *interfaces*. We wish to specify interfaces at their most general level: only the required behavior any component implementing them must have is detailed. The rest is left up to the implementer of the component. An interface model allows clients and implementers to understand the behavior of a software component that correctly implements the interface.

We believe that one should implement a component using classes, i.e., using object-oriented programming, but that the specification should be done at the interface level. The key idea connecting a class to its interface is that the class must be a *behavioral subtype* [34] of its interface. An interface specification describes the minimal behavior expected of all of its subtypes: the behavior of a class can be more constrained than that of its interface.

This paper's contribution is to provide a clean layer in which full behavioral specifications can be written. Specification languages should not be tightly coupled to implementation languages. Precise semantics are crucial for a specification language; implementation languages are oriented towards execution efficiency, as indeed they should be. AsmL has a formal semantics which provides a mathematical foundation for the specification effort. It provides features that aid in the refinement process for developing components that correctly implement their specifications.

The paper is organized as follows. Section 2 provides more detail on exactly what an interface specification looks like in AsmL. Section 3 discusses our notion of refinement and provides an example. Then, in Section 4, we show how to handle component creation and parameterization within AsmL. The next two sections explain how to compose specifications: Section 5 for data-linking and behavior-linking and Section 6 for aggregation and delegation. An overview of similar approaches is discussed in Section 7. Section 8 summarizes and presents limitations and future work.

2 Specifications

We write executable specifications of components in AsmL (the Abstract State Machine Language). AsmL is based on the theory of Abstract State Machines [22]. ASMs are transition systems: their states are first order algebras, that is, interpretations of a functional signature. The transition relation is specified by transition rules (in the sequel simply called rule) describing the modification from one state to the next, namely in the form of guarded updates, i.e., assignment statements that are executed if a boolean condition holds. A sequential run of an ASM program P is a finite or infinite sequence of states $S_0, S_1, \ldots$ where each S_i, $i > 0$, is obtained from S_{i-1} by executing the updates of P at S_{i-1}. The updates generated in a particular step are called the *update set* for the step.

To deal with industrial applications, we have extended ASMs with submachines, objects, exception handling [23] and a very powerful type system (as have others, see [2, 8, 10]). AsmL is freely available for non-commercial research or teaching purposes from our web site [16]. It is currently used within Microsoft for modeling, rapid prototyping, analyzing and checking of APIs, devices and protocols.

We introduce AsmL at the same time as we develop the examples. Only a small subset of AsmL will be used. Our first, very small, example is a specification of a counter interface.

```
interface ICounter
  var ct as Integer = 0
  Counter() as Integer
    return ct
  Increment()
    ct := ct + 2
```

To specify components we use interfaces. Stateful interfaces have member variables, which are also called *model variables*. Model variables are not part of the signature of the interface; they are provided only to give meaning to the method bodies. They are accessible only through the methods defined in the containing interface and its subtypes.

Method bodies in an interface are called *model programs*: they specify the effect that any implementation must respect. Method bodies typically refer to member variables. If a method body updates a member variable, it defines an ASM rule. ASM rules are inherently parallel. This synchronous parallelism comes in handy when specifying independent updates. For example to swap two variables you write:

```
swap()
  x := y
  y := x
```

Sequential composition is the unusual case; to discourage its use, we require a "heavy" notation for it. The sequential AsmL specification for swapping the values of two variables uses an ASM *sub-machine*:

```
swap()
  var t = x
  step x := y
  step y := t
```

AsmL also provides exception handling. Combined with synchronous parallelism, this eases specifications: when an exception is thrown all updates that are produced in the protected block are undone.

The simple transition semantics also simplifies the translation of AsmL rules into predicates. For this purpose we use a slight variation of weakest preconditions. This allows the counter also to be specified in more declarative terms.

```
interface ICounter
  var ct as Integer ct = 0
  Counter() as Integer
    require true
    ensure result = ct and ct = ct'
  Increment()
    require true
    ensure ct' = ct + 2
```

The keywords require and ensure are used for pre- and post-conditions, respectively. Priming (e.g., x') denotes the value in the next state of a run. The keyword result refers to the value returned by the method.

In general, any straight-line method body can be automatically replaced with a pre- and post-condition pair that specifies the same behavior. Loops and recursion require manually-supplied invariants and bounds.

In AsmL a method application changes only those variables that occur in the computed update set; variables not mentioned in the update set are not changed. If a method

body is only described by a pre-/post-condition pair one has to specify explicitly which variables change and which retain their values. If no method body and no pre-/post-condition pair is given, the method can do whatever it wants to, except that is has to respect any interface invariants and constraints (as described in Section 3).

Not only do pre- and post-conditions fail to scale with larger specifications [12], but we have found that real users prefer writing executable specifications instead of pre-postcondition pairs. In AsmL, users can use high-level data structures, users can write nondeterministic specifications, users get atomic transition semantics, and users get ease of reasoning due to referential transparency within each step. Furthermore they can immediately execute the written AsmL specifications.

3 Refinement

A specification is useful only in so far as it defines properties that are true for any implementation. In essence, this is Liskov and Wing's notion of behavioral subtyping [34]: a subtype should always be substitutable for a basetype in all contexts. ASMs can be used in a more general theory of refinement (see e.g. [41]), but for our purposes it suffices to restrict our attention to the $1 : n$ refinements possible in the syntactic framework of classes implementing interfaces. That is, any component implementing an interface must support the syntactic interface; it may do less or more work within each method, but the protocol by which a client uses the functionality is fixed by the syntax of the interface.

There is a well-known problem with specifications and behavioral subytping: a subtype might violate properties of its basetype. For example, in the case of the *ICounter* specification, one cannot reason that the value is always even: as specified, a subtype could increment the counter only by one. Likewise, the counter cannot be assumed to always be positive, a subtype might introduce a decrement method. In order to compensate for this, Liskov and Wing require invariants, which are properties of a single state, and constraints, which in AsmL are properties of consecutive states. For instance, to ensure the two abovementioned properties, we can add to the *ICounter* interface:

```
interface ICounter...
  invariant even(ct)
  constraint ct ≤ ct'
```

The ellipsis (three dots) is part of our literate programming environment [31]; it indicates that this is a continuation of a previous construct.

AsmL also introduces an alternative construct for an operational specification of the permitted state transitions of any method in any subtype: the *others* clause. For instance, to ensure the even stronger property that any other method can increment *ct* only by a multiple of two in the range from 0 to 20 one can write:

```
interface ICounter...
  others(...)
    choose i in {0, 2..20}
      ct := ct + i
```

Any additional method defined in any subtype of *ICounter* will inherit the derived post-condition from the others method.

Our notion of refinement for synchronization properties depends on the concept of a *mandatory call*. Certain method calls in the model programs are identified as communications that any implementation must make during the execution of the corresponding method. All calls to non-local public interface methods are mandatory calls. This includes constructors, see Section 4 for an example. Note that it is the call site that is mandatory, not the method definition. An implementation is free to make additional calls; the model indicates the minimal behavior that must be observed. Thus, we say that an AsmL specification provides a *minimal model* for any implementation.

Classes that implement an interface must be a behavioral subtype of the latter. But the implementation typically chooses a different representation of its fields. Contrary to Liskov and Wing's formulation, we do not require that the class defines an abstraction function (see also Hoare [28]) which relates the concrete state of the class to the abstract state of the interface. In other work [6] we outline a scheme that provides for run-time checking of the subtype relationship without an abstraction function.

However, providing an abstraction function allows for a higher level of verification; AsmL allows a class to define one with the abstraction construct. Suppose that the class that implements the *ICounter* uses a "successor" representation for a counter. Then the abstraction function is just two times its successor representation.

```
class CCounter implements ICounter
  var succ as Integer = 0
  abstraction
    ICounter.ct = 2 * succ
  Counter() as Integer
    return 2 * succ
  Increment()
    succ := succ + 1
```

In this particular example, it is obvious how *CCounter* fulfills the obligations it inherits when implementing the *ICounter* interface. However, in general, abstractions can be much more complicated.

There is no requirement that an AsmL specification be implemented in AsmL. AsmL provides native COM connectivity (as well as COM Automation) and so can be used directly with a component implemented in any programming language.

One interface may also refine another interface, either by extension (see Sections 5 and 6) or implementation. Again, the former interface must be a behavioral subytpe of the

latter interface.

To simplify rapid prototyping, i.e., executing of specifications, AsmL classes don't have to provide their own definitions. As long as interface methods are specified by method bodies, interfaces are executable exactly as written. Thus a class can *reuse* the definitions of the interfaces. The simplest implementation for *ICounter* then becomes:

```
class CCounter reuses ICounter
```

Thus it is often sufficient to close a specification by merely providing a class that reuses the specification.

4 Creation and Parameterization

In this section, we consider two prerequisites for composing interfaces. First, there must exist a way to specify the *creation* of a reference to an interface. An interface is merely a view on a component (namely a particular subset of the component's functionality): what does it mean to have a new reference to one? Second, an interface can be dependent on external values (and/or objects); a completely closed interface is not particularly interesting. The simplest forms of dependency are ones required for *parameterizing* an interface: by type and by value.

Creation. At the interface level there are only interfaces, not components. So if one wishes to access a new interface, where does it come from?

One solution would be to parameterize all interfaces by a *factory interface* that can be used to request the desired interface. A factory interface contains a method which will deliver an interface reference upon request, given some sort of identifier for the interface. But this merely pushes the problem back one level: where does the specification of the factory interface get the interface reference to return? What exactly are the properties of the returned interface?

While factory interfaces are very useful at the implementation level in order to decouple component creation and allow subclassing [17], AsmL interfaces are already expressed at the abstract level. A clearer picture of the desired properties is needed.

When a component is created, there are several assumptions about the resulting reference. Abstracting from the specifics of implementation issues, such as storage allocation, leaves us with the following properties: the component supporting the requested interface

1. should have a unique identity,
2. should not be aliased, and
3. should provide the requested interface in one of its initial states.

Such an interface is guaranteed to be private to the component that is requesting it, unless it explicitly decides to share the reference either by creating aliases or by passing the reference to a third party. For this concept, we use the keyword new with an interface:

```
interface IHistory
  var s as ICounter = new ICounter
```

However, it is important to note that the use of new does not necessarily imply the creation of an object as it would when used on a class. As long as properties 1–3 are ensured for *s*, then it does not matter if a new class object is created by actually calling a constructor or not.

The above example specifies that within the interface *IHistory*, the name *s* refers to an interface *ICounter* on *some* component. Only *IHistory* has a reference to this component. Furthermore, this component is in its initial state, i.e., *s.ct* is equal to zero, and will remain so until changed by a call from within *IHistory*. The fact that the component has a unique identity will be utilized in Section 6.

Sometimes a new interface is requested on an already referenced component, i.e., an existing interface reference. In AsmL that is modeled by a type cast:

```
// ... i is an interface reference to IA ...
let j = i as IB
...
```

This corresponds to using the COM method *QueryInterface* [11]. When the type cast is successful, the requested interface is *not* necessarily in its initial state.

Parameterization. An interface can be dependent on a type, i.e., it can be a generic interface. A generic interface specifies a family of interfaces all of whom instantiate the generic parameter for some particular type. A typical example for a generic interface is the *IState* specification:

```
interface IState(T)
  private var value as T
  Set(v as T)
    value := v
  Get() as T
    return value
```

The *IState* specification says nothing about its initial state; it is also dependent on a value of type *T* that must be supplied to the *constructor* when an instance of *IState* is created. AsmL provides a default constructor that has the same name as the interface. The default constructor takes a parameter for each of the uninitialized member variables:

```
interface IState(T) ...
  IState(v as T)
    value = v
```

In order to be instantiated, the interface *IState* is dependent on both the type parameter *T* and supplied argument for *value*. Note that it is just a coincidence that the type

of *value* is itself *T*. Multiple constructors with different parameter lists are also allowed.

The visibility attribute private on *value* means that it may not be modified by a method within any subtype. Therefore the only way to modify *value* is to call *Set*. This guarantees the property that once a client calls *Set*, *value* will remain unchanged until the next call to *Set*. In other words, any component implementing *IState* will act like a program variable.

5 Linking Specifications

While it is important to be able to specify interfaces in isolation, true component-oriented programming can be realized only when sub-units are composed to make larger units. This implies that we must be able to compose interfaces as well, since the specification for the composition of two components should be the composition of their individual specifications.

5.1 Data Linkage

Linking two specifications through shared data — state-coupled specifications — allows for multiple viewpoints on the same component, while ensuring that the component stays in a consistent state. This represents a common pattern; our example uses the idea of different units for a single measurement [20]. For instance, suppose there are two interfaces.

```
interface IMetricLength extends IState⟨Integer⟩
  IMetricLength() extends IState(0)
interface IUSLength extends IState⟨Integer⟩
  IUSLength() extends IState(0)
```

The specification for *IMetricLength* implicitly keeps *value* in metric units, e.g., centimeters. Meanwhile, the specification for *IUSLength* implicitly keeps *value* in inches. Note that neither interface is parameterized: the generic parameter *T* from *IState* has been instantiated to *Integer*. Also, the explicit constructors take no arguments. But they call the constructors of the interface they are extending; the initial state is thus fully determined.

Suppose we would like to specify a component that provides *both* interfaces with a consistent shared value. Whatever changes are made through one interface should be reflected in the other interface. This is easily specified via a *linking invariant* which constrains any implementation to meet this condition. Fig. 1 shows the structure of the composition.

```
interface IMetricAndUSLength
    extends IMetricLength and IUSLength
  invariant
    IMetricLength.value * 2.54 = IUSLength.value
```

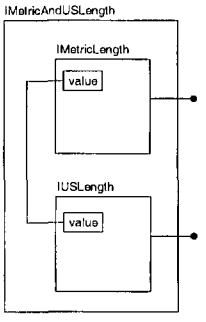


Figure 1: Linking two interfaces by shared data

A crucial feature of AsmL is that all methods and member variables from inherited interfaces are kept distinct. The interface *IMetricAndUSLength* does not identify the methods *Get* and *Set* from the two interfaces; the combined interface has all four methods. AsmL, just as C# [25], does *not* fold methods with the same name and signature when extending multiple interfaces. This is especially important for generic interfaces. Java [19], for instance, is unable to keep the methods distinct.

The behavior of any component implementing the interface *IMetricAndUSLength* must respect the invariant (as well as the individual behaviors specified in each interface). How it does so is left up to the component; one way is to keep *value* in one unit and converting it for the other interface:

```
class CMetricAndUSLength
    implements IMetricAndUSLength
  var metricValue as Integer = 0
  abstraction
    IMetricLength.value = metricValue
    IUSLength.value = metricValue / 2.54
  IMetricLength.Set(v as Integer)
    metricValue := v
  IMetricLength.Get() as Integer
    return metricValue
  IUSLength.Set(v as Integer)
    metricValue := v * 2.54
  IUSLength.Get() as Integer
    return metricValue / 2.54
```

This example differs from the traditional Observer pattern [17] in that both of the original interfaces are *peers*; neither is distinguished as the subject holding the "correct" value (although the component decided to implement it that way).

5.2 Behavior Linkage

In this section, we present an example of two components that are coupled through their interacting behaviors instead of through shared state. We use the Observer pattern [17] which involves two components: a subject and a set of ob-

servers, called *views*.

The *IView* interface is trivial: it contains a method *Update* that is to be called by the subject, and a method for registering the subject with the view so it has access to the subject.

```
interface IView⟨T⟩
  var subject as ISubject⟨T⟩
  Update()
    // behavior goes here, to be defined by subtype
  SetSubject(s as ISubject⟨T⟩)
    subject := s
```

The subject holds some state; whenever the state is changed, each view is notified. This is a generalization of the Reader/Writer paradigm. The specification of a subject is an extension of *IState* that has methods for adding, removing, and alerting views:

```
interface ISubject⟨T⟩ extends IState⟨T⟩
  var views as Set⟨IView⟨T⟩⟩ = {}
  Set(val as T)
    step base.Set(val)
    step Notify()
  private Notify()
    forall v in views
      v.Update()
  Attach(v as IView⟨T⟩)
    views += { v }
  Detach(v as IView⟨T⟩)
    views -= { v }
  others(...)
    ensure value = value'
```

There are three interesting properties that this specification prescribes for any implementation:

1. A subject calls the *Update* method of each view whenever its *Set* method is called. Because *Update* is a public interface method, this call is a mandatory call. An implementation is free to call *Update* more than once, perhaps for fault-tolerance purposes.

2. Views are synchronized with subjects. That is, all views receive a notification with the subject in the same state. This is because the forall loop used within *Notify* is a parallel loop.

3. A view can perform any behavior within its *Update* method. Obviously, it would be unwise to call the subject's *Set* method: allowing the state to change during a callback is known to create problems [43]. The specification can easily be modified to disallow it.

Because of the others clause, no subtype of *ISubject* is allowed to add a method that alters *value* other than by calling *ISubject.Set*. This may be too restrictive; one can specify instead a constraint that connects state changes to *value* with calls to *Update* for each view.

The method *Notify* is marked private to emphasize that it is not a mandatory call. It is only the call to *Update* during the execution of *Set* that must be observable.

6 Aggregating Specifications

In addition to linking interfaces, we use AsmL to define interfaces that re-use existing behaviors to create new functionality. This explores another way of composing specifications which can be seen as aggregation or delegation depending on the details of how it is specified.

We take the example of a radio button group in a graphical user interface from [26]. A radio button group is a set of radio buttons that operate in a mutually-exclusive manner. At most one of them can be selected at any one time; selecting one radio button unselects all of the others in the group. Each button in the group must display itself appropriately as either selected or not.

A radio button group can be seen as an example of reusing the Subject/View contract (i.e., the Observer pattern [17]): each radio button is a view on a subject whose state reflects which button is currently selected. To begin the specification, we first specify the behavior of buttons in general.

6.1 Buttons

We model a button as a user-interface element that has a text label and allows the user to select it, say by clicking on it with the mouse.

```
interface IButton
  var label as String
  var chosen as Boolean
  GetLabel() as String
    return label
  SetLabel(s as String)
    label := s
  Select()
    choose b in { false, true }
      chosen := b
```

Of course, the interface would have additional methods relating to its size, color, etc.

A checkbox button acts as a toggle: clicking it reverses its current state.

```
interface ICheckBoxButton extends IButton
  Select()
    chosen := not chosen
```

A radio button, by way of contrast, is idempotent: clicking on it sets it to true. The only way to unselect it is to select another radio button in the same group.

```
interface IRadioButton extends IButton
 Select()
  chosen := true
```

A single radio button may seem useless, but could be used for signing a document or some other irreversible operation.

6.2 ButtonView

A radio button, as specified in *IRadioButton*, is not immediately composable into a group. As stated, the interface does not provide any functionality for synchronizing its state with other buttons in the same group. This clearly separate behavior can be added in a modular fashion.

We *extend* the *IRadioButton* interface with an *IView* interface. It describes a button that just behaves like a radio button, but responds to a new input notifying it that some state has changed somewehere else.

```
interface IRadioButtonView extends IRadioButton
        and IView⟨IRadioButton or Undef⟩
 // explicit constructor
 IRadioButtonView(s as String)
  extends IRadioButton(s, false) and IView(Undef)
 Update()
  chosen := subject.Get() = me
  // redraw appropriately...
 Select()
  base.Select()
  subject.Set(me)
```

The explicit constructor initializes the button to be unselected. In addition it initializes the view's subject to be undefined. Note that in AsmL reference types don't contain a null value. But disjunctive types, here exemplified by the type *IRadioButton* or *Undef* give you the flexibility to add *Undef* when needed. The keyword base refers to the immediate supertype, in this case *IRadioButton*.

Given this interface, it is now easy to define the behavior of a radio button group.

6.3 ButtonGroup

The requisite behavior of having the radio buttons be mutually exclusive is achieved by wiring the augmented radio buttons from Section 6.2 together with a component that implements the *ISubject* interface into a Subject/View relationship.

```
interface IRadioButtonGroup
 var bs as Set⟨IRadioButtonView⟩
 var subj as ISubject⟨IRadioButton or Undef⟩
```

Figure 2 shows the structure of the interface, although not the multiplicity of views.

A radio button group has its own interface: there are operations that make sense for a button contained in a group,

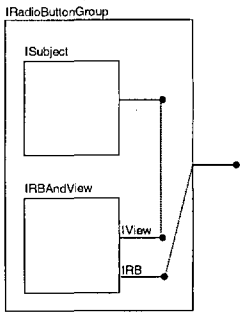


Figure 2: The *IRadioButtonGroup* interface

but not for the collection as a whole and vice versa. Selection of one button in the group is specified by delegating the *Select* call to the appropriate button.

```
interface IRadioButtonGroup ...
 Select(s as String)
  choose b in bs where b.GetLabel() = s
   b.Select()
 ... other methods ...
```

The constructor for *IRadioButtonGroup* takes a set of labels for the radio buttons to be generated, generates the subcomponents and wires them together.

```
interface IRadioButtonGroup ...
 IRadioButtonGroup(ls as Set⟨String⟩)
  subj = new ISubject(undef)
  bs = { new IRadioButtonView(l) | l in ls }
  forall b in bs
   b.SetSubject(subj)
   subj.Attach(b)
```

Informal Reasoning. Now that the composition has been created, it can be reasoned about. Here we give an informal outline of *IRadioButtonGroup*'s correctness for maintaining the mutual exclusion of a selected button.

The only external action that can cause an update is for one of the radio buttons b_1 to b_n in the group to have their *Select* method called. Without loss of generality let's assume that b_1 is selected. This, in turn, will cause b_1.*IRadioButtonView.Select* to be called, which will call b_1.*IRadioButton.Select*. So b_1.*chosen* becomes true.

The button b_1 will also call $Set(b_1)$ on the shared subject. *First*, its *value* becomes b_1. *Next*, the shared subject will call back to every button in the group via *IRadioButtonView.Update*. For every button the *Get* method called on the shared subject will return b_1 — this is the value that was just stored. For b_1 this generates another update of b_1.*chosen* to true; this is a non-conflicting update. In contrast, the *chosen* field of the buttons $b_2 \ldots b_n$, become false, since b_1 is different from any of $b_2 \ldots b_n$.

As a result, we are guaranteed that the group makes an atomic step which preserves the property that at most one button can be selected at any time.

Given AsmL's transactional semantics, it is possible for two buttons to execute their *Select* methods in the same step. Each method will cause an update in the subject for two different values (the value of me for each of the buttons). These updates are *conflicting*; AsmL's runtime checks for different values being assigned to the same location at the end of each step and will signal an exception.

7 Related Work

As long as there have been programmers, there has been concern with the meaning of the artifacts they create by formally specifying the programming process, e.g. [27]. Here, we concentrate specifically on work involving components.

There is a long tradition within the object-oriented community that is concerned with specification, whether formal or not. Meyer, of course, is famous for his ideas on *design by contract* [35]. Over a decade ago, Helm et al. [26] pointed out the necessity for *contracts* and how they can be used as a structuring concept for specifications, but their contracts were a) not executable, and b) confused the *wiring* of components with the specification of their *interaction*. America [1] did some of the early work on behavioral subtyping. The most standard formulation of behavioral subtyping follows that of Liskov and Wing [34]. Most of this work used only pre- and post-conditions for methods, or did not consider using a separate specification language.

Leavens and Dhara provide specifications for Java components using a language called JML [33]. Like us, they insist on behavioral subtyping as a refinement notion and also use *model programs* in addition to pre- and post-conditions. However, their work is limited to Java programs; AsmL can be used in conjunction with any implementation language. They make the distinction between *strong* and *weak* subtyping; we restrict our attention to strong subtyping since AsmL does not prohibit aliasing.

Besides JML, there has been a lot of work on using assertions to specify Java interfaces, e.g., Contract Java [15], iContract [13], jContractor [30], and Jass [7] all implement various schemes to implement design by contract for Java programs. JISL, the Java Interface Specification Language [40], translates and inserts specifications into Java programs. It uses pre- and post-conditions and is used to primarily specify and check frame properties.

Edwards [14] uses specifications for components to generate wrapper components that check the pre- and post-conditions. An abstraction function is required because the conditions are expressed in terms of abstract values. But without model programs, synchronization properties cannot be specified.

Soundarajan and Tyler [42] use trace variables in specifications to record method calls in order to reason incrementally about subtypes. Their trace variables are similar to our mandatory calls, but they also do not have model programs.

Jonkers [29] has interface specifications that are not executable; he also does not insist on absolute rigor in a specification. But his ideas of how to specify interfaces are very similar to ours.

The theoretical background for component specification is mostly based on the refinement calculus by Back and Wright [4] and Morgan [39]. Constructs for object-oriented programming are added to a notation for sequential computing and class refinement is defined such that it respects supertype behavior [3]. To declare a class as a subtype of another means to do a proof in the refinement calculus that the predicate transformer semantics of the class hold the correct relationship with those of the superclass. However, there does not seem to be a concern with directly executing specifications. Sekerinski et al. have explored the restrictions on component-oriented programming that are needed in order to be able to prove refinement in the presence of recursive re-entrance [37]. They have also done a small case study of proving the correctness of Java Collections Frameworks [38]. They extend Java with a specification language and claim that it has a formal mathematical foundation: "every executable statement of the Java language... that we use has a precise mathematical meaning". We take that to mean that only a subset of Java is used.

8 Conclusions

The need for behavioral specifications is widely recognized, especially in component-oriented programming. AsmL provides an industrial-strength tool for writing such specifications. It provides all of the features necessary to express the properties needed for behavorial subtyping.

AsmL is agnostic with regards to verification technology. An AsmL specification can be subjected to analysis with a variety of formal methods, for instance, a refinement calculus proof.

The executability of AsmL specifications opens possibilities that go beyond those traditionally associated with specification languages. A formal specification is the boundary between the informal understanding of a system and its digital incarnation. At the design stage, exploration of the specification provides insight and feedback about the appropriateness of the formalization. During the coding process, the specification can be used, in special domains, to derive test cases and perform conformance testing [18, 21]. An executable specification allows conformance checking, i.e., assertion monitoring, to ensure that an implementation's behavior is allowed by the specification [5, 6]. Furthermore, AsmL's COM connectivity means that it can be used in a language-neutral setting: any language can be used to implement the specification.

There are many areas that need to addressed in future work. For example, adding automatic support to enforce the kind of restrictions needed for refinement proofs [37] or other proof tools.

Acknowledgements

We wish to thank the rest of our research group in Microsoft Research, FSE. Clemens Szyperski, Egon Börger, and Crispin Goswell reviewed earlier drafts of this work and made many useful suggestions.

References

[1] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, NY, June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[2] M. Anlauff. XASM – An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer-Verlag, 2000.

[3] Ralph Back, Anna Mikhajlova, and Joakim von Wright. Class refinement as semantics of correct subclassing. Technical Report 147, Turku Centre for Computer Science, December 1997. Available from `www.tucs.abo.fi` at `/publications/techreports/TR147.html`.

[4] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[5] Mike Barnett, Lev Nachmanson, and Wolfram Schulte. Conformance checking of components against their non-deterministic specifications. Technical Report MSR-TR-2001-56, Microsoft Research, June 2001. Available from `http://research.microsoft.com/pubs`.

[6] Mike Barnett and Wolfram Schulte. Spying on components: A runtime verification technique. In *Workshop on Specification and Verification of Component-Based Systems, OOPSLA 2001*, pages 7–13. Published as Iowa State Technical Report #01-09a, October 2001.

[7] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass — Java with Assertions. Available from `http://semantik.informatik.uni-oldenburg.de` at `~jass/doc/index.html`.

[8] H. Baumeister and A. Zamulin. State-Based Extension of CASL. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods (Proceedings of IFM 2000)*, volume 1945 of *LNCS*, pages 3–24. Springer, 2000.

[9] Antoine Beugnard, Jean-Marc Jézéquel, Nöel Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–44, July 1999.

[10] E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer, 2000.

[11] Don Box. *Essential COM*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1998.

[12] Martin Büchi and Emil Sekerinski. Formal methods for component software: The refinement calculus perspective. In *Proceedings of the Second Workshop on Component-Oriented Programming (WCOP)*, June 1997. Available from `ftp://ftp.abo.fi` at `/pub/cs/papers/mbuechi/FMforCS.ps.gz`.

[13] A. Duncan and U. Hölze. Adding contracts to Java with handshake. Technical Report TRCS98-32, University of California at Santa Barbara, December 1998.

[14] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2), 2001.

[15] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *OOPSLA 2001*, pages 1–15. ACM SIGPLAN, September 2001.

[16] Microsoft Research Foundations of Software Engineering, 2001. `http://research.microsoft.com/fse`.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[18] Uwe Glässer, Yuri Gurevich, and Margus Veanes. Universal plug and play machine models. Technical Report MSR-TR-2001-59, Microsoft Research, June 2001. Available from `http://research.microsoft.com/pubs/`.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[20] Crispin Goswell, 2001. Personal communication.

[21] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Conformance testing with abstract state machines. Technical Report MSR-TR-2001-97, Microsoft Research, October 2001. Available from `http://research.microsoft.com/pubs`.

[22] Yuri Gurevich. Evolving algebra 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, Oxford, UK, 1995.

[23] Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Toward industrial strength abstract state machines. Technical Report MSR-TR-2001-98, Microsoft Research, October 2001. Available from `http://research.microsoft.com/pubs`.

[24] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[25] Anders Hejlsberg and Scott Wiltamuth. C# language specification, version 0.22. Available at `http://msdn.microsoft.com/library/default.asp`.

[26] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented system. *ACM SIGPLAN Notices*, 25(10):169–180, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[28] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[29] H.B. Jonker. Ispec: Towards practical and sound interface specifications. In *IFM'2000*, volume 1954 of *LNCS*, pages 116–135, Berlin, Germany, November 1999. Springer-Verlag.

[30] Murat Karaorman, Urs Holzle, and John Bruno. jContractor: A reflective Java library to support design by contract. Technical Report TRCS98-31, University of California, Santa Barbara. Computer Science., January 19, 1999.

[31] Donald E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, May 1984.

[32] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[33] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[34] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[35] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[36] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[37] Leonid Mikhajlov, Emil Sekerinski, and Linas Laibinis. Developing components in the presence of re-entrance. Technical Report TUCS-TR-239, TUCS - Turku Centre for Computer Science, February 1999.

[38] Anna Mikhajlova and Emil Sekerinski. Ensuring correctness of Java Frameworks: A formal look at JCF. Technical Report TUCS-TR-250, TUCS - Turku Centre for Computer Science, March 1999.

[39] Carroll Morgan. *Programming from Specifications*. Prentice Hall International, Hempstead, UK, 1990.

[40] P. Müller, J. Meyer, and A. Poetzsch-Heffter. Making executable interface specifications more expressive. In C. H. Cap, editor, *JIT '99 Java-Informations-Tage 1999*, Informatik Aktuell. Springer-Verlag, 1999. Available from `http://www.informatik.fernuni-hagen.de` at `/pi5/publications.html`.

[41] G. Schellhorn. *Verification of Abstract State Machines*. PhD thesis, Universität Ulm, Ulm, Germany, 1999. Available from `http://www.informatik.uni-ulm.de` at `/pm/mitarbeiter/gerhard/`.

[42] Neelam Soundarajan and Benjamin Tyler. Testing components. In *Workshop on Specification and Verification of Component-Based Systems, OOPSLA 2001*, pages 1–6. Published as Iowa State Technical Report #01-09a, October 2001.

[43] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

Towards a rigorous and effective functional contract for components

F. J. Galán Morillo, V. Diaz and J. M. Cañete Valdeón.
Dept. of Languages and Computer Systems. Faculty of Computer Science. Av. de Reina Mercedes s/n. 41012. Sevilla.
Phone: 34 95 455 27 73, Fax: 34 95 455 71 39
E-mail: galanm@lsi.us.es

Keywords: abstract data type, component, design by contract, correctness, formal specification, constructive specification

Received: May 11, 2001

The abstract data type (ADT) is the basis for the information-hiding design philosophy that makes software easier to analyze and understand, and that support maintenance and reuse. For these reasons, ADTs can be used to write contracts between specifiers that describe type requirements in an abstract and declarative way and programmers that implement components for these requirements. The construction of components from software specifications could not be a systematic activity if the specification language has powerful abstraction and expression capabilities. Therefore, it is very important to study different forms of type specification in order to understand its programming repercussions. The form of specification is essential to consider a type specification as a rigorous and effective functional contract for components between a specifier and a programmer.

1 Introduction

The abstract data type (ADT) is the most promising programming idea to support software engineering. It is the basis for the information-hiding design philosophy that makes software easier to analyze and understand, and that support maintenance and reuse. There are several formal specification techniques and accepted theories of ADT correctness (Ehrig & Mahr 1985, Ehrig & Mahr 1990, Wirsing 1990). On the other hand, different methods have been proposed to the elaboration of a program in some systematic manner starting from a specification. In the early days, imperative program derivation was one of the first area of active research, mainly focusing on manual program derivation (Dijsktra 1976). Also, other programming paradigms were targets of derivation methods (i.e., functional and logic programming paradigms (Deville & Lau 1994). Recently, synthesis research is focusing mainly on automatic or semi-automatic methods (Fribourg 1993, Billington & Dromey 1996, Flener 1995, Wiggins et al. 1992, Avellone et al. 1999). However, these methods are not yet sufficiently applied in industry.

The ADT can be considered as a contract between a specifier which describes type requirements in an abstract and declarative way and a programmer which implements these requirements. The contract has two opposite faces. From a specifier point of view, expressive languages are needed in order to construct specifications in a effective way. However, from a programmer point of view, the implementation model imposes many restrictions and then programming from abstract and declarative specifications could be an ineffective activity. Balancing these opposite points of view is needed in order to get effective contracts (Galán et al. 1999). Type specifications can be done in many different forms. Some of these forms can be classified as non-constructive forms. There is not a clear relationship between non-constructive types and their respective implementations. Figure 1 represents an example of a non-constructive type. From a programmer point of view, such characterizations represent non effective contracts because:

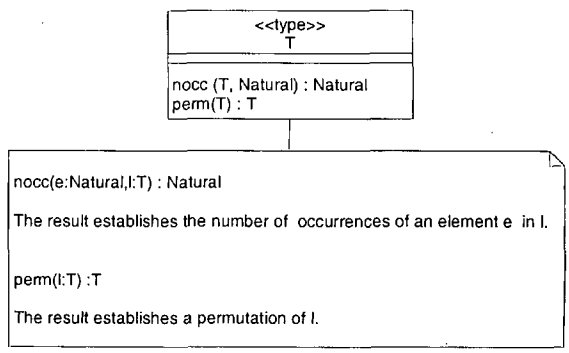


Figure 1: Non-constructive type specification.

1. The programmer could consider that this kind of specifications does not assist him to implement a type:
   (a) Which data models has the specifier in mind? The specifier supposes that the name *T* is sufficient to characterize data models. In this way, the specifier consider a default knowledge in the contract but, probably, this knowledge is not considered by the programmer.
   (b) The programmer can consider that some operation has a non adequate characterization. For example, in the *perm* operation, what does "*The re-*

sult establishes a permutation of l" mean?. Probably, the programmer needs a more detailed explanation in order to understand the semantics of *perm*.

2. In addition, the programmer can consider that this kind of specifications does not assist him to test implementations because he cannot construct tests without considering his own data representation decisions.

Therefore, it is important to search for new kinds of type specifications in order to write rigorous and effective functional contracts for components between specifiers and programmers.

The work is organized as follows. Section 2 establishes a loose classification for different forms of operation specification and their repercussions from a programmer point of view. Section 3 analyzes the reasons why a type specification considered as a rigorous contract is difficult to carry out. Section 4 establishes a specification discipline to overcome the problem of writing formal and effective type contracts. Finally, we establish the conclusions.

2 Preliminary Definitions

This section establishes a definition for Abstract Data Type (ADT). A loose classification for different forms of operation specification is made in relation to their repercussions from a programmer point of view.

A *Data Type* is a collection of data domains, designated basic data items, and operations on these domains such that all data items of the data domains can be generated from the basic data items by use of the operations. Moreover the data domains are assumed to be countable.

An *Abstract Data Type* (ADT) T is a class of data types which is closed under renaming of data domains, items and operations and hence independent of representation.

A *Functional Part of a Component C* for an abstract data type T is a realization (implementation) of (the semantics of) T.

We say that $Spec_{Op}$ is a *pre/post specification* for an operation Op in an ADT T iff it describes the behavior of Op in a declarative pre/postcondition style. $Spec_{Op}$ is called *constructive* if there exists a clear relationship between $Spec_{Op}$ and its implementation. $Spec_{Op}$ is called *recursive* if instances of Op are used in $Spec_{Op}$ postcondition.

An ADT T is called *non-constructive* if it does not have any data information or it has some non-constructive operations.

3 Towards Effective Contracts

From a programmer point of view, the type specification in figure 1 represents a loose contract because there is not data information and there is not a clear relationship between operation specifications and their respective implementations. A more constructive description would be desirable

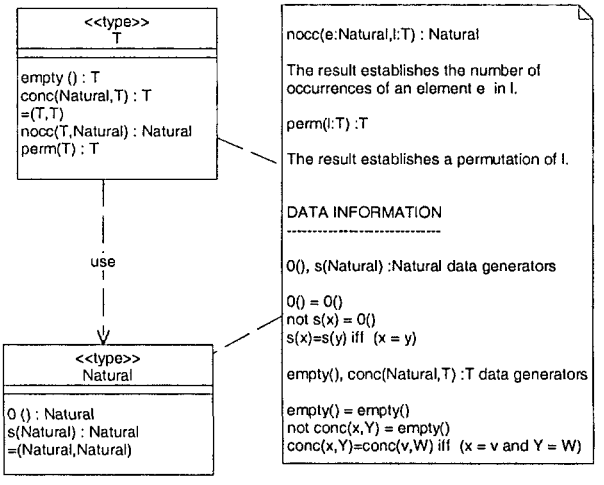


Figure 2: Reachable type.

for a programmer. This section analyzes the reasons why a type specification considered as a rigorous contract is difficult to carry out.

3.1 Reachable Types and Testing

It is important, from a programmer point of view, that specifications assist in testing activities. Specifications such as the one in figure 1 do not assist in such sense because programmers cannot construct tests without considering their own data representation decisions. Hence, reachable types constitutes the first specification restriction needed to obtain types amenable to be tested. Reachable types are obtained by adding a set of function symbols for generating data.

A type T is *reachable* iff each element of the domain of T is represented, at least, by a ground term (i.e., term without variables). The set of function symbols in T will act as data generators.

In addition, an identity relation $=$ between data terms is included for each type specification. These extensions are important in order to avoid rigid specification contexts and to maintain the same (abstract) data model in the minds of specifiers and programmers.

For example, in figure 2, we shows a reachable type T. Data information in T is represented by data generators $\{empty, conc\}$. Therefore, and considering this information, a programmer will be able to define tests for operations in T.

3.2 Understanding Operation Specifications

An important question here is to define the reasons why a specification is difficult to program. For example, $Spec_{perm}$, in figure 1, is difficult to program because its semantics is established by the phrase "*The result establishes a permutation of l*". But the name of the operation coincides with this explanation and there is not any other interesting information. Thus, we conclude that $Spec_{perm}$ is not effective

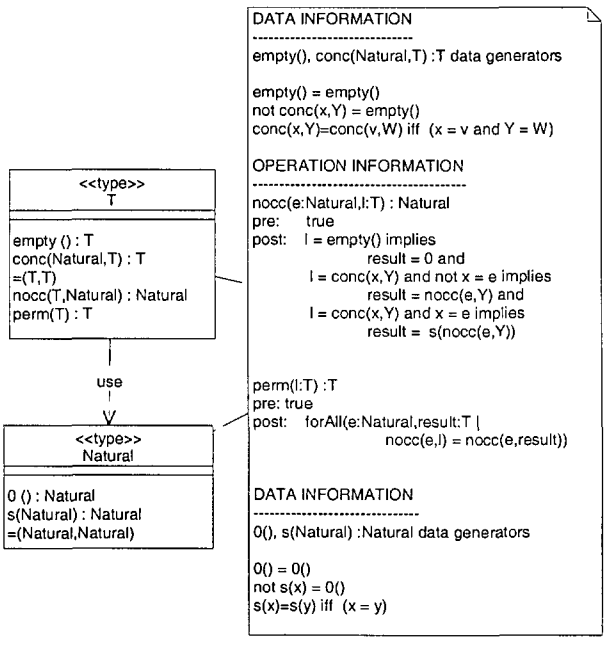


Figure 3: Formal ADT.

from a programmer point of view. On the other hand, the formal specification $Spec_{nocc}$, in figure 3, has the following intuitive meaning: *"The number of occurrences of any natural element e in a T-term empty() is equal to zero. If the number of occurrences of e in Y is equal to n then the number of occurrences in a T-term of the form conc(e, Y) is equal to s(n) (i.e., n + 1). If the number of occurrences of e in Y is equal to n then the number of occurrences in conc(x, Y) with x ≠ e is n"*. $Spec_{nocc}$ represents an abstract and expressive description of a *set of programs* which computes occurrences of natural numbers in T-terms but its form is closed to the structure of a program and then we tend to consider $Spec_{nocc}$ as a *constructive specification*. However, $Spec_{perm}$ represents also an abstract and expressive description of a *set of programs* which computes T-permutations but its form is not closed to the structure of a program and then we tend to consider $Spec_{perm}$ as a *non-constructive specification*.

A *pre/post specification* $Spec_{Op}$ for an operation Op is an expression of the form:

$$Op(x_1 : T_{x_1}, ..., x_n : T_{x_n}) : T_{result}$$
$$pre : Pre(x_1, ..., x_n)$$
$$post : Post(x_1, ..., x_n, y_1, ..., y_m, result)$$

where $Op(x_1 : T_{x_1}, ..., x_n : T_{x_n}) : T_{result}$ is the operation signature, $Pre(x_1, ..., x_n)$ is a first order formula called precondition and $Post(x_1, ..., x_n, y_1, ..., y_m, result)$ is a first order formula called postcondition. The symbol Op is called the *defined operation*.

A *pre/post specification* is called *total* iff $Pre(x_1, ..., x_n)$ is equivalent to *true*. A *pre/post specification* is called *partial* iff $Pre(x_1, ..., x_n)$ is not equivalent to *true*. A *recursive pre/post specification* is a pre/post specification where some elements in $Post(x_1, ..., x_n, y_1, ..., y_m, result)$ are defined on the Op symbol. (e.g., $Spec_{nocc}$ in figure 3). We consider only well founded recursive specifications.

An *explicit pre/post specification* is a non-recursive pre/post specification (e.g., $Spec_{perm}$ in figure 3).

A *non-constructive pre/post specification* is a pre/post specification where some quantified variables v, with $v \in \{x_1, ..., x_n, y_1, ..., y_m, result\}$ and $v \notin \{x_1, ..., x_n\}$, are unbounded (e.g., $Spec_{perm}$ in figure 3 presents the variables e and *result* as unbounded variables). A *constructive pre/post specification* is a specification where each quantified variable v, with $v \in \{x_1, ..., x_n, y_1, ..., y_m, result\}$ and $v \notin \{x_1, ..., x_n\}$, is bound by identity (e.g., $Spec_{nocc}$ in figure 3 binds the variables x and Y to the variable l and the variable *result* to the terms 0, $nocc(e, Y)$ and $s(nocc(e, Y))$).

Constructive pre/post specifications represent an effective setting to construct implementations and to test implementations against their formal specifications. For example, $Spec_{nocc}$ could be implemented as follows:

$nocc(e : Natural, l : T) : Natural$
$pre : true$
if $(l = empty())$ then return 0;

else if $(l = conc(x, Y)$ *and not* $x = e)$ then
return $nocc(e, Y)$;
else return $s(nocc(e, Y))$;
endif
endif

$post : l = empty()$ *implies* $result = 0$ *and*
$l = conc(x, Y)$ *and not* $x = e$ *implies*
$result = nocc(e, Y)$ *and*
$l = conc(x, Y)$ *and* $x = e$ *implies*
$result = s(nocc(e, Y))$

The realization of the functions *empty* and *conc* and the identity $=$ are needed to complete the implementation of $Spec_{nocc}$. However, this constructive step is unnecessary if the programmer only want to test algorithmic properties. For example, the programmer could select the set of "abstract" data $D = \{empty, conc(0, empty), conc(0, s(0), empty), ...\}$ to be tested. For each $t \in D$, a symbolic computation is made from $Spec_{nocc}$ and then a concrete computation is made from its implementation. From a programmer point of view, an implementation is not well constructed if tests, at implementation level, do not coincide with tests at specification level.

Non-constructive pre/post specifications do not represent an effective setting to construct and test implementations. For example $Spec_{perm}$ is difficult to program because its postcondition has some unbounded quantified variables (i.e., the variables e and *result*) and there is not a clear way (i.e., terminating way) to interpret these variables in implementation terms.

4 Writing Effective Contracts

We propose a form of specification based on stratification and redundancy. This combination balances between expression capabilities and proper levels of constructivism.

Let OP be the set of operation symbols declared in the ADT T. Let $Spec_{Op}$ be the pre/post specification of any $Op \in OP$. Let $Body_{Op}$ be the set of operation symbols in the postcondition of Op (e.g., in figure 3, $Body_{nocc} = \{nocc, =\}$ and $Body_{perm} = \{nocc, =\}$).

1. If T does not include use dependencies (e.g. see use relations in figures 2,3 and 4) from other types then T is *stratified* iff there exists a mapping *level* from OP to the set of natural numbers such that
   (a) For each $Op \in OP$, $level(Op) \geq level(s)$, for all $s \in Body_{Op}$
   (b) Each recursive $Op \in OP$ is constructive.
2. If T includes use dependencies from other types then, let TU be the set of used types in T and let IOP be the set of imported operations from types in TU. T is *stratified* iff
   (a) Each type in TU is stratified and
   (b) There exists a mapping *level* from $IOP \cup OP$ to the set of natural numbers such that for each $Op \in IOP$, $level(Op) = 0$ and for each $Op \in OP$, $level(Op) \geq level(s)$, for all $s \in Body_{Op}$.
   (c) Each recursive $Op \in OP$ is constructive.

   In figure 3 we show a stratified ADT.

The stratification of an ADT can be represented graphically by means of a dependency graph. Let Op_1 and Op_2 be two operations defined in an ADT T. We say Op_1 *depends upon* Op_2 in T iff $Op_2 \in Body_{Op_1}$. In the dependency graph, an operation symbol is represented by a node and a dependency by a directed arc. From a programmer point of view, a stratified ADT induces a method to construct implementations. The levels in the stratified ADT establish an implementation order between operation specifications. To follow this order is not mandatory but constructivism is reinforced if we construct implementations in a bottom-up way. For example, from the specification in figure 4, a programmer would start with the implementation of *nocc* and then would continue with the implementation of *perm*.

4.1 Incremental Writing of Contracts

We propose a method to guide the writing of stratified ADTs. The first step in the writing of contracts is to propose reachable types. This step is important in order to have a clear and common data domain between specifier and programmer. In our example, we would start the writing of T by establishing data information sections for *Natural* and T types respectively (see figure 2). The next steps will establish a kernel sufficiently expressive. Usually, at these steps, operations with recursive and constructive specifications are written. These operations are constructed from identity relations and operations previously written. In our example, we would continue the writing of T by including $Spec_{nocc}$ (see figure 3). When a sufficiently expressive kernel has been established, we propose the use of explicit specifications for the remaining specifications. In our example, we would continue the writing of T by including $Spec_{perm}$ (see figure 3). This method is not mandatory but we consider that it constitutes an assistance to the writing of stratified ADTs. In figure 4 we shows a formal ADT T with stratification information.

4.2 Eliminating Unbounded Variables

Subsection 3.2 established unbounded variables as the major reason of non-constructivism. In some situations, it is possible to rewrite specifications by reducing the number of unbounded variables and preserving semantics. This activity is not systematic but the specifier must consider it. For example $Spec_{perm}$ has two unbounded variables. The specifier would rewrite the specification in the following form:

$$
\begin{aligned}
&perm(l : T) : T \\
&pre : true \\
&post : forall(a : Natural\ in\ l, \\
&\qquad b : Natural\ in\ result, result : T \mid \\
&\qquad nocc(b, l) = nocc(b, result)\ and \\
&\qquad nocc(a, l) = nocc(a, result))
\end{aligned}
$$

This rewriting preserves semantics and, in addition, it assists programmers to implement (the functional part of) components because the rewritten specification has *result* as the only unbounded variable. The programmer must consider only one unbounded variable in order to construct *perm*-behavior examples. Hence, these "opportunistic" decisions enable programmers to understand specifications and to construct tests for implementations.

4.3 Augmenting Constructivism by means of Redundancies

Incremental writing of ADTs and reduction of unbounded variables are not sufficient. Regarding the bibliography about systematic program development, usually, constructivism is obtained by adding new formulas to the specification. For example, invariants and weakest preconditions constitutes a well known formal technique to design iteration-based implementations from pre/post specifications (Billington & Dromey 1996, Dromey 1988). Following these ideas, we propose a particular method based on symbolic evaluations. A symbolic evaluation of an operation represents an abstract execution (usually, developed by a constructive proof (Fribourg 1993, Wiggins et al. 1992)) with respect to a set of data values. Such evaluations are

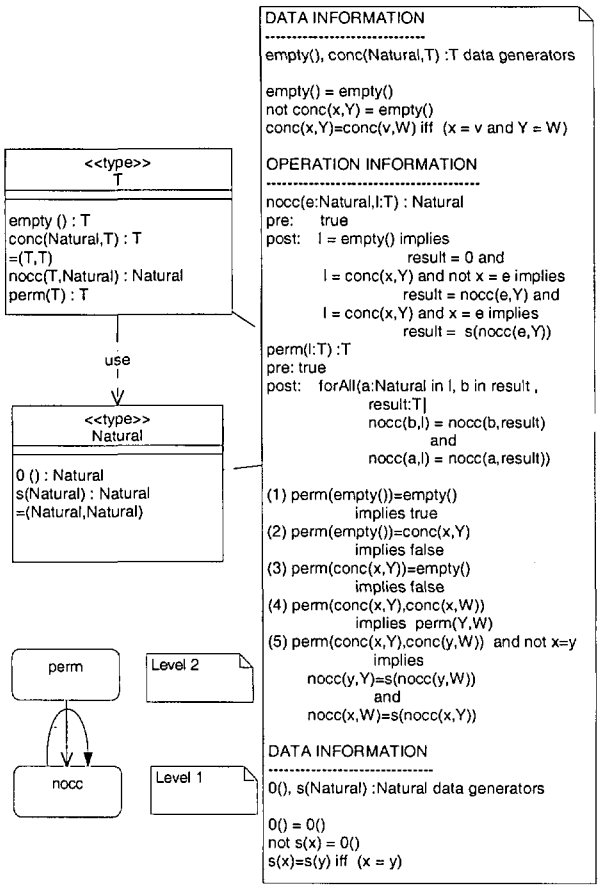


Figure 4: ADT specification with stratification and redundant information

added to the specification in order to establish constructive information. For example, a symbolic evaluation of the *perm* operation poses the following questions:

$$(1)\, perm(empty()) = empty()?$$
$$(2)\, perm(empty()) = conc(x, Y)?$$
$$(3)\, perm(conc(x, Y)) = empty()?$$

$$(4)\, perm(conc(x, Y)) = conc(x, W)?$$
$$(5)\, perm(conc(x, Y)) = conc(y, W)?$$

In order to answer these questions we execute S_{perm} in abstract terms. An important thing here is the form the specifier answers these questions. We consider that clausal form constitutes a kind of redundancy which assists programmers in operational terms. Figure 4 shows an example of ADT specification with redundancies in clausal form. This information is redundant and, probably the programmer could deduce it from the specification but we consider that the specifier is a qualified person to develop this task because he is able to bound interesting redundancies needed to induce recursions or iterations.

The following incomplete implementation could be induced from redundancies for the relation *perm*:

$perm(l : T) : T$
$pre : true$
if $(l = empty())$ then return $empty()$; from (1)
else if $(l = conc(x, Y))$ then
return $conc(x, perm(Y))$; from (4)
else ... from (5)
endif
endif
$post : forall(a : Natural\ in\ l,$
$b : Natural\ in\ result, result : T \mid$
$nocc(b, l) = nocc(b, result)\ and$
$nocc(a, l) = nocc(a, result))$

The programmer has an incomplete but useful implementation frame. The rest of the implementation can be established within this frame, reducing thus the search space. The programmer is able to continue the implementation in a top-down manner identifying different situations and programming each of these situations as a separate problem. For example, the programmer would only center its efforts on solving situations such as $perm(conc(x, Y)) = conc(y, W)$ considering $not\ x = y$.

Once the (functional part of the) component has been constructed, interfaces are used to characterize its services. Only functional properties are required by its users. Therefore, stratification and redundant informations are not needed for interface characterizations.

5 Conclusions

We propose a particular form of type specification based on constructive terms. This form is essential in order to consider ADTs as rigorous and effective contracts for (functional parts of) components between specifiers and programmers. ADTs can be seen as software contracts from two different points of view. From a specifier point of view, expressive languages for describing ADTs are needed in order to construct specifications in a effective way. However, from a programmer point of view, implementation models impose many restrictions and then programming from abstract and expressive specifications could represent an ineffective activity. To overcome this problem, we search for the right balance between these two opposite positions. For us, a proper balance requires some mutual concessions. For example, (a) specifiers and programmers are bound to unique and abstract data models. (b) We do not want considerable restrictions on the form of specification. Therefore, we cannot take advantage of the strengths of previous formal characterizations (e.g., equational characterizations). (c) An effective contract must not only describe, in abstract terms, what must be implemented but, in addition, it must advise in "abstract terms" how to do it (e.g., stratification information and redundancies). Our work represents an attempt to establish a specification method not only well founded but effective. At this moment, much work remains

to be done in order to consider these kinds of specifications as the basis for describing rigorous and effective contracts for (functional part of) components. Our work represents only a starting point to do it.

References

[1] Avellone A., Ferrari M. & Miglioli P. (1999) Synthesis of Programs in Abstract Data Types. *Proceedings of the LOPSTR'98 Workshop.* p. 81-100.

[2] Billington D. & Dromey, G. (1996) The Co-invariant Generator: An Aid in Deriving Loop Bodies. *Formal Aspects of Computing* 4, p. 108-126.

[3] Deville, Y. & Lau K-K. (1994) Logic Program Synthesis. *J. Logic Programming*, 19/20, p. 321-350.

[4] Dijsktra, E. W. (1976) A Discipline of Programming. *Prentice-Hall*, Englewood Cliffs, N.J.

[5] Dromey R. G. (1988) Systematic Program Development. *IEEE Transaction of Software Engineering.* 14(1) p. 12-29.

[6] Ehrig H. & Mahr B. (1985) Fundamentals of Algebraic Specification 1. Equations and Initial Semantics. *Springer-Verlag.*

[7] Ehrig H. & Mahr B. (1990) Fundamentals of Algebraic Specification 2. Module Specifications and Constraints. *Springer-Verlag.*

[8] Flener, P. (1995) Logic Program Synthesis from Incomplete Information. *Kluwer Academic Publishers.*

[9] Fribourg, L. (1993) Extracting Logic Programs that Use Extended Prolog Execution and Induction. *Constructing Logic Programs*, Wiley & Sons.

[10] Galán F. J., Cañete J. M. & Troyano J. A. (1999) On the Formal Translation of Object Oriented Software Specification: A Balanced Approach. *Proceedings of the V International Conference ISAS'99 IEEE Computer Society.*

[11] Wiggins G., Bundy A., Kraan I. & Hesketh J. (1992) Synthesis and Transformation of Logic Program from Constructive, Inductive Proof. *Proceedings of the LOPSTR'91.* p. 27-45.

[12] Wirsing M. (1990) Algebraic Specification. *Handbook of Theoretical Computer Science. Formal Models and Semantics* Elsevier, p. 677-780.

Approach to component based synthesis of fault tolerant software

Behrooz Parhami
University of California, Santa Barbara, CA 93106, USA
Phone: +1 805 893 3211, Fax: +1 805 893 3262
parhami@ece.ucsb.edu

Keywords: components, design diversity, fault tolerance, multichannel computation, multiversion software, software reliability, safety, software reuse, weighted voting.

Received: June 6, 2001

N-version programming (NVP) and acceptance testing (AT) are established methods for obtaining highly reliable results from imperfect software. In NVP, several program modules are executed independently and the final result is derived by voting on the module outputs. In AT (as embodied, for example, in the recovery-block construct), outputs of a program module are subjected to an acceptance test and in the event of failing the test, alternate modules are invoked, until a module produces results that pass the test. Various symmetric combinations of NVP and AT techniques have also been suggested. We have found that a more general view, allowing the insertion of ATs at arbitrary points within a suitably constructed multichannel computation graph can lead to higher reliability and/or greater cost-effectiveness compared to the previously envisaged hybrid schemes such as consensus recovery blocks, recoverable N-version blocks, and N-self-checking programs. Accordingly, we introduce MTV graphs, and their simplified data-driven version called DD-MTV graphs, as component-based frameworks for the creation, representation, and analysis of hybrid NVP-AT schemes. MTV graphs model variations in fault-tolerant software architectures built of computation module (M), acceptance test (T), and voter (V) components. Following the definition of (DD-)MTV graphs, we present several examples of hybrid NVP-AT schemes, as instances of fault-tolerant software based on our component-based approach, and quantify the resulting reliability improvements. We show, for example, that certain, somewhat asymmetric, combinations of M, T, and V components lead to higher reliabilities and/or lower cost than previously proposed symmetric arrangements. We conclude that our component-based approach facilitates design space exploration for fault-tolerant software and leads to reliability improvements due to the double effect of architectural optimization and component refinement afforded by reuse.

1 Introduction

Applications of highly dependable computer systems are no longer limited to exotic space exploration and defense systems. A multitude of information and control systems in avionics, transportation, transaction processing, and process monitoring also rely on existence of ultrareliable computational resources [32], [43]. With the continually increasing complexity of hardware and software modules, and the attendant impossibility of implementing perfect (defect- or fault-free) components, the use of multi-channel computations with *design diversity* [2], [16], [17], [23], [51] has emerged as a practical and cost-effective approach.

Diverse multichannel computations take advantage of the property that, with adequate testing, malfunctions caused by residual design defects are rare and occur only for highly unusual sets of input conditions. It is thus likely that malfunctions of independently constructed modules, based on the same initial specifications, occur for different input states. This design diversity approach has been found useful for hardware subsystems [45] and for data [1] as well, but its primary application area is in constructing highly reliable software systems based on one of two distinct paradigms: Voting on multiple versions and acceptance testing of results [13], [24].

Following the success of hardware and data replication methods in tolerating physical faults in computing systems, the use of N-Version Programming (NVP) was proposed to allow tolerance of software design flaws [3], [8]. In NVP, several program modules are executed independently and the final result is obtained by voting on the module results. Voting, as used here, covers a wide variety of techniques in terms of sophistication, flexibility, and computational complexity [15], [30], [33] and need not be implemented through simple matching and majority rule. Several other terms, such as "consensus" [3] and "adjudication" [9] have been used to describe the decision process that computes an output based on possibly inexact or incomplete results provided by multiple redundant modules.

An important objection to NVP is that independence of design flaws in multiple versions cannot be guaranteed and that commonly used specification and software design techniques may lead to related faults in independently designed versions [21]. Such related faults may cause identical or similar errors and thus lead to an

incorrect voter output. However, this is a criticism of dependability analyses for NVP and the attendant quantitative claims of reliability improvement. The usefulness of the NVP approach has never been in doubt. In some contexts, diversity is more readily ensured [35]. Also, attempts have been made to model the effects of correlated failures (e.g., [12], [25], [27]).

The technique of acceptance testing, e.g. as proposed in the recovery-block scheme [38], [39] is also based on design diversity. An acceptance test is an application-dependent routine that either accepts a result or declares it incorrect/suspect. Our confidence in an "accepted" result being actually "correct" depends on the thoroughness (coverage) of the acceptance test and its own reliability. ATs come in many different forms; from simple reasonableness checks to more complex, high-coverage validators. Assuming that the correctness of module result can be judged fairly accurately by applying an AT, alternate modules can be invoked sequentially in some specified order until one has produced a result passing the AT. This result is taken to be correct.

The main problem with redundancy techniques that rely solely on AT for validating a result is the difficulty of designing good acceptance tests that are both simple and thorough. Certain computations are easily checked through mathematical properties that relate the outputs to the inputs or by way of inverse computations where they exist [6]. In a great majority of cases, however, the only sure way of checking result validity with high confidence is to simply recompute using algorithms and/or hardware with diverse designs or operational characteristics.

Reliability evaluation for NVP and AT schemes has been presented in [5], [11], [23], [37], [41], [50], [52]. Linguistic constructs for describing adjudication and exception handling schemes for multi-version programs have been proposed in [26]. Kim et al [18], [19], [20] have studied architectures for, and design issues pertaining to, implementing a modified form of the recovery block scheme in a distributed environment. The resulting distributed recovery block (DRB) scheme uses concurrent execution of *try blocks* to allow fast forward recovery. Dugan and Lyu [10] discuss the relative merits of DRB, NVP, and NSCP (see the next paragraph).

Several groups have tried to combine NVP and AT. One such attempt is consensus recovery blocks (CRB) in which n versions are executed and their results are compared [40]. If there is agreement between two or more versions, then their common result is assumed correct and used. Otherwise, the n disagreeing results are subjected to an AT is some prespecified order and the first to pass the test is taken as the correct output. Another case is recoverable N-version blocks (RNVB) in which ATs are run on module outputs and only those that pass are provided to the voter [14]. This approach has also been suggested in [5], [16] and, under the name "N self-checking programming" (NSCP), in [22], [24]. Conceptually related to the efforts above, but different in terms of components and implementation, is *processor-data-check method*, and its graph-theoretic formulation, in algorithm-based fault tolerance [4], [44], [36].

Clearly, these are just examples of the ways in which NVP and AT approaches can be combined. CRB essentially applies NVP (with relaxed 2-out-of-n voting) and AT schemes sequentially and one at a time. Because no AT is applied in the case of, say, two agreeing results, there is some chance of an erroneous output being propagated. Furthermore, a common AT is assumed and design diversity is not applied to the AT. Both RNVB and NSCP approaches envisage applying ATs uniformly to all versions. This leads to an increase in complexity since ATs may essentially be duplicates of computational modules. We have found that a more general view, allowing the insertion of ATs at arbitrary points within a suitably constructed multichannel computation graph can lead to higher reliability and/or greater cost-effectiveness compared to the previously envisaged hybrid schemes such as consensus recovery blocks, recoverable N-version blocks, and N-self-checking programs.

Accordingly, we introduce MTV graphs, and their simplified data-driven version called DD-MTV graphs, as component-based frameworks for the creation, representation, and analysis of hybrid NVP-AT schemes. MTV graphs model variations in fault-tolerant software architectures built of computation module (M), acceptance test (T), and voter (V) components [34]. Following the definition of (DD-)MTV graphs, we present several examples of hybrid NVP-AT schemes, as instances of fault-tolerant software architectures developed based on our component-based approach, and quantify the reliability improvements they achieve. We show, for example, that certain, somewhat asymmetric, combinations of M, T, and V components can lead to higher reliabilities than previously proposed symmetric arrangements having comparable or higher complexities. We conclude that our component-based approach facilitates the exploration of the design space for fault-tolerant software and leads to reliability improvements due to the double effect of architectural optimization and component refinement afforded by reuse.

The rest of this paper is organized as follows. Section 2 contains basic definitions and assumptions as well as examples of a more general hybrid NVP-AT schemes in order to motivate the subsequent discussion. Sections 3 and 4 analyze NVP schemes in which 1 or k of the n versions, respectively, have been replaced by ATs. Section 5 deals with an example of more general combining schemes. Section 6 examines the effect of correlated failures. Conclusions and directions for further research appear in Section 7.

2 Terminology and assumptions

The question that we have set out to answer is how to combine the techniques of NVP and AT in an optimal way in order to achieve the best possible results. More specifically, our ultimate goal is to be able to combine diverse components (software modules, acceptance tests, voting algorithms) in a systematic way in order to maximize the correctness probability of the output with a given overall complexity or to achieve a desired correctness probability with minimal cost.

Unfortunately, in view of difficulties in estimating reliability and cost parameters, except in very limited cases [42], these problems are currently intractable when posed in their full generality. So, in this initial study, we endeavor to obtain results for a rather limited set of more specific questions with several simplifying assumptions.

Our hope is that with further research, the domain can be broadened and the assumptions gradually relaxed.

Here are our main objectives for this paper:

1. Demonstrate that certain novel, somewhat asymmetric, arrangements of modules, acceptance tests, and voters could be more reliable than earlier proposals.
2. Find optimality results in certain special cases; e.g., when the arrangement contains a single acceptance test or only one level of voting.
3. Provide examples of how the models and techniques used to analyze the special cases can be extended to deal with more complicated arrangements.
4. Generate interest in further systematic studies of the methodology, and tradeoff issues, for component-based synthesis of fault-tolerant software.

The following definitions and assumptions are needed in our discussions and analyses.

2.1. Definition – *MTV graph*: An MTV (Module-Test-Voter) graph is a directed acyclic graph with one "In", one "Out", and possibly one "Error" node, plus any number of nodes of three other types: Modules (M), acceptance tests (T), and voters (V).

M: Computes a result based on its inputs and sends it to a T or V node or to Out.
T: Accepts its input and forwards it or rejects it and activates some M or T nodes.
V: Forwards the result of weighted plurality voting or activates some M or T nodes.

The inner workings of M and T components are application-dependent. We make no assumption about these parts except that they have known, statistically independent reliability parameters (see 2.2-2.4). Voter components are more formally described in Def. 2.5. The nodes are connected by directed edges representing data transfers and controls. In diagrams, forwarding of data (the "P" output of an AT or the voting result from a V node) is represented by solid edges while activation controls (the "F" output of an AT or an indecisive voting outcome) are represented by dotted edges. ♦

Figures 1 and 2 contain examples of MTV graphs whose full meanings will become clear following the introduction of several more assumptions and definitions.

2.2. Assumption – *Reliability parameters of computation modules*: Each computation module M_i produces a result which is correct with fixed probability q_i and incorrect with fixed probability p_i, uniformly over its input space. In case $q_i + p_i < 1$, the module may be viewed as (partially) self-checking or fail-safe, abstaining from producing any result with probability $1 - q_i - p_i$. In the rest of this paper, we assume $q_i + p_i = 1$. ♦

Assumption 2.2 is controversial in that it is very difficult, if not impossible, to accurately estimate the reliability q_i of a software module [7]. We justify this assumption by noting that any system-level reliability analysis must be based on the reliability parameters of the components used. Accurate reliability estimates will be available with greater ease as we gain experience with the design and use of multiversion software. A Component-based strategy is helpful in this regard because component reuse fosters the gathering of more accurate reliability and performance data. Additionally, when comparing various arrangements of modules and tests, occasionally we obtain results indicating that one scheme is better than another for a wide range of parameter values or even for all values of a certain p_i. Hence, such comparative evaluations are less sensitive to, or totally independent of, the availability of accurate estimates for the p_is. The comments above apply to Assumption 2.3 as well.

2.3. Assumption – *Reliability parameters of acceptance tests*: A correct result passes an acceptance test T_i (outgoing edge labeled "P" is taken) with fixed probability q'_i and fails it (outgoing edge "F" is taken) with fixed probability p'_i, uniformly over the space of correct inputs. Similarly, T_i rejects an incorrect result with fixed probability q''_i and accepts it with fixed probability p''_i, uniformly over the space of incorrect inputs (note that in all definitions, q_is are close to 1 whereas p_is are near 0). When $q'_i + p'_i < 1$ or $q''_i + p''_i < 1$, the AT is viewed as (partially) self-checking or fail-safe, abstaining from judging an input with probability $1 - q'_i - p'_i$ for correct inputs and $1 - q''_i - p''_i$ for incorrect ones. Henceforth, we assume $q'_i + p'_i = q''_i + p''_i = 1$. ♦

The reason we do not start by assuming $q'_i = q''_i$ is that ATs behave asymmetrically with respect to correct and incorrect inputs. An AT that is itself defect-free, always accepts a correct input. Thus, p'_i is typically small and is related to the probability of a defect in the AT design. On the other hand, even a defect-free AT may accept an incorrect input due to imperfections in the testing algorithm (imperfect coverage). Hence, p''_i lumps together two sources of errors: imperfect coverage and defective design. We typically have $p''_i > p'_i$ (perhaps even, $p''_i >> p'_i$) for simple, low-complexity ATs. For a more comprehensive AT (e.g., one that duplicates the computation and decides by comparing the two values), coverage can be very high or even perfect. In such cases, q'_i and q''_i are comparable, though not necessarily equal.

2.4. Assumption – *s-independence of module and AT failures*: Each M and each AT fails s-independently of other Ms and ATs, unless otherwise noted. Hence, the probability of k modules M_i ($1 \le i \le k$) coincidentally producing erroneous results is $\Pi_{i \in [1, k]} p_i$. ♦

Assumption 2.4 is perhaps our most important assumption and the one most likely to be criticized. So let us try to justify it briefly. As noted in the introduction, the assumption of failure independence for multiversion software has been scrutinized and questioned from early on [21]. We think that these criticisms are valid and must be considered very seriously when trying to compute absolute reliability values for multichannel computations. However, the problem is much less serious for the types of analyses presented in this paper. Here, we try to determine if one scheme offers reliability improvement over another. Intuitively, since dependent failures are likely to affect the reliabilities of both schemes being compared, we can have a higher confidence in such relative figures of merit than in absolute reliabilities. We relax this independence assumption in Section 6 in order to validate, in part, this intuition. More work is clearly needed in this direction.

2.5. Definition – *Weighted plurality voter*: Given n input data objects $x_1, x_2, \ldots, x_n$, with associated nonnegative real votes (weights) $v_1, v_2, \ldots, v_n$, a V node computes the output object y and its vote w such that y is "supported by" a number of input data objects with votes

totalling w and no other y' is supported by inputs having more votes. If $w \leq (\sum_{i\in[1,n]} v_i)/2$, then the outcome y may be nonunique. In such cases, an erroneous voter output will be pessimistically assumed. The output weight w selects one of the outgoing voter edges along which y or an activation signal must be sent (see, e.g., Fig. 1). When input votes are not explicitly specified, it is assumed that $v_1 = v_2 = \ldots = v_n = 1$. Various definitions of the term "supported by" lead to different voting schemes such as exact, inexact, and approval voting (e.g., with exact voting, an input object x_i supports y iff $x_i = y$). These variations, although important, are beyond the scope of this paper [30], [33]. ♦

2.6. Assumption – *Perfect voters*: Voters are perfect and act instantaneously. This assumption is reasonable because voters are simpler than modules or ATs and are designed just once for use with many different modules and test types. They can be made highly reliable through careful design and extensive testing (much more so than what is reasonable for any single application or its associated ATs). ♦

Consider Fig. 1 in which 5VP and an alternative scheme with the same number of M/T modules, and thus with lower or equal complexity, are shown. T is an acceptance test with pass/fail (P/F) outcome. A result that passes T is simply forwarded to the output. When T rejects its input, it activates M_4. The voter in Fig. 1c can produce one of three mutually exclusive outputs: (1) No agreement, leading to activation of M_4, (2) agreement between two inputs, leading to the application of T on the agreed-upon result, and (3) agreement among all three inputs, yielding an acceptable output.

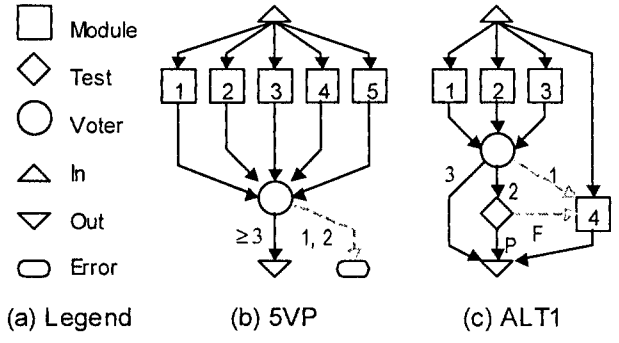


Fig. 1. Representations of 5VP and ALT1, an alternative scheme, as MTV graphs.

Figure 2 shows MTV graphs corresponding to two other hybrid NVP-AT schemes. These schemes have been proposed in the literature as alternatives to 3VP, although both imply greater cost than 3VP. Figure 2a represents a 3-channel RNVB or NSCP. Each of the three computation channels is made self-checking through the insertion of an AT after the computation module. The V node in Fig. 2a is a weighted plurality voter for which each input weight is set to 0 or 1, based on the outcome of the associated AT.

Figure 2b represents a 3-channel CRB. If the voter observes agreement between two or all three of its inputs, the agreed-upon result is forwarded to the output. On the other hand, when there is no agreement, results from the computation channels are successively subjected to an AT and the first to pass the test is forwarded to output. The ATs in Fig. 2b are labeled T_1, T_2, and T_3, but they may all represent the same test. The advantage of this scheme over 3VP is that it is guaranteed to produce the correct result whenever 3VP would produce the correct result. Additionally, the added AT mechanism may salvage a correct result from disagreeing modules.

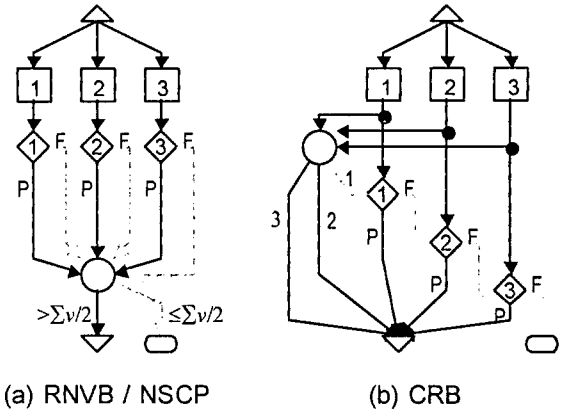


Fig. 2. MTV graphs representing two different 3-channel hybrid NVP-AT schemes.

Returning now to the initial example, the 5VP scheme of Fig. 1b can tolerate up to two module failures, but can produce an incorrect result with some triple failures. In fact, any worst-case analysis must assume that 5VP fails when there are three module failures. The alternative scheme shown in Fig. 1c also tolerates any two failures in the M/T nodes. This claim is proven by considering the following four cases which exhaust all possible double failures.

Case 1 – Two failures in $\{M_1, M_2, M_3\}$: T and M_4 are fault-free. If the two faulty modules produce mutually supportive results, then the error is caught by T and M_4 is executed. Otherwise, M_4 is executed directly. Either way, the correct result is produced.

Case 2 – Two failures in $\{T, M_4\}$: M_1, M_2, and M_3 are fault-free and produce pairwise mutually supportive results. The voter outputs is the correct result.

Case 3 – One failure in $\{M_1, M_2, M_3\}$ and one in T: Because M_4 is fault-free, the output is correct whether or not T accepts the correct result received from the voter.

Case 4 – One failure in $\{M_1, M_2, M_3\}$ and one in M_4: T is fault-free and accepts the voter's majority result.

Triple failures can lead to an incorrect output for the alternative scheme in a manner similar to 5VP. For example, if M_1, M_2, and M_3 are faulty but produce incorrect pairwise mutually supportive results, an incorrect value will be output. Failure of M_1, T, and M_4 also can lead to incorrect output. A side benefit of the alternative scheme of Fig. 1c is that whereas all five modules must run to completion in 5VP, the alternative scheme rarely needs to execute M_4.

Let us now analyze the two schemes of Fig. 1 with respect to reliability (probability of producing a correct result). Assuming $p_1 = p_2 = p_3 = p_4 = p = 1 - q$:

$$Q_{5VP} = q^5 + 5q^4p + 10q^3p^2 = q^2(1 + 2p + 3p^2 - 6p^3)$$

$$Q_{ALT1} = q^3 + 3q^2p(1 - p'p) + 3qp^2q''q$$
$$= q^2[1 + 2p + 3p^2 - 3p^2(p' + p'')]$$

The equation for the reliability of the alternate configuration in Fig. 1c, Q_{ALT1}, is derived as follows. The first term, q^3, is the probability of having no fault in $\{M_1, M_2, M_3\}$. The second term, $3q^2p(1 - p'p)$, covers the event of having a single fault in $\{M_1, M_2, M_3\}$, in which case correctness of the result is guaranteed unless T rejects the correct majority result *and* M_4 is faulty. The third term, $3qp^2q''q$, corresponds to having two faults in $\{M_1, M_2, M_3\}$. In this event, the worst case is the agreement of the two faulty modules because it leads to the requirement for T to reject the incorrect majority result (probability q'') *and* for M_4 to be fault-free (probability q) to obtain the correct result. If the two faulty modules in $\{M_1, M_2, M_3\}$ disagree with each other and with the correct result, then the only requirement for producing the correct output is for M_4 to be fault-free.

Comparing the two expressions above, we find that $Q_{ALT1} > Q_{5VP}$ iff $p' + p'' < 2p$. In the special case of $p' = p'' = s$, the alternate scheme ALT1 offers reliability improvement over 5VP iff $s < p$; i.e., the alternate scheme is better if the acceptance test component T is more reliable than each module M_i. In practice, T can often be made simpler, and thus more reliable, than M_i. This may be due to the inherent simplicity of verifying the result's correctness (e.g., by means of a mathematical relationship or an inverse computation) or by virtue of T using extra information ("certification trail") provided by the computation modules [46], [47], [48].

The examples above were intended to demonstrate the power of MTV graphs as representation and analysis tools for multichannel computations. In particular, it was shown that previously proposed hybrid NVP-AT approaches can be represented as simple MTV graphs and that these graphs can also represent more general hybrid schemes that have not been dealt with in the past and that can potentially offer higher reliability and/or lower overall complexity. Next, we define a modified form of MTV graphs in order to simplify the discussion in the remainder of this paper.

2.7. Definition – *Data-driven MTV (DD-MTV) graph*: A DD-MTV graph is a modified MTV graph with no Error node and only single-output M, T, and V nodes.

M_i: Attaches the weight w_i to its output y_i.

T_i: Modifies the weight w of y to $w + a_i(w)$ or $w - r_i(w)$ upon acceptance/rejection.

V_i: Produces data object y of weight w from its inputs, as detailed in Def. 2.5.

The Error node is not needed because error can be indicated by a subset of possible weights (low values) for the final result. The elimination of control edges and the resultant graph's uniformity simplifies the enumeration and analysis of various alternatives, while still retaining the power to accurately model most hybrid schemes. The weight augmentation and reduction functions, $a_i(w)$ and $r_i(w)$, used to adjust the weight of an accepted and rejected input, respectively, are nonnegative functions to be determined (see Assumption 2.9). ♦

Figure 3 depicts several examples of DD-MTV graphs, each having six M/T nodes. These can be viewed as lower-complexity alternatives to 6VP. The examples in Fig. 3 clearly show the wide variety of multichannel computational arrangements that can be modeled easily by DD-MTV graphs [29].

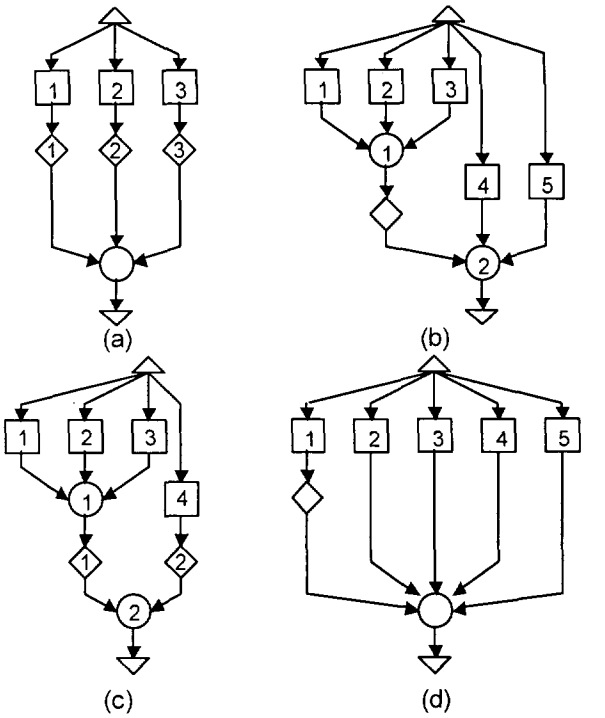


Fig. 3. DD-MTV graphs with six M/T nodes.

2.8. Assumption – *Uniformity of modules and ATs*: In the rest of this paper, modules will be assumed to have identical reliability, complexity, and execution-time parameters. Thus, the subscript i will be omitted from parameters such as p_i, q_i and the weight $w = 1$ is attached to all module outputs. Similarly, ATs will be uniformly treated by eliminating the subscript i from their respective parameters such as p'_i and p''_i. ATs will be taken to have perfect coverage and lower or equal complexity compared to modules, thus leading to the assumptions $p' \leq p$ and $p'' \leq p$. ♦

2.9. Assumption – *Weight augmentation/reduction functions for ATs*: Selection of appropriate weight augmentation and reduction functions, $a(w)$ and $r(w)$, can have important effects on the overall reliability of the system modeled by a particular DD-MTV graph. In this paper, we assume $a(w) = r(w) = 1$. These simple constant functions can be intuitively justified when ATs have near-perfect coverage and are of comparable complexity to modules, and they have worked well in practice. However, an extensive study of techniques for optimally choosing these functions is required. ♦

The stage is now set for a more detailed examination of certain classes of DD-MTV graphs and the systems they model. Before that, we recap the abbreviations used and introduce some needed notation.

2.10. Notation and nomenclature – The following is a list of symbols and abbreviations used in the paper:

AT	Acceptance Test(ing)
CRB	Consensus Recovery Block (Fig. 2b)
$C_{x,y}$	Binomial coefficient = $x!/[y!(x-y)!]$
DD-MTV	Data-Driven MTV graph (Def. 2.7)
M	Module; node in (DD-)MTV graph
MTV	Module-Test-Voter graph (Def. 2.1)
NSCP	N-Self-Checking Program (Fig. 2a)
nVP	n-Version Program(ming); e.g., 3VP
P	Failure probability = unreliability = $1 - Q$

Q (Q_X)	Reliability (of system or configuration X)
$R_{k,m}$	Reliability of k-out-of-m system
RNVB	Recoverable N-Version Block (Fig. 2a)
T	Test (AT) node in (DD-)MTV graph
V	Voter node in (DD-)MTV graph ♦

3 Replacing one version with an AT

When one module in 3VP is replaced by an acceptance test, a recovery block scheme with one alternate is obtained (Fig. 4a). Note that Fig. 4a is a correct model of recovery block scheme as far as reliability estimation is concerned. The fact that M_1 and M_2 appear to be running in parallel rather than M_2 following M_1, and then only in case T rejects M_1's result, is irrelevant to the reliability calculation. In this section, we generalize this notion by analyzing the effect of replacing one module in nVP with an AT (Fig. 4b). Reliability expressions for 3VP and the alternate scheme ALT2 of Fig. 4a are as follows:

$$Q_{3VP} = q^3 + 3q^2p = q(1 + p - 2p^2)$$

$$Q_{ALT2} = qq' + qp'q + pq''q = q[1 + p - p(p' + p'')]$$

The equation for the reliability of the alternate configuration in Fig. 4a, Q_{ALT2}, is derived as follows. The first term, qq', is the probability of M_1 producing the correct result *and* T accepting it. The second term, $qp'q$, covers the event of M_1 producing the correct result, T rejecting it, *and* M_2 getting the correct result. The third term, $pq''q$, corresponds to M_1 producing an incorrect result, T catching the error, *and* M_2 being fault-free. Comparing the expressions above, we find that $Q_{ALT2} > Q_{3VP}$ iff $p' + p'' < 2p$. Hence the discussion preceding Def. 2.7 applies here as well. Figure 5 shows the unreliability $P = 1 - Q$ of 3VP and ALT2 when $p' = p''$.

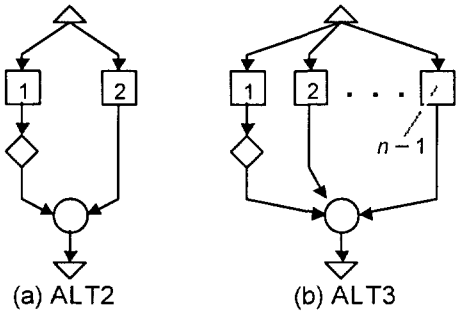


Fig. 4. Replacing one module with an acceptance test in 3VP and nVP.

It is relatively straightforward to generalize the analysis above to the comparison of nVP and the alternative scheme ALT3 depicted in Fig. 4b. However, we first need some notation. Let $R_{k,m}$ be the reliability of a homogeneous k-out-of-m system in which each module fails with probability p (for brevity, the parameter p is not explicitly shown).

$$R_{k,m} = \sum_{j \in [k, m]} C_{m,j}\, q^j p^{m-j}$$

where $C_{m,j}$ is the binomial coefficient. $R_{k,m}$ is defined to be 0 for $k > m$ and 1 for $k \leq 0$. We now write reliability equations for nVP and the ALT3 scheme of Fig. 4b as follows. To simplify the formulas, let $h = \lfloor n/2 \rfloor$. Each of the following expressions is written by considering the four possible cases with respect to the presence of faults in two modules or in one module and its associated AT and for each case figuring out how many of the remaining $n - 2$ modules must be fault-free in order to guarantee a correct result.

$$Q_{nVP} = R_{h+1,n} = q^2R_{h-1,n-2} + 2pqR_{h,n-2} + p^2R_{h+1,n-2}$$

$$Q_{ALT3} = qq'R_{h-1,n-2} + qp'R_{h,n-2} + pq''R_{h,n-2} + pp''R_{h+1,n-2}$$

To compare these reliabilities, let us compute their difference $\Delta Q = Q_{ALT3} - Q_{nVP}$:

$$\begin{aligned}\Delta Q &= q(p - p')R_{h-1,n-2} + [p(p - p'') - q(p - p')]R_{h,n-2} \\ &\quad - p(p - p'')R_{h+1,n-2} \\ &= q(p - p')[R_{h-1,n-2} - R_{h,n-2}] \\ &\quad + p(p - p'')[R_{h,n-2} - R_{h+1,n-2}]\end{aligned}$$

Because each of the two terms within the square brackets is positive, a sufficient condition for reliability improvement over nVP is immediately obtained as $max(p', p'') < p$, which always holds by Assumption 2.8.

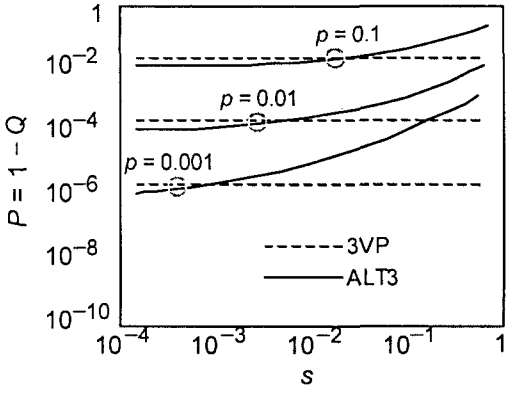


Fig. 5. Unreliability $P = 1 - Q$ of 3VP and ALT2, assuming $p' = p'' = s$.

To continue the analysis, we note that:

$$\begin{aligned}R_{k-1,m} - R_{k,m} &= C_{m,k-1}q^{k-1}p^{m-k+1} \\ &= m!q^{k-1}p^{m-k+1}/[(k-1)!(m-k+1)!]\end{aligned}$$

Hence, we can rewrite $\Delta Q = Q_{ALT3} - Q_{nVP}$ as:

$$\begin{aligned}\Delta Q &= q(p - p')(n-2)!q^{h-1}p^{n-h-1}/[(h-1)!(n-h-1)!] \\ &\quad + p(p - p'')(n-2)!q^hp^{n-h-2}/[h!(n-h-2)!] \\ &= \{(n-2)!/[h!(n-h-1)!]\}\, q^hp^{n-h-1} \\ &\quad \times [(n-1)p - hp' - (n-h-1)p'']\end{aligned}$$

Therefore, the sign of $\Delta Q = Q_{ALT3} - Q_{nVP}$ depends on the sign of the last expression within square brackets. For n odd, we have $h = (n-1)/2$ and $\Delta Q > 0$ iff $p' + p'' < 2p$. For n even, we have $h = n/2$ and $\Delta Q > 0$ iff $(n/2 - 1)(p' + p'') + p' < (n-1)p$. In this latter case, p' is somewhat more important than p''. As an example, for $n = 6$, we must have $3p' + 2p'' < 5p$ if the alternate with one AT (Fig. 3d) is to be more reliable than 6VP.

4 Replacing k versions with ATs

We now consider the case where k of the n modules are removed ($k \leq n/2$) and replaced by ATs following k of the remaining $n - k$ modules. As shown in the MTV graph of Fig. 6a, k branches with modules M_1, M_2, . . . , M_k include acceptance tests T_1, T_2, . . . , T_k and $n - 2k$ branches have just modules (indexed from k+1 to n–k).

As in Section 3, let $R_{k,m}$ be the reliability of a homogeneous k-out-of-m system in which each module fails with probability p and let $h = \lfloor n/2 \rfloor$ for notational convenience. We can then write:

$$Q_{n\text{VP}} = R_{h+1,n} = \sum_{i\in[h+1,\,n]} C_{n,i}\, q^i p^{n-i}$$

$$Q_{\text{ALT4}} = \sum_{i\in[0,\,k]}\sum_{j\in[0,\,k-i]}\{C_{k,i}C_{k-i,j}(pp'')^i(pq''+qp')^j \times (qq')^{k-i-j}R_{h+2i+j-2k+1,n-2k}\}$$

The reliability expression Q_{ALT4} for the alternate configuration is derived as follows. Let of the k branches containing ATs, i have faults in both the module and in the AT, j have a fault in either the module or the AT but not both, and $k-i-j$ be fault-free. The i branches with double faults all potentially produce incorrect results with weight 2. The j branches containing single faults produce results with weight 0, whether the fault is in M or in T. Finally, the $k-i-j$ fault-free branches produce correct results with weight 2. If c of the remaining $n-2k$ modules produce correct results, the following condition must be met for the output to be guaranteed correct:

$$c + 2(k-i-j) > (n-2k-c) + 2i \quad \Rightarrow \quad c \geq h + 2i + j - 2k + 1$$

This justifies the term $R_{h+2i+j-2k+1,n-2k}$ in the expression for Q_{ALT4}. The remaining terms are the probabilities of the indicated number of faults raised to appropriate powers. For example, the probability that M is faulty but T catches the error or M is fault-free but T rejects its output is $pq'' + qp' = p + p' - pp' - pp''$.

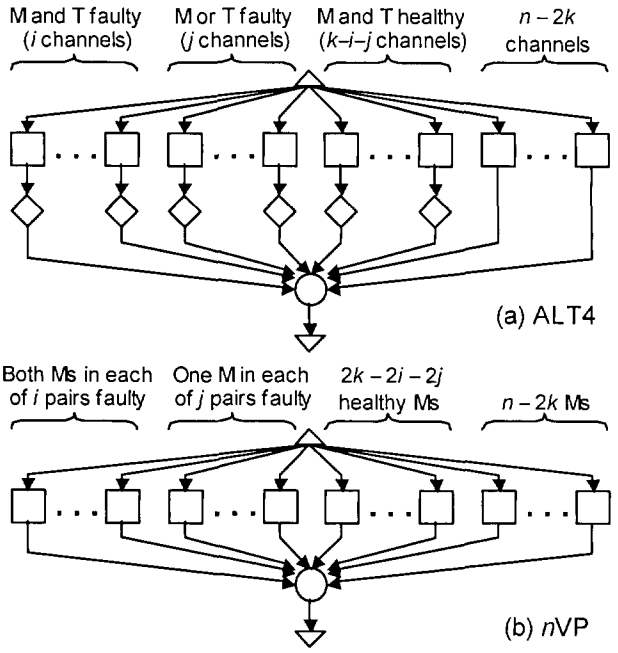


Fig. 6. Replacing k modules with ATs in nVP and the notation for reliability analysis.

As written, the expressions for $Q_{n\text{VP}}$ and Q_{ALT4} are difficult to compare without resorting to numerical calculation. To facilitate comparison, we rewrite the expression for $Q_{n\text{VP}}$ in the following way. We divide the set of n modules into k module pairs plus $n-2k$ individual modules, as shown in Fig. 6b. Each of the k pairs can have 2, 1, or 0 faulty modules. Let i, j, and $k-i-j$ be the number of such pairs, respectively. The k module pairs contribute incorrect results with total weights of up to $2i+j$ and correct results with total weights of at least $2k-2i-j$. Again, if the remaining $n-2k$ modules produce c correct results and $n-2k-c$ incorrect ones, we must have $c + 2k - 2i - j > (n-2k-c) + 2i + j$, or $c \geq h + 2i + j - 2k + 1$, to guarantee a correct output. The probabilities of having 2, 1, ore 0 faulty modules in a pair of modules are p^2, $2pq$, and q^2, respectively. Thus:

$$Q_{n\text{VP}} = \sum_{i\in[0,\,k]}\sum_{j\in[0,\,k-i]}\{C_{k,i}C_{k-i,j}(p^2)^i(2pq)^j \times (q^2)^{k-i-j}R_{h+2i+j-2k+1,n-2k}$$

Comparing the corresponding ij terms in the expression for Q_{ALT4} to the above expression for $Q_{n\text{VP}}$ provides some insight but no general conclusion. For example, for $p' = p'' = s$, corresponding terms become identical and the two schemes are equivalent with respect to reliability. For $p' = p'' = s$, the ij term in Q_{ALT4} divided by the ij term in $Q_{n\text{VP}}$ yields the ratio:

$$(s/p)^i[(p+s-2ps)/(2pq)]^j[(1-s)/q]^{k-i-j}$$

For particular values of s and p satisfying $s < p$, the first and the second term above are always less than 1 while the third term is always greater than 1. Hence, the ratio can be less than or greater than 1 depending on the values of i and j and no conclusion can be drawn based on this term-by-term comparison.

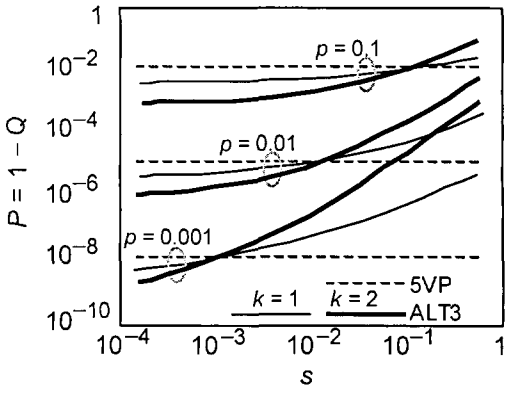


Fig. 7. Unreliability P=1–Q of 5VP and ALT4 with k = 1 or 2, assuming $p' = p'' = s$.

For n = 3, the only acceptable value for k is 1 and Fig. 5 depicts the corresponding changes in the unreliability $P = 1 - Q$. To note the effect of changing k, the expressions for $Q_{n\text{VP}}$ and Q_{ALT4} have been evaluated for n = 5, with k = 1 or 2, and p = 0.1, 0.01, or 0.001, assuming $p' = p'' = s$. Figure 7 depicts the resulting unreliabilities as functions of s.

As expected, the unreliabilities $P_{n\text{VP}}$ and P_{ALT4} are identical for $s = p$ (see the crossover points in Fig. 7). The 5VP scheme is uniformly better for $s > p$. In the case of $s < p$, both alternates are uniformly better than 5VP and the alternate with k = 2 is better than that with k = 1. It is worth noting that the improvement in reliability achieved for $s < p$ is smaller than the degradation suffered for $s > p$, particularly for larger k. Therefore, modules must be replaced with ATs only if the condition $s < p$ is reasonably certain.

5 More general schemes

As seen from the examples given in Fig. 3, DD-MTV graphs and the systems they model can be composed in many different ways. The systems discussed and analyzed in Sections 3 and 4 all involve a single level of voting. In this section, we discuss a system involving two levels of voting as an example of more general schemes. It is hoped that several other arrangements will be covered in the continuation of this research.

Consider the MTV graph ALT5 depicted in Fig. 8 as an alternative to nVP. The result of k-way voting on k of the modules is given to T and is then combined with the results of $n - k - 1$ modules through a second-level weighted voter. This may be viewed as a generalized recovery block scheme in which the primary computation consists of a k-way voted block and the alternate consists of an $(n - k - 1)$-way parallel block. In an actual implementation, modules in the alternate block may be executed sequentially until sufficient votes are collected, given the outcome produced by the primary voting block. The reliability of ALT5 is:

$$\begin{aligned} Q_{\text{ALT5}} = {} & \sum_{i\in[0,\lfloor k/2\rfloor]}\{C_{k,i}q^i p^{k-i} \\ & \times [q''R_{\lfloor (n-i)2\rfloor,n-k-1} + p''R_{\lfloor (n-i)2\rfloor+1,n-k-1}]\} \\ & + \sum_{i\in[\lfloor k/2\rfloor+1,\,k]}\{C_{k,i}q^i p^{k-i} \\ & \times [q'R_{\lfloor (n-k-i)2\rfloor,n-k-1} + p'R_{\lfloor (n-k-i)2\rfloor+1,n-k-1}]\} \end{aligned}$$

The reliability expression Q_{ALT5} is derived as follows. Let there be i correct results among the k channels of the primary voting block. This event has the probability $C_{k,i}q^i p^{k-i}$. Now if $i \le \lfloor k/2 \rfloor$, the plurality voting result must be assumed incorrect in the worst case. If T rejects this incorrect result (probability q''), its weight is decreased to $i - 1$ and a correct final output will be produced as long as at least $\lfloor (n - i)/2 \rfloor$ of the remaining $n - k - 1$ modules are fault-free. On the other hand, if T erroneously accepts the incorrect result (probability p''), thus increasing its weight to $i + 1$, then at least $\lfloor (n - i)/2 \rfloor + 1$ of the remaining $n - k - 1$ modules must be fault-free for the final result to be guaranteed correct. Recall that $R_{j,m} = 0$ for $j > m$. Similarly, if $i \ge \lfloor k/2 \rfloor + 1$, then the voter output is correct and has a weight of i. A similar argument justifies the second half of the expression for Q_{ALT5}.

To compare Q_{ALT5} to $Q_{n\text{VP}}$, we divide the n modules into three groups of k, 1, and $n - k - 1$ modules. If i is the number of fault-free modules in the first group, then:

$$\begin{aligned} Q_{n\text{VP}} = {} & \sum_{i\in[0,\,k]}\{C_{k,i}q^i p^{k-i} \\ & \times [qR_{\lfloor (n-k-i)/2\rfloor,n-k-1} + pR_{\lceil (n-k-i)2\rceil+1,n-k-1}]\} \end{aligned}$$

The two terms within square brackets correspond to the case of the single module in the second group being fault-free (probability q) or faulty (probability p), respectively, leading to different requirements for the number of fault-free modules in the third group ($i + 1 + c > n - k - 1 - c$ in the first case and $i + c > n - k - 1 - c$ in the second, where c is the required number of fault-free modules in the third group). Again comparison of the expressions for Q_{ALT5} and $Q_{n\text{VP}}$ leads to no general conclusion. For example, in the case of $p' = p'' = p$, we note that of the corresponding pairs of terms in the expressions for Q_{ALT5} and $Q_{n\text{VP}}$, some are larger in Q_{ALT5} and others are larger in $Q_{n\text{VP}}$.

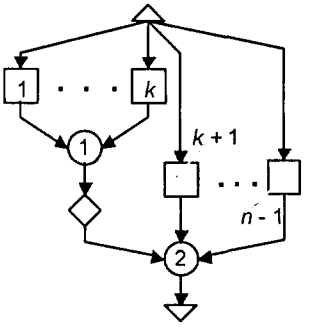


Fig. 8. A DD-MTV containing two levels of voting (ALT5).

To get a feel for the relative values of Q_{ALT5} and $Q_{n\text{VP}}$ and conditions under which the alternative scheme offers a higher reliability than nVP, consider the special case depicted in Fig. 3b ($n = 6$, $k = 3$). The relevant reliability equations in this case are:

$$Q_{6\text{VP}} = q^2(1 + 2p + 3p^2 - 16p^3 + 10p^4)$$

$$\begin{aligned} Q_{\text{ALT5}}\{n{:}6,\, k{:}3\} = {} & q^2[1 + 2p + 3p^2 - 3p^3 \\ & - p^2(1 + 2p)p' - 3p^2(1 - p)p'] \end{aligned}$$

To compare these reliabilities, let us compute their difference $\Delta Q = Q_{\text{ALT5}} - Q_{6\text{VP}}$:

$$\Delta Q = q^2p^2[p(13 - 10p) - (1 + 2p)p' - 3(1 - p)p'']$$

Therefore, for the alternative scheme to be better than 6VP, we must have:

$$(1 + 2p)p' + 3(1 - p)p'' < p(13 - 10p)$$

Observe that p'' is more important than p' in that it is multiplied by a larger factor. In the special case of $p' = p'' = s$, The condition above becomes $s < p(13 - 10p)/(4 - p)$ or $s < 3p + p(1 - 7p)/(4 - p)$. Thus, for p reasonably small, reliability improvement is guaranteed as long as s is no larger than $3p$.

One cannot draw general conclusions on the basis of a single example, but it is interesting to pinpoint the cause of the reliability improvement in this special case. Both 6VP and the scheme depicted in Fig. 3b produce the correct result when at least four M/T nodes are fault-free. To see this in the case of Fig. 3b, consider the following five cases which exhaust all possible double failures.

Case 1 – Two failures in $\{M_1, M_2, M_3\}$: Fault-free T rejects the incorrect voter output, reducing its weight from 2 to 1. Correct output is produced because M_4 and M_5 are both fault-free.

Case 2 – One failure in $\{M_1, M_2, M_3\}$ and one in T: T rejects the correct voter output, reducing its weight from 2 to 1. The output would be correct even if M_4 or M_5 were faulty.

Case 3 – One failure in $\{M_1, M_2, M_3\}$ and one in $\{M_4, M_5\}$: T accepts the correct voter output, increasing its weight to 3. The output is independent of M_4 or M_5.

Case 4 – One failure in T and one in $\{M_4, M_5\}$: T rejects the correct voter output, reducing its weight from 3 to 2. The fault-free module in $\{M_4, M_5\}$ creates a correct majority.

Case 5 – Two failures in $\{M_4, M_5\}$: T is fault-free and accepts the unanimous voter output, increasing its weight from 3 to 4. M_4 and M_5 cannot affect this output.

Cases 2 and 3 above show that some triple failures are also tolerated by the alternative scheme; hence the improved reliability. In certain instances, as in Cases 3 and 5 above, the output of T obviates the need for executing M_4 or M_5. These cases correspond to up to one fault in $\{M_1, M_2, M_3\}$, with T fault-free, and have a probability of $q^2(1 + 2p)(1 - s)$.

One should note that if there were no AT between the two voting levels in Fig. 8, reliability would actually degrade compared to a single-level scheme with the same number of modules. The reason is that correct minority results in the first level are discarded whereas they may help establish a correct majority if combined with correct outputs from the remaining modules. So the AT is a key component in this multilevel voting configuration. Multilevel voting without some form of intermediate validation is simply not beneficial.

6 Dealing with correlated failures

General analysis of various hybrid redundancy schemes with correlated failures becomes significantly more complex. In this section, we present a simplified analysis based on a highly pessimistic view of correlated failures: that they affect a set of modules and ATs in the worst possible way, causing the modules to produce identical incorrect results and an AT to reject any correct result and to accept any incorrect result. We obtain lower bounds for the reliabilities of pure and hybrid schemes and show the bounds corresponding to certain hybrid schemes to be higher. However, this does not necessarily imply that the hybrid schemes are more reliable, because $a > b$, $a' > b'$, and $b > b'$ do not imply $a > a'$. On the other hand, reliability of a complex system can never be computed exactly and we usually settle for lower bound guarantees. From this viewpoint, a system for which the lower-bound, or guaranteed reliability level, is higher must be considered better.

In what follows, we compare nVP and ALT3 configurations (see Fig. 4b) with regard to correlated failures and show ALT3 to be superior. The comparison is based on combinatorial analysis. Admittedly, the application of this method would be cumbersome for more complicated configurations. However, our aim here is to validate, in part, the intuition that the possibility of correlated failures does not alter our earlier conclusions. The insight gained from this example analysis will help us understand why replacement of some modules with ATs improves the probability of obtaining a correct result under both statistically independent and correlated failure scenarios.

Because nVP and ALT3 differ only in the use and placement of M_1, M_n, and T, our model postulates the occurrence of correlated failures in c modules among $\{M_2, \ldots, M_{n-1}\}$ and includes probability parameters relating to how M_1, M_n, and/or T may be affected. The parameters β, β', σ, μ, τ, ν, ν', defined below, should be interpreted as "probability of *event*, given that c modules among $\{M_2, \ldots, M_{n-1}\}$ contain correlated/common failures". Nodes unaffected by correlated failures can still suffer from random failures, with corresponding parameters as defined in Section 2. The events associated with the conditional probabilities for nVP and ALT3 are:

NVP	β	Both M_1 and M_n are affected
	σ	Single module: M_1 or M_n is affected
	ν	Neither M_1 nor M_n is affected
ALT3	β'	Both M_1 and T are affected
	μ	M_1 is affected but T is not
	τ	T is affected but M_1 is not
	ν'	Neither M_1 nor T is affected

Clearly, we have $\beta + 2\sigma + \nu = \beta' + \mu + \tau + \nu' = 1$. Also, given that two modules are more similar than a module and an AT, the following might be considered reasonable assumptions:

$$\tau \le \sigma \le \mu \quad \text{and} \quad \beta' \le \beta \le \sigma \le \nu \le \nu'$$

These inequalities are essentially the crux of our comparative evaluation, in much the same way that the assumption $p' + p'' < 2p$ was essential in proving improvements with independent failures. One last item of notation: Because the reliability of a k-out-of-$(n - c - 2)$ system, $R_{k,n-c-2}$, is used repeatedly in the following analysis, we denote it by R_k for brevity. Recall that h was defined as $\lfloor n/2 \rfloor$.

We next derive upper bounds on the reliability reduction due to correlated failures in nVP and ALT3. The "≈" sign denotes proportionality rather than equality.

$$\begin{aligned}\Delta Q_{n\text{VP}} &\approx \beta(1 - R_{h+1}) + 2\sigma[q(1 - R_h) + p(1 - R_{h+1})] \\ &\quad + \nu[q^2(1 - R_{h-1}) + 2pq(1 - R_h) + p^2(1 - R_{h+1})] \\ &= 1 - [(\beta + 2\sigma p + \nu p^2)R_{h+1} + 2q(\sigma + \nu p)R_h + \nu q^2 R_{h-1}]\end{aligned}$$

$$\begin{aligned}\Delta Q_{\text{ALT3}} &\approx \beta'(1 - R_{h+1}) + \mu[q''(1 - R_h) + p''(1 - R_{h+1})] \\ &\quad + \tau[q(1 - R_h) + p(1 - R_{h+1})] + \nu'[qq'(1 - R_{h-1}) \\ &\quad + p'q(1 - R_h) + pq''(1 - R_h) + pp''(1 - R_{h+1})] \\ &= 1 - [(\beta' + \mu p'' + \tau p + \nu' pp'')R_{h+1} \\ &\quad + (\mu q'' + \tau q + \nu' p' q + \nu' pq'')R_h + \nu' qq' R_{h-1}]\end{aligned}$$

$$\begin{aligned}\Delta Q_{n\text{VP}} - \Delta Q_{\text{ALT3}} &\approx (\beta' - \beta + \mu p'' + \tau p - 2\sigma p + \nu' pp'' - \nu p^2)R_{h+1} \\ &\quad + (\mu q'' + \tau q - 2\sigma q + \nu' p' q + \nu' pq'' R_h - 2\nu pq)R_h \\ &\quad + q(\nu' q' - \nu q)R_{h-1}\end{aligned}$$

To simplify the expression for $\Delta Q_{n\text{VP}} - \Delta Q_{\text{ALT3}}$, note the following equalities:

$$R_h = R_{h+1} + C_{n-c-2,h}\, q^h p^{n-c-2-h}$$

$$R_{h-1} = R_{h+1} + C_{n-c-2,h}\, q^h p^{n-c-2-h} + C_{n-c-2,h-1}\, q^{h-1} p^{n-c-1-h}$$

Substituting the preceding in the expression for $\Delta Q_{n\text{VP}} - \Delta Q_{\text{ALT3}}$, the coefficient for R_{h+1} becomes 0. Dividing both sides by $(n - c - 2)! q^h p^{n-c-2-h}/[h!(n - c - 1 - h)!]$, yields:

$$\begin{aligned}\Delta Q_{n\text{VP}} - \Delta Q_{\text{ALT3}} &\approx (n - c - 1 - h)[q(\mu q''/q + \tau - 2\sigma + \nu' - \nu) \\ &\quad + p(\nu' q'' - \nu q)] + hp(\nu' q' - \nu q)\end{aligned}$$

Because by our assumptions both $\nu' q'' - \nu q$ and $\nu' q' - \nu q$ are nonnegative, a sufficient condition for the difference $\Delta Q_{n\text{VP}} - \Delta Q_{\text{ALT3}}$ to be nonnegative is to have $\mu q''/q + \tau - 2\sigma + \nu' - \nu \ge 0$.

$$\begin{aligned}\mu q''/q + \tau - 2\sigma + \nu' - \nu &= \mu(q'' - q)/q + (\mu + \tau + \nu') - (2\sigma + \nu) \\ &= \mu(q'' - q)/q + (1 - \beta') - (1 - \beta) = \mu(q'' - q)/q + \beta - \beta'\end{aligned}$$

This last expression is nonnegative by our assumptions; hence $\Delta Q_{n\text{VP}} \ge \Delta Q_{\text{ALT3}}$ and the conclusion that correlated failures have a less serious effect on ALT3 than on nVP.

7 Conclusion

The methodology presented in this paper unifies previously proposed hybrid NVP-AT schemes and leads to many new variants. Given the extensive literature available in software fault tolerance and continued rapid developments in the field, such unifying methodologies (see, e.g., [49]) are clearly in demand and must be given high priority by the researchers in the field and within our educational programs. A component-based approach is particularly appropriate in that it allows:

- Easy exploration of the vast architectural design space for fault-tolerant software
- Building up of trust and the emergence of trusted components due to reuse
- Swapping of components for more reliable versions as they become available

Continued research in this area will enhance the utility of the proposed general framework for the study of hybrid NVP-AT schemes, leading to more specific design techniques, performance comparisons, and tradeoff guidelines. Results of such extended studies will contribute both to fundamental understanding of voting and acceptance testing as "dependability enhancement" mechanisms [29] and to their practical application in the realization of ultrareliable computations from standard components. A number of specific problems for future investigation are suggested directly by the discussions in the preceding sections of this paper. Examples of promising research directions include:

- Optimal weight augmentation and reduction policies; the $a(w)$ and $r(w)$ functions
- Effects of unequal module complexities and/or reliabilities as well as imperfect voters
- Effects of different voting schemes on optimal configurations and their reliabilities
- Optimal number of modules to be replaced by ATs (parameter k of Fig. 6a)
- Optimal partitioning of n modules for two-level voting (parameter k of Fig. 8)
- More general multilevel voting schemes and their attendant design tradeoffs
- Effects of combined correctness and timeliness requirements [28], [31]

The ultimate goal is to solve the following problem:

> Given a set of components with associated values for p, p', and p'', as well as other system cost and reliability parameters (in particular those characterizing correlated failures), what is the most cost-effective choice and arrangement of computation modules, ATs, and voters?

As this problem is quite challenging, any approach to its solution will necessarily proceed through a number of simpler intermediate problems. For example, one might ask: What is an optimal arrangement of n M/T modules to maximize the overall reliability?

8 References

[1] Ammann P.E. & Knight J.C. (1988) Data Diversity: An Approach to Software Fault Tolerance. *IEEE Trans. Computers*, Vol. 37, pp. 418-425.

[2] Avizienis A. & Kelly J.P.J. (1984) Fault Tolerance by Design Diversity: Concepts and Experiments. *IEEE Computer*, Vol. 17, No. 8, pp. 67-80.

[3] Avizienis A. (1985) The *N*-Version Approach to Fault-Tolerant Software. *IEEE Trans. Software Engineering*, Vol. 11, pp. 1491-1501.

[4] Banerjee P. & Abraham J.A. (1986) Bounds on Algorithm-Based Fault Tolerance in Multiple-Processor Systems. *IEEE Trans. Computers*, Vol. 35, pp. 296-306.

[5] Belli F. & Jedrzejowicz P. (1990) Fault-Tolerant Programs and Their Reliability. *IEEE Trans. Reliability*, Vol. 39, pp. 184-192.

[6] Blum M. & Wasserman H. (1996) Reflections on the Pentium Division Bug. *IEEE Trans. Computers*, Vol. 45, pp. 385-393, 1996.

[7] Butler R.W. & Finelli G.B. (1993) The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Trans. Software Engineering*, Vol. 19, pp. 3-12.

[8] Chen L. & Avizienis A. (1978) *N*-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation. *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 3-9.

[9] Di Giandomenico F. & Strigini L. (1990) Adjudicators for Diverse-Redundant Components. *Proc. Symp. Reliable Distributed Systems*, pp. 114-123.

[10] Dugan J.B. & Lyu M.R. (1994) System-Level Reliability and Sensitivity Analyses for Three Fault-Tolerant System Architectures. *Proc. Int'l Working Conf. on Dependable Computing for Critical Applications*, pp. 295-307.

[11] Dugan J.B. & Lyu M.R. (1994) System Reliability Analysis of an N-Version Programming Application. *IEEE Trans. Reliability*, Vol. 43, pp. 513-519.

[12] Eckhardt D.E. & Lee L.D. (1985) A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors. *IEEE Trans. Software Engineering*, Vol. 11, pp. 1511-1517.

[13] Eckhardt D.E. et al (1991) An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability. *IEEE Trans. Software Engineering*, Vol. 17, pp. 692-702.

[14] Gantenbein R.E., Shin S.Y. & Cowles J.R. (1991) Evaluation of Combined Approaches to Distributed Software-Based Fault Tolerance. *Proc. Pacific Rim Symp. Fault-Tolerant Systems*, pp. 70-75.

[15] Gersting, J.L., Nist R.L., Roberts D.B. & Van Valkenburg R.L. (1991) A Comparison of Voting Algorithms for N-Version Programming. *Proc. Hawaii Int'l Conf. System Sciences*, pp. 253-262.

[16] Kelly J., McVittie T. & Yamamoto W. (1991) Implementing Design Diversity to Achieve Fault Tolerance. *IEEE Software*, Vol. 8, pp. 61-71, July.

[17] Kersken M. & Saglietti F., Eds. (1992), *Software Fault Tolerance: Achievement and Assessment Strategies*, Springer.

[18]Kim K.H. & Welch H.O. (1989) Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults in Real-Time Applications. *IEEE Trans. Computers*, Vol. 38, pp. 626-636.

[19]Kim K.H. & Kavianpour A. (1993) A Distributed Recovery Block Approach to Fault-Tolerant Execution of Application Tasks on Hypercubes. *IEEE Trans. Parallel & Distributed Systems*, Vol. 4, pp. 104-111.

[20]Kim K.H. (1994) Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults. *Proc. Conf. Distributed Computing Systems*, pp. 526-532.

[21]Knight J.C. & Leveson N.G. (1986) An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming. *IEEE Trans. Software Engineering*, Vol. 12, pp. 96-109.

[22]Laprie J.-C., Arlat J., Beounes C., Kanoun K. & Hourtolle C. (1987) Hardware- and Software-Fault-Tolerance: Definition and Analysis of Architectural Solutions. *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 116-121.

[23]Laprie J.-C., Arlat J., Beounes C. & Kanoun K. (1990) Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *IEEE Computer*, Vol. 23, No. 7, pp. 39-51.

[24]Leveson N.G., Cha S.S., Knight J.C. & Shimeall T.J. (1990) The Use of Self Checks and Voting in Software Error Detection: An Empirical Study. *IEEE Trans. Software Engineering*, Vol. 16, pp. 432-443.

[25]Littlewood B. & Miller D.R. (1989) Conceptual Modeling of Coincident Failures in Multiversion Software. *IEEE Trans. Software Engineering*, Vol. 15, pp. 1596-1614.

[26]Liu C. (1992) A General Framework for Software Fault Tolerance. *Proc. Workshop Fault-Tolerant Parallel & Distributed Systems*, pp. 84-91.

[27]Nicola V.F. & Goyal A. (1990) Modeling of Correlated Failures and Community Error Recovery in Multiversion Software. *IEEE Trans. Software Engineering*, Vol. 16, pp. 350-359.

[28]Parhami B. (1990) A Unified Approach to Correctness and Timeliness Requirements for Ultrareliable Concurrent Systems. *Proc. Int'l Parallel Processing Symp.*, pp. 733-747.

[29]Parhami B. (1991) A Data-Driven Dependability Assurance Scheme with Applications to Data and Design Diversity. in *Dependable Computing for Critical Applications*, Ed. by A. Avizienis and J.C. Laprie, Springer, pp. 257-282.

[30]Parhami B. (1992) Optimal Algorithms for Exact, Inexact, and Approval Voting. *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 404-411.

[31]Parhami B. & Hung C.Y. (1993) Scheduling of Replicated Tasks to Meet Correctness Requirements and Deadlines. *Proc. Hawaii Int'l Conf. System Sciences*, pp. 506-515.

[32]Parhami B. (1994) A Multi-Level View of Dependable Computing. *Computers and Electrical Engineering*, Vol. 20, pp. 347-368.

[33]Parhami B. (1994) Voting Algorithms. *IEEE Trans. Reliability*, Vol. 43, pp. 617-629.

[34]Parhami B. (1996) Design of Reliable Software via General Combination of N-Version Programming and Acceptance Testing. *Proc. Int'l Symp. Software Reliability Engineering*, pp. 104-109.

[35]Partridge D. (1997) The Case for Inductive Programming. *IEEE Computer*, Vol. 30, No. 1, pp. 36-41.

[36]Prata P. & Silva J.G. (1999) Algorithm Based Fault Tolerance Versus Result-Checking for Matrix Computations. *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 4-11.

[37]Pucci G. (1990) On the Modeling and Testing of Recovery Block Structures. *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 356-363.

[38]Randell B. (1975) System Structure for Software Fault Tolerance. *IEEE Trans. Software Engineering*, Vol. 1, pp. 220-232.

[39]Randell B. (1987) Design Fault Tolerance. In *The Evolution of Fault-Tolerant Computing*, Ed. by A. Avizienis, H. Kopetz, and J.-C. Laprie, Springer, pp. 251-270.

[40]Scott K., Gault J.W. & McAllister D.F. (1983) The Consensus Recovery Block. *Proc. Total System Reliability Symp.*, pp. 74-85.

[41]Scott K., Gault J.W. & McAllister D.F. (1987) Fault-Tolerant Software Reliability. *IEEE Trans. Software Engineering*, Vol. 13, pp. 582-592.

[42]Scott R.K. & McAllister D.F. (1996) Cost Modeling of *N*-Version Fault-Tolerant Software Systems for Large *N*. *IEEE Trans. Reliability*, Vol. 45, pp. 297-302.

[43]Siewiorek D.P. & Swarz R.S. (1992) *Reliable Computer Systems: Design and Evaluation*, Digital.

[44]Sitaraman R.K. & Jha N.K. (1993) Optimal Design of Checks for Error Detection and Location in Fault-Tolerant Multiprocessor Systems. *IEEE Trans. Computers*, Vol. 42, pp. 780-793.

[45]Sklaroff J.R. (1976) Redundancy Management Techniques for Space Shuttle Computers. *IBM J. Research & Development*, Vol. 20, pp. 20-28.

[46]Sullivan G.F. & Masson G.M. (1990) Using Certification Trails to Achieve Software Fault Tolerance. *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 423-431.

[47]Sullivan G.F. & Masson G.M. (1991) Certification Trails for Data Structures. *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 240-247.

[48]Sullivan G.F., Wilson D.S. & Masson G.M. (1995) Certification of Computational Results. *IEEE Trans. Computers*, Vol. 44, pp. 833-847.

[49]Suzuki M., Katayama T. & Schlichting R.D. (1994) Implementing Fault Tolerance with an Attribute and Functional Based Model. *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 244-253.

[50]Tai A.T., Meyer J.F. & Avizienis A. (1993) Performability Enhancement of Fault-Tolerant Software. *IEEE Trans. Reliability*, Vol. 42, pp. 227-237.

[51]Voges U., Ed. (1988) *Software Diversity in Computerized Control Systems*, Springer.

[52]Xu J. & Randell B. (1997) Software Fault Tolerance: $t/(n-1)$-Variant Programming. *IEEE Trans. Reliability*, Vol. 46, pp. 60-68.

Evolution of fault-prone components in legacy systems: a case study

Magnus C. Ohlsson
Dept. of Communication Systems
Lund University, Box 118
SE-221 00 Lund, Sweden
Magnus_C.Ohlsson@telecom.lth.se

Keywords: software evolution, fault-prone components, prediction, tracking, legacy systems

Received: June 29, 2001

Prediction of problematic software components is an important activity today for many organisations as they manage and maintain their legacy systems and the maintenance problems they cause. This means that there is a need for methods and models to identify problematic components. We apply a model for classification of software components as green, yellow and red according to the number of times they required corrective maintenance over successive releases. Further, we apply principal component analysis (PCA) and box plots to investigate the causes for the code decay and structural changes. The case study includes five system releases and 80 software components. A large set of non fault-prone components was identified. The system did not contain any large structural changes, which was indicated by the PCA and the box plots. Most of the changes were small fault corrections. A number of design improvement suggestions had been identified by the developers but not carried out. Overall, the model was successful in identifying the most problematic components and provided information about the evolution of the system. The strength of the model was the combination of both a short-term view and a long-term view.

1 Introduction

Many of the systems around us today are what we refer to as legacy systems, i.e. systems that evolve and go through a number of maintenance releases and naturally inherit functionality and characteristics from previous releases. The maintenance releases could include both corrective actions and enhancement with new functionality. Even though a system is being improved, both from a quality view and from a functional view, some parts of the systems may be difficult to maintain due to different reasons. Examples are lack of documentation, poor processes and deteriorating architectures. Another reason could be adding new functionality, resulting in components with high complexity and problematic relationships. To avoid problems with this type of components, it is necessary to identify them and keep track of their evolution, i.e. both have a short-term view and a long-term view.

A number of models have been presented to classify components and to predict whether they will be fault-prone in the future. Most of them have a short-term view and are based on the outcome from one project, validated for a second project and finally used in a third project and refined based on the outcome. Another approach has been to take the outcome from one project and divide the data set into two parts and build the model based on one half and validate it for the other half or build the model in one iteration and test it in the subsequent. Further, most models are based on statistics, for example, Principal Components Analysis [1], Boolean Discriminant Analysis [2], Spearman Rank Correlation [3], Optimised Set Reduction [4], Regression Analysis [5] and Classification Trees [6][7].

An important issue is to enable practitioners from industry to use the models and therefore they should be easy to use but still be capable of performing well. Therefore our objectives of this study are to create models that are easy to use and embody a long-term view. Another objective is to further evaluate and refine existing methods, which we propose to do. The approach used in this paper is based on how components evolve over successive releases to capture both the short-term view and, most important, the long-term view. Instead of using only two successive releases we base our models on a number of successive releases. This should handle different problems like, for example, fluctuations between releases.

The paper is organised as follows. In Section 2 we present the background for the approach presented in Section 3. The approach is illustrated in a case study is in Section 4. Finally, a summary can be found in Section 5.

2 Background

2.1 Classification Model

To enable identification of the problematic legacy components, we use a model for classification of software components based on fault-proneness [8]. The components are classified according to a colour code depending on the amount of decay. The components should be classified as green, yellow or red. The amount of decay should be judged based on the outcome of previous releases and the criterion is the number of faults. Other possible criteria are time to perform certain types of maintenance activities or that the structure of the component is becoming more and more difficult to understand and handle. An important issue is to use available data instead of launching a new measurement program. This could be argued about, but our concern has been to not increase the burden for the developers and instead to use existing data to calculate our measures. The number of faults is therefore often appropriate. The colouring scheme denoted GYR, which is illustrated in Figure 1, should be interpreted as follows:

- **Green components (normal evolution).** Green represents normal evolution and some amount of fluctuation is normal. These components are easily updated, i.e. new functionality may be added and faults corrected without too much effort. Furthermore, we do not need system experts to maintain the components. The components should be tracked from release to release to be able to find trends and when a component exceeds a certain limit (referred to as the lower limit) it becomes yellow.
- **Yellow components (code decay).** As a component exceeds the lower limit, it is classified as yellow, so particular attention has to be paid to this component to avoid future problems. Components in the yellow region are candidates for specific directed actions. These may include launching a more thorough development process or identifying a component as a candidate for reengineering. If the yellow components are not treated properly, the components may exceed the upper limit and become a red component.
- **Red components ("mission impossible").** A red component is difficult and costly to maintain. It is often the driving factor for schedules and cost. In other words, the red components have a tendency to end up on the critical path of a software project. In order to change the components, we need experts, and the components are often viewed as "mission impossible" tasks. The components are no longer candidates for reengineering; they need reengineering.

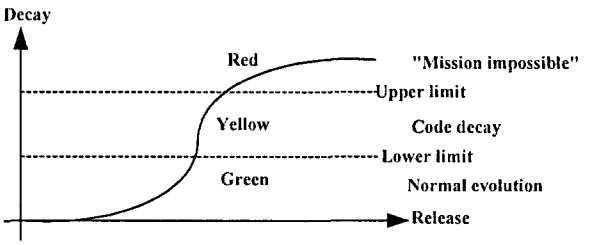


Figure 1. Growing amount of decay for a legacy component [8].

The classification scheme embodies two limits, one lower and one upper, to be able to separate the three classes. These limits should be determined based on the historical data and continuously updated according to the fact that we improve the quality of the components. It is, of course, not possible to state generally where the limits are located. The limits are governed by our interpretations of green, yellow and red legacy components, and they are dependent on factors, which must be determined for each case separately. The interpretation may depend on, for example, company, application domain, system and customer requirements.

It is important to have a long-term view even though some components indicate a high level of decay because there might be confounding factors or the release as a whole affecting the results, e.g. lack of resources, unsatisfactory process etc. For example, we may have found that components with more than a certain number of faults in testing should be classified as yellow, but when looking at the system as a whole we realise that the total number of faults is very high. This may indicate that the component as such is not the problem, we may have a problem with the release as a whole. Therefore, the whole system must be studied in conjunction with the individual components prior to finally identifying certain components as being of a specific class. One possibility is to visualise the trends by plotting the degree of decay for each release and graphically make a decision (see figure 2). Another possibility is to use rankings or standardisation to handle the variations.

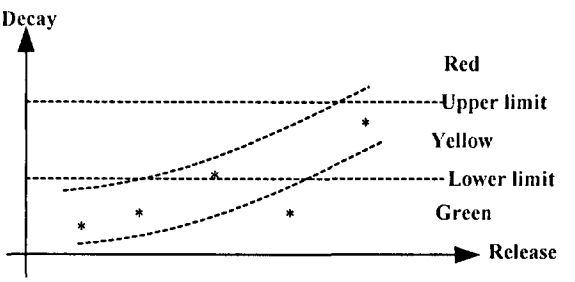


Figure 2. Trend for a number of different releases [8].

As mentioned earlier, decay could depend on structural changes. For example, new components, inheriting from other classes, are added and cause problems because of coupling problems or lack of cohesion. An important part is therefore to find and analyse structural changes in the code. The GYR model proposes to use Principal

Components Analysis (PCA) [9]. This analysis method groups a number of correlated variables into a number of factors. Changes in the number of factors and variables changing factors are indicators that the system is not stable and indicate areas for further analysis.

2.2 Evaluation

To evaluate how well a model performs it is necessary to have some criterion that relates to its ability to point out components as fault-prone and non fault-prone. Here we use two types of errors that may be conducted when trying to predict fault-prone components:

- Committing a Type I error means classifying actually fault-prone components as non fault-prone.
- Committing a Type II error means classifying actually non fault-prone components as fault-prone.

The rate of the two error types as such has been used as an evaluation criterion [2][6]. The approach, which minimises the misclassifications, is then considered the best. The overall misclassification is the total number of the misclassifications of Type I and II, normalised by the total number of components. The criteria can be found in Table 1.

		Prediction from Release *n*	
		Non fault-prone	Fault-prone
Outcome of Release *n+1*	Non fault-prone	A	B
	Fault-prone	C	D
Type I rate: C/(C+D)% Type II rate: B/(A+B)% Overall misclassification: (B+C)/(A+B+C+D)%			

Table 1. Evaluation criteria.

3 Approach

The approach used in this paper is an extension of the GYR model [8], described above, and includes five main steps, which are applied to the case study in Section 4. The different steps in the approach try to identify problematic components, both from a short-term view and a long-term view, to reveal structural changes and to find indicators of decay among software components. The steps are the following:

1. **Determine variables to include in the analysis.** Based on the purpose of the models, different criteria for inclusion of variables exist. For code decay, the most interesting variables to include are the number of reported faults and those related to the nature and size of code changes, including structural changes.
2. **Identify most fault-prone components (short-term view).** The short-term view first classifies fault-prone components in each release and ranks them according to the fault reports written against them. The advantage of using ranks instead of number of faults is that differences, related to number of faults reported in the releases, are nullified. Components among the top 25 percent in terms of fault reports written against the component are considered fault-prone. Thus all components whose number of fault reports is in the upper quartile are considered fault-prone. In case of ties in ranking that would cause more than 25 percent of the components to be included, the smaller set is chosen. Secondly, looking at two consecutive releases, we analyse how components change status according to this classification. Components that are fault-prone in two successive releases are considered red, those that are not fault-prone (normal) in either release are considered green. Components that change status (from fault-prone to normal or from normal to fault-prone) are classified as yellow.
3. **Identify most fault-prone components (long-term view).** The long-term view utilises the same concepts as the short-term view except for how the rankings are handled. With the long-term view the rankings are accumulated after each release. The advantages are that fluctuations between the releases are taken care of and components with rankings in the middle segment or close to the fault-prone threshold that consistently generate faults but are not considered fault-prone, are identified. The classification criterion for green, yellow and red are the same as for the short-term view, i.e. components which are fault-prone in two successive releases are considered red, those that are not fault-prone in either release are green and components that change status are classified as yellow.
4. **Analyse structural changes.** PCA on the variables of interest determines which groups of variables are related and how this relationship changes across releases. For code decay analysis, analysis of structural changes may help maintainers to identify and to react to major changes in the components. Changes in the system's structure might be indicators of future problems regarding the ability to change the component. Interesting indicators may be when the number of factors increases or decreases, or when some variables switch between PCA factors over successive releases. We intend to use PCA to reveal relationships and characteristics of changes made to the system.
5. **Analyse variable distributions.** To further analyse the reasons for code decay and structural changes, box plots should be created. They visualise the distribution of variables and aid the interpretation. They can be very useful to identify outliers and, more specifically, factors that may affect decay among the software components.

An important part of this study is also to further evaluate the usefulness of the GYR model when it is used in different domains. The model was developed based on a Telecommunication system [8]. In this study we apply it to a completely different system to further evaluate and refine the model.

4 Case Study

4.1 Step 1: Environment

This case study is based on data from 80 software components and five releases of a development tool for developing real-time software. It utilises the object-oriented design language SDL [10] and the trace language MSC [11] to generate code for integration with several real-time operating systems. The system studied is one of the company's main products and each sixth month a new version is released (for more information see [12]). Therefore, the releases investigated are considered typical for releases of this particular system.

The development tool is mainly developed with C and C++ although it contains some automatically generated code. The approximately size of the whole system is 5.000 KLOC. For each of the components, the number of fault reports from fault handling and test cases are counted from their internal fault database. A disadvantage is that some of the faults reported are postponed and not assigned until a subsequent release and might therefore be counted twice. The reason for including a postponed fault is that it is considered a fault as long as it is evident.

To collect data from the code we used Logiscope® from Telelogic. Logiscope® parses the code and collects a large amount of different measures, which describes different aspects of the code. Since some of them are not related to our objectives of this study we have selected the most appropriate ones. The selection can be found in Table 2 and includes class specific, inheritance and use-graph related metrics that relate to the length of a chain of use and usage of other classes [13]. Together with the number of faults, 19 different measures were collected for the modules included in the system.

Measure		Description
class	cl_cyclo	Sum of the methods static cyclomatic complexities in a class
	cl_data_class	Number of class type attributes for a class
	cl_data_vare	Total number of times attributes which are external to a class are used by the class's methods
	cl_data_vari	Total number of times a class's attributes are used by the class's methods
	cl_func_calle	Total number of calls from a class's methods to functions defined outside a class
	cl_func_calli	Total number of calls from a class's methods to member functions of the same class
	cl_dep_deg	Class coupling is an indicator of the degree of dependency of a class
	cl_dep_meth	Number of methods (within a class) which depend on other classes
	cl_locm	Lack of cohesion of methods
	cl_data_inh	Number of inherited attributes
	cl_func_inh	Number of inherited methods
Inheritance graph	in_depth	Depth of inheritance tree
	in_dderived	Number of classes which directly inherit from a class
	in_derived	Total number of classes which inherit from a class directly or not
Use graph	cu_level	Maximum length of a chain of use starting from a class
	cu_cused	Number of classes used by a class directly or not
	cu_cusers	Total number of classes which directly use a class or not
Faults		Number of fault reports from fault handling and test.

Table 2. Measures collected.

4.2 Fault-Prone Components

4.2.1 Step 2: Identification with a Short-Term View

The purpose of the short-term prediction is to evaluate our hypothesis that it is always the same components that are among the most fault-prone ones. Of course it is natural to have some fluctuation because new components may be integrated in the system or some may be detached or replaced. This should be considered when doing the predictions, but in this case study the information has not been available. A way of using the GYR model, with a short-term view, is to define green, yellow and red components as follows:

- Green – not in the fault-prone quartile for the two consecutive releases.
- Yellow – in the fault-prone quartile for one of the two releases.

- Red – in the fault-prone quartile for both releases.

Compared to Table 1, this means that the number of green components can be found in cell A, the yellow components in the B and C cells, while number of red components can be found in cell D.

	Release 1 to Release 2	Release 2 to Release 3	Release 3 to Release 4	Release 4 to Release 5
Correct non f-p (A)	61	56	55	55
Correct f-p (D)	13	12	13	11
Type I error (C)	4	7	6	6
Type II error (B)	2	5	6	8
Type I rate	23.5%	36.6%	31.6%	35.3%
Type II rate	3.2%	8.2%	9.8%	12.7%
Overall misclassification	7.5%	15%	15%	17.5%

Table 3. Prediction results.

Table 3 show the results for all five releases with a short-term view. The table includes all information from Table 1 but in more compact format. The results indicate an increasing problem of pinpointing fault-prone components because the Type II error (B) increases, i.e. it becomes more and more difficult to pinpoint the right components as fault-prone. The amount of red components (D) is stable between 11 and 13 components, which also is true for the green components (A). The ones that are problematic to predict are the yellow ones (B and C), i.e. Type I and Type II errors. It is those components that should be further investigated to find out what the reasons have been for the shift in classification, i.e. why they became fault-prone or what activities improved them to "green" status.

Type II error indicates the number of components that were improved from fault-prone to non fault-prone. It could be seen as an indication of improving quality. The major problem is components that decay from being normal to fault-prone, i.e. Type I error. The Type I rate is fairly high (23.5 percent to 35.3 percent) and should be further investigated. As mentioned earlier, one reason can be that it was new components that were integrated into the system and therefore could not have been classified as fault-prone earlier. Unfortunately, no information was available about components that were new (or deleted) in each release.

4.2.2 Step 3: Identification with a Long-Term View

To avoid to problems with the Type I and Type II errors, which are mostly correlated to the fluctuations between the releases, we propose to accumulate the rankings. Accumulated rankings have two advantages. First, components with rankings in the middle segment or close to the fault-prone threshold that consistently generate faults but are not considered fault-prone, are identified. Secondly, fluctuations among components that, for example, are classified as fault-prone in Release *n*, non fault-prone in Release *n+1* and finally fault-prone in Release *n+2* are smoothed out and only the consistently fault-prone components are highlighted.

The disadvantage is that new components from later releases, which consistently also are fault-prone, never are able to accumulate rankings so that they will be among the most fault-prone ones with the long-term view. But these components will be visible with the short-term view or as soon as the most fault-prone components from previous releases are taken care off. Another possibility could be to normalise a component's accumulated ranking according to the number of releases it was evident in. This has not been done because of lack of information.

The results from using the accumulated rankings are summarised in Table 4, which could be mapped to Table 1. For example, in column 3 the accumulated rankings from Release 1 and Release 2 are used to predict the outcome from Release 3, which are the rankings from Release 1 to Release 3. For column 1 this means that the rankings from Release 1 are used to predict the accumulated rankings from Release 1 and Release 2.

	Release 1 to Release 2	Release 2 to Release 3	Release 3 to Release 4	Release 4 to Release 5
Correct non f-p (A)	62	61	60	60
Correct f-p (D)	15	18	18	18
Type I error (C)	3	1	1	1
Type II error (B)	0	0	1	1
Type I rate	16.7%	5.3%	5.3%	5.3%
Type II rate	0%	0%	1.6%	1.6%
Overall misclassification	3.8%	1.3%	2.5%	2.5%

Table 4. Accumulated prediction results.

Compared to the results in Section 4.2.1, the misclassification rates are much lower and more stable. In column 2 the values have not stabilised, but as more releases and rankings are accumulated the trend stabilises even though there are two components in columns 4 and 5 that are misclassified. Table 5 shows the range of fault reports for the twenty most fault-prone components and their rankings in Release 5. The ten least fault-prone components all had an accumulated ranking of 23.

Problematic	61-70
Accumulated ranking	238-309
Problematic	71-80
Accumulated ranking	329-386

Table 5. Accumulated rankings for the components.

We can see that the threshold from the tenth most fault-prone components and down decreases, which support our hypothesis that the same components are the most fault-prone from release to release. It is noticeable that the maximum number is 400 and that 386 is very close to this value, which also is an indication that supports the hypothesis.

In this case we have not specified the upper and lower limits as presented in Section 2.1 but it could easily be done and further evaluated. For example, the maximum ranking a component can get in a release in this study is 80, which means that after two releases it is 160 and after five it is 400 (as mentioned above). Again we can use the quartiles and say that the upper limit is 75 percent of the maximum ranking and include the 50 percent quartile as the lower limit. The limits may not be the best ones and could be chosen differently, but in this case it is chosen to illustrate the classification method. The results can be found in Table 6. Release 1 is not included since no rankings could be accumulated.

	Rel. 2 Upper: 120 Lower: 80	Rel. 3 Upper: 180 Lower: 120	Rel. 4 Upper: 240 Lower: 160	Rel. 5 Upper: 300 Lower: 200
Red	16	13	12	11
Yellow	4	12	14	16
Green	60	55	54	53

Table 6. Results from using upper and lower limits.

The results show that the red components are practically stable while more and more components are classified as yellow. These components should be paid attention to before it is too late. A disadvantage with limits is that there are always components that are just below a limit, which actually need special attention. This is the main problem using classification models, but being aware of the disadvantages and the fact that many problematic components are identified makes it beneficial.

4.3 Step 4: Structural Changes

PCA is a statistical method that groups correlated variables in factors. This eases the process of identifying underlying structures in complex data sets. The basic idea behind using PCA is that a major change in the structure may be a potential source of future problems. It should, however, be remembered that we do not expect that this type of model gives us necessarily the optimal prediction in terms of problematic components. An expert must take a closer look at the components being depicted with this type of model, or for that matter, any other model. We do not expect to replace the expert, but to point the expert in the right direction, guide further work with the system and provide better understanding of the system's evolution.

To extract the factors we applied PCA with an orthotran/varimax transformation and only extracted factors with an eigenvalue larger than one or until the explained variance was at least 75 percent. Dark grey areas indicate the factors where the variables have the highest factor loadings, i.e. correlation between variable and factor. Light grey areas indicate variables with loadings higher than 0.5.

The system analysed in this study is large and analysing all of the components would be a too extensive task. Therefore, we have focused on some of the components that are considered fault-prone and some that are not. The results from analysing the files in Component A, which is one of the larger and more fault-prone ones, can be found in Table 7. The differences among the releases are very small and therefore the result from one release is provided.

	Factor			
Variable	1	2	3	4
cl_cyclo	.984	-.009	.015	.016
cl_data_class	.667	-.122	.155	-.484
cl_data_vare	.882	-.017	-.025	.045
cl_data_vari	.934	-.061	.042	-.065
cl_func_calle	.989	-.003	.011	.013
cl_func_calli	.974	-.010	.013	.032
cl_dep_deg	.957	.016	.024	-.008
cl_dep_meth	.993	.005	.035	-.011
cl_locm	.075	.751	.108	.200
cl_data_inh	-.418	.836	-.072	.265
cl_func_inh	-.069	.609	-.085	.429
in_depth	-.142	.826	-.156	.121
in_dderived	.040	.043	.840	.130
in_derived	.004	-.021	.886	.006
cu_level	.079	.870	.171	-.282
cu_cused	.100	.876	.187	-.320
cu_cusers	.098	.093	.308	.536
Eigenvalue	7.039	3.906	1.687	.978
Accumulated variance	41.4%	64.4%	74.3%	80.1%

Table 7. PCA result from Release 1 for Component A.

Among the releases there are small variations in the factor loadings and the factors' explained variance. The only variables that are not stable are cu_level, cu_used and cu_cusers, which are use-graph metrics (see Section 4.1). In Release 2 and 3, cu_users are not grouped with any of the variables, while in Release 4 and 5 it has a negative factor loading related to the other two variables,

i.e. if the number of classes used by a component increase the number of classes that use it (directly or not) decrease. The reason why cu_level and cu_cused have high loadings in two factors are difficult to tell but should be further investigated even though the variations are not that large.

In Component B (it is also one of the more fault-prone components) the pattern is a little bit different. In Table 8 the result from Release 3 is presented. The results from the other releases are again very similar except for Release 2, which had one more factor. The reason why Release 2 is different could be an indication of some large changes made to the system. Compared to Component A, some of the variables are grouped in separate factors. For example, cl_data_class and cl_data_vari form a factor on their own and are related to the class attributes and how the class uses them.

	Factor				
Variable	1	2	3	4	5
cl_cyclo	.956	.154	.008	.169	.058
cl_data_class	.153	.018	-.006	.937	.070
cl_data_vare	.896	.027	-.051	-.107	-.077
cl_data_vari	.369	.059	.026	.883	.013
cl_func_calle	.965	.190	.001	.107	.043
cl_func_calli	.836	.105	-.001	.387	.051
cl_dep_deg	.880	.246	-.006	.140	.045
cl_dep_meth	.876	.227	.051	.319	.082
cl_locm	.069	.644	.025	-.046	.059
cl_data_inh	-.007	-.048	.028	-.018	.889
cl_func_inh	.106	.424	-.005	.123	.775
in_depth	.040	.654	-.040	.063	.598
in_dderived	-.005	.060	.933	.002	.028
in_derived	-.010	.098	.932	.014	-.015
cu_level	.266	.845	-.003	.116	.101
cu_cused	.303	.823	.001	.126	.060
cu_cusers	.072	.736	.202	-.035	.044
Eigenvalue	6.719	2.833	1.768	1.562	1.171
Accumulated variance	39.5%	56.2%	66.6%	75.8%	82.7%

Table 8. PCA result from Release 3 for Component B.

Looking at some of the other components in the system the results is almost identical to Component A, i.e. very stable factors. This could have two explanations. Firstly, since PCA groups correlated variables the changes made to the system are of the same magnitude and do not affect the structure of the components and the relationships between them. Secondly, the changes are small patches and fault fixes that do not affect the overall structure or the underlying architecture. In discussions with people from Telelogic it seems that the reason for the stable structure depends on the fact that most of the changes are patches according to some faults reported. But many improvements have been proposed to improve the structure but not carried out. The problem is, as in most cases, lack of time and resources.

4.4 Step 5: Data Distribution

To further investigate changes to the system in more detail we compare the actual measurements or distributions of these variables. Figure 3 and Figure 4 show box plots of some of the variables for the releases. The variables were selected based on the results from the PCA. For example, cu_level was included because of its high loadings in two factors (see Section 4.3). A box plot shows the 25th and 75th quartiles as a box with the 50th (the median) as a line in the box. The 10th and 90th quartiles are shown as vertical lines and the small circles are outliers [14].

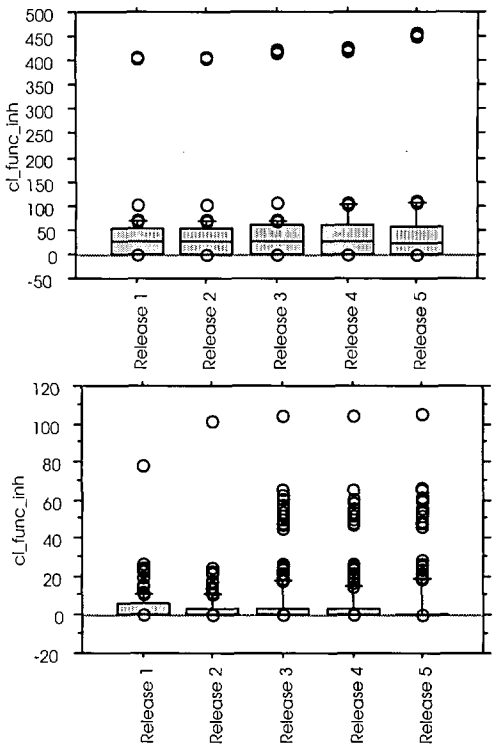


Figure 3. Box plots of cl_func_inh for Component A and Component B.

In Figure 3, the results for cl_func_inh are presented. This variable describes the number of inherited methods, which are not overridden in that class. Again there are some outliers and the values of Component A's classes is higher than for Component B. For Component A the values are very stable. For Component B there is a group of classes separated from the rest in Release 3 and onwards. The box is also becoming smaller and smaller, which is an indication of that more and more classes are overridden and therefore this subset of classes is separated from the rest.

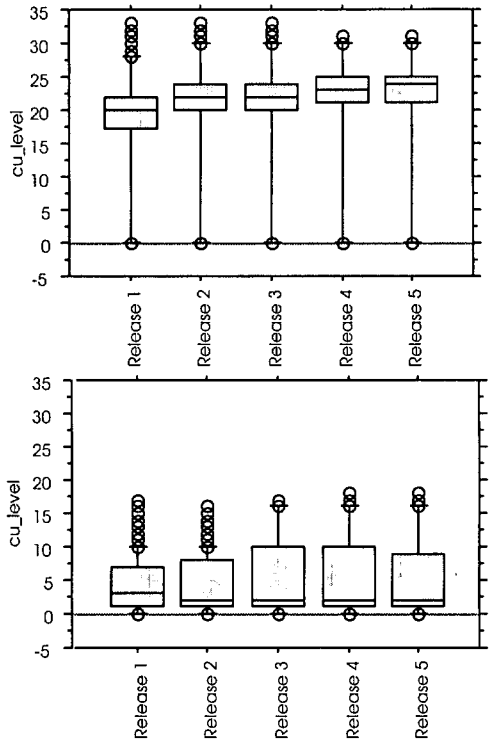


Figure 4. Box plots of cu_level for Component A and Component B.

In Figure 4 the distributions of cu_level are presented to further show the difference between the components. It is a use-graph metric describing the maximum length of a chain of use. For Component A it continuously increases over the releases. We can also see that the median is increasing from Release 1 to Release 5. For Component B it is the opposite where the median instead decreases even though the upper quartile increases and finally stabilises. Overall it is a fairly stable system.

5 Summary

Identification and prediction of fault-prone components are very important tasks to be able to direct effort to improve quality. It is also important to identify causes for code decay to understand and avoid future problems by identifying problematic constructions.

This paper reported on a study where the objectives were to focus on the evolution of software components and possible causes for fault-proneness and decay. The purpose was also to use an existing model to further evaluate its usefulness and appropriateness in different domains. Therefore we extended a model that classifies components as green, yellow and red depending on how fault-prone they were over successive releases, and finally applied it in a case study.

The study also investigated structural changes and tried to identify causes for the problems. The approach applied in the study used PCA to reveal structural characteristics of the changes made to the components and box plots to highlight and visualise distributions of different measures. The fault data used in this study was extracted from an internal fault database, while the structural data was extracted with Logiscope®. The data came from five successive releases.

The result, for fault-prone components with a short-term view, shows some variations between the releases. The overall misclassification rate varies between 7.5 percent and 16.5 percent. Some of the variation could be explained by the components added to the system. These components were sometimes fault-prone in the release where they were introduced and not fault-prone in the successive release. With a long-term focus it was possible to identify the most problematic components in the releases and avoid the previous mentioned problems.

Both the short-term and the long-term view have advantages and disadvantages. The short-term view helps focus on components that are problematic for the moment while the long-term view captures the consistently problematic ones even though they not necessarily are among the most fault-prone ones in each release. The strength is to use both views in conjunction with each other to capture different aspects and for guiding improvements.

The second part of the approach, structural changes, did not provide any novel results. The different components analysed had very stable factors with only small variations in factor loadings and variable groupings. There were only some small differences between the components. This was an indication of a stable structure of the components (or even the system as a whole). The box plots also confirmed this. In discussions with people at Telelogic we found that most of the changes only affected small parts of the components, i.e. a few lines of code. Developers have identified code parts and classes, which could be re-designed and if these changes are carried out the PCA and box plots should indicate this.

Finally, the approach has overall been successful in identifying that the most fault-prone components in one release are the same in the successive. Even if the structural analysis only showed stable components, it is still valuable to carry out this part and especially in the future if the system is re-designed to be able to keep tracking the evolution of fault-prone components.

6 Acknowledgement

We would like to thank VINNOVA for partly funding the research under grant for LUCAS, Center for Applied Software Research at Lund University and all the people at Telelogic for their support, help and fruitful discussions. We would also like to thank Claes Wohlin for his review comments.

7 References

[1] T.M. Khoshgoftaar, E.B. Allen, K.S Kalaichelvan and N. Goel, "Early Quality Prediction: A Case Study in Telecommunications", IEEE Software, pp. 65-71, January 1996.

[2] N.F. Schneidewind, "Software Metrics Model for Integrating Quality Control and Prediction", Proceedings of the International Symposium om Software Reliability Engineering, pp. 402-415, November 1997, Albuquerque, New Mexico, USA.

[3] N. Ohlsson, M. Helander and C. Wohlin, "Quality Improvement by Identification of Fault-Prone Modules using Software Design Metrics", Proceedings of the International Conference on Software Quality, pp. 1-13, 1996, Ottawa, Ontario, Canada.

[4] L.C. Briand, V.R. Basili and C.J. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High-Risk Software Components", IEEE Transactions on Software Engineering, 19(11), pp. 1028–1044, 1993.

[5] M. Zhao, C. Wohlin, N. Ohlsson and M. Xie, "A Comparison between Software Design and Code Metrics for the Prediction of Software Fault Content", Information and Software Technology, 40(14), pp. 801-809, 1998.

[6] T.M. Khoshgoftaar, E.B. Allen, W.D. Jones and J.P. Hudepohl, "Assessing Uncertain Predictions of Software Quality", Proceedings of the International Software Metrics Symposium, Metrics'99, pp. 159-168, November 1999, Boca Raton, Florida, USA.

[7] T.M. Khoshgoftaar, E.B. Allen, W.D. Jones and J.P. Hudepohl, "Classification Tree Models of Software Quality Over Multiple Releases", Proceedings of the International Symposium on Software Reliability Engineering, ISSRE'99, pp. 116-125, November 1999, Boca Raton, Florida, USA.

[8] M.C. Ohlsson and C. Wohlin, "Identification of Green, Yellow and Red Legacy Components", Proceeding of International Conference on Software Maintenance, ICSM'98, pp. 6-15, November 1998, Bethesda, Washington D.C., USA.

[9] R.L. Gorsuch, "Factor Analysis", 2:nd edition, Laurence Erlbaum Associates, Hillsdale, New Jersey, 1983.

[10] ITU-T, "Standard Z.100: SDL - Specification and Description Language", International Telecommunication Union, 1992.

[11] ITU-T, "Recommendation Z.120: MSC - Message Sequence Chart", International Telecommunication Union, 1996.

[12] B. Regnell, P. Beremark and O. Eklundh, "A Market-Driven Requirements Engineering Process - Results from an Industrial Process Improvement Programme", Requirements Engineering Journal, 3(2), pp. 121-129, 1998.

[13] Telelogic Technologies Toulouse, "Logiscope C++ Audit – Reference Manual", Version 5.0, 2000.

[14] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell and A. Wesslén, "Experimentation in Software Engineering – An Introduction", Kluwer Academic Publishers, 1999.

The need for speed: a practitioner's view of rapid application development in e-business

Patricia Carando
IT Essential Technologies
43024 Hedgeapple Court, Ashburn, VA 20147 USA
Phone: 703.724.4367, Fax: 703.724.4367
carando@aol.com

Keywords: e-business, reuse, software engineering

Received: June 10, 2001

This paper reflects a practitioner's view on the present state of component-based software development in eBusiness; the observations are motivated by a recent software effort that created an eBusiness trading system. We observe which types of components significantly increased the speed of our development and which types, while promising, did not realize their potential. Of particular concern here are the reasons the promising components did not work in our application; we reflect on what issues should be addressed when attempting to employ components of this type.

1 Introduction

A Practitioner's View. For the modern practitioner of software development, the current landscape of possibility is huge. There is a plethora of components, tools, and techniques from which to choose in developing new software applications. These components-tools-techniques all promise to enhance the software process in some way; and most promise to enhance the speed at which applications can be built, because speed of development is a live-or-die proposition for nearly all technology companies.

This pressure has acted as a kind of evolutionary force on the way we do software development. In the 10+ years since this practitioner was a researcher in software reuse, burning issues in software engineering and reuse research have become commonplace realities. The controversy over object technology has all-but disappeared as Java captures the hearts and minds of today's programmers. With it, the "reflex to reuse"—the initial impulse to *find* a component that does what you need rather than *write* it—has spread from tiny communities, like Smalltalk users or Common Lisp users, to the sizeable community of Java.[1] Large-scale architectures like the Software Backplane™ [1] and the Portable Common Tools Environment (PCTE) [22] that supported "pluggable" components for CASE tools, have given way to the more general architectures like the Common Object Request Broker Architecture (CORBA) [20,21] and the Java 2 Platform, Enterprise Edition (J2EE) [24].

[1] It is likely that the ready-availability of reusable components for these languages played a significant part in the active reuse within these communities; but perhaps an equally strong force was the visual development environments that made the search for and understanding of the components a transparent process.

Successes and Non-Successes. While this paints a rosy picture, not all changes were successes. Some of these changes might be called *non-su*ccesses rather than failures because, although they contributed in many ways to software development, they failed to hit their target—impacting the cost or speed of software production.

One example of success is the architectures for CORBA and J2EE. These have given the multi-user system developer a big boost in capability. Before the implementations of these standards, this practitioner built numerous systems that required custom solutions for marshalling and de-marshalling, multi-threading for concurrent users, relational database access, and database connection pooling. The availability of CORBA and the CORBA services or J2EE could have reduced these time-consuming coding efforts to one of declaration: marshalling and de-marshalling reduced to defining IDL interfaces for communicating processes; multi-threading for concurrent users reduced to choosing which policy to employ; relational database accessed reduced to utilizing JDBC [25] calls; database connection pooling reduced to specifying the parameters for database login and the size of connection pools. The availability of implementations of these architectures has been a big win in rapid application assembly.

An example of a non-success is object technology. Touted as a foundation for reuse and enhanced productivity, it didn't pay off as anticipated. Object technology supports encapsulation, believed to be an effective means of creating *Software ICs* [11,12]—integrated components for software. It was hoped that these Software ICs, like their computer hardware

analogues, would promote standardization and use of interchangeable parts, so that software could be assembled, not crafted. The use of inheritance as a means of customizing an existing component was seen as another form of reuse, allowing a class designer to capitalize on the work of others. These capabilities of the technology have--in the author's opinion—paid off in terms of better design and *perspicuous representation*; they have not paid off in terms of cost. Research has indicated that there was no real impact to the cost of software in using the small, concrete components supported by the reuse of object classes [6]. The Software ICs were not sufficiently significant contributors to the effort of system building to impact its overall cost.

Choosing Wisely. Awareness of what are major enablers of speedy development and what are minimal—though valuable—enablers, guides the conscious and unconscious choices of most software practitioners. The modern "Software BackPlanes" like CORBA and its services and J2EE are major enablers of development. Where the requirements call for a multi-user server, one or both technologies are obvious considerations. Small-scale components, such as collection classes, are part of detailed design choices. These are usually reflexive choices on the part of the developer. Who would develop their own List/Set/Queue, when these classes are readily available? But these are examples—large and small—of *horizontal components*; horizontal components are assets reusable across application domains or industry sectors. Horizontal components fill an obvious need in rapid application assembly and have been some of the most successfully reused components. However, *vertical components* serve an equally important need in rapid development within a specific domain, such as manufacturing or finance. Traditionally, they have been less successful, possibly because of inadequate domain analysis or immaturity of the domain [10].

This paper discusses our experiences in rapid application assembly with a large-scale horizontal component (J2EE) and a large-scale vertical component, the eBusiness framework, part of the Commerce Server from BEA [3] In the following section we describe the motivation for the application and the challenges faced, particularly our need for rapid decision-making when selecting architectural components. In Section 3 we describe the software and hardware architecture chosen; and in Section 4 we identify the matches and mismatches in utilizing a vertical component for a domain new to computer support. Section 5 is a reflection on what we and other practitioners should consider when selecting large components for reuse.

2 Motivation and Challenges

The application that is the basis of the experiences in this paper is *eSource*. eSource is an innovative business-to-business (B2B) application that allows members of the American Warehouse and Manufacturers Association (AWMA) to trade with one another electronically. The application provides the ability for trading members to log on to the web site, view the on-line catalogue of their trading partners, select items from the catalogue, and then generate purchase orders for those items. The purchase orders are transmitted to the trading partners in one of the specified formats: electronic data interchange (EDI)[2], e-mail, or FAX. The major benefits of the application to the AWMA Community are:

1. Members who previously have little-to-no automation support for purchase order creation can easily create purchase orders for their trading partners.
2. The application enables trading partners that are not EDI-enabled to trade with EDI-only members; this is a tremendous benefit to the members who do not want to make an extremely complex and costly investment into EDI technology.

The eSource application was created for a group of entrepreneurs who have formed a company around the technology. They are referred to in this paper as "the Business Analysts." The Business Analysts were very familiar with the AWMA members and with the trading domain.

Hedonic Considerations. Prior to the development of eSource, another company had created a prototype system for the AWMA Community. The Business Analysts indicated that the prototype had been expensive to build and had taken too long to create. Moreover, the prototype had failed to impress. Complaints were registered about the prototype's limited functionality—an inherent aspect of a prototype—but the Business Analysts felt that one of the major concerns with the prototype was the user interface. This was a simple interface that was free of adornment, merely emphasizing the functional aspects of the system. The implication of the Community's complaints was that they felt the interface was pedestrian. It was unsuitable as a vehicle to carry the Community into the 21st century.

While focus on the attractiveness of the interface may seem superficial, there are valid reasons to consider its importance. For many individuals, shopping is a hedonic experience—a source of pleasure that is separate from the goal of purchasing products [16]. Research with traditional shopping environments indicates that environments that are made more enjoyable and exciting may inject positive affect into the product evaluation and decision-making process [16] and as a result, reflect more positively on the environment that supports the process. The implications for eSource were clear: "the eSource experience" must be not only functional but fun.

[2] EDI (Electronic Data Interchange) is a standard format for exchanging business data. The standard is American National Standards Institute X12 and the Data Interchange Standards Association developed it.

The elaborate sites on the World Wide Web had set a very high acceptability an level for eSource.

Model and Metaphor Challenges. The diversity of the AWMA community presented another challenge to the system design. Many members of the community have deep computer knowledge, so the on-line shopping experience would fit their present knowledge. Many other members, however, have very little computer experience and limited access to the Internet. For these individuals great care was required in the choice of the model for interaction. Not only must the interface exploit a powerful metaphor[3], so as to be intuitive to the computer neophyte, but the task model [5], as implemented in the application must be responsive over slow dial-up lines and resilient to unexpected communication drops.

The Speed Challenge. The Business Analysts had an opportunity to develop an application that would be adopted by the AWMA Community, but the window of opportunity for funding was very limited. The failure of the previous prototype had made the Community wary of follow-on efforts. It was necessary to quickly create and make available an application the Community could try and consider. Thus, with funding in jeopardy, the clients anxious, and the use cases and requirements incomplete, we dashed forward.

3 The Application Architecture

One of many eCommerce platforms under evaluation in the development lab was BEA's WebLogic [2] . BEA had recently added two components to the EJB 1.1 compliant WebLogic server: the Commerce Server [3] and the Personalization Server [4]. These were added to allow WebLogic to compete in the eCommerce market. A demo application, "Buy Beans", was packaged with the evaluation software. The demo highlighted the capabilities of the components; it also was visually dramatic. The Business Analysts, upon seeing the Buy Beans demo, felt it was a very close match to the functionality they needed and the look-and-feel was a major selling point. The final eSource interface is based on the Buy Beans demo.

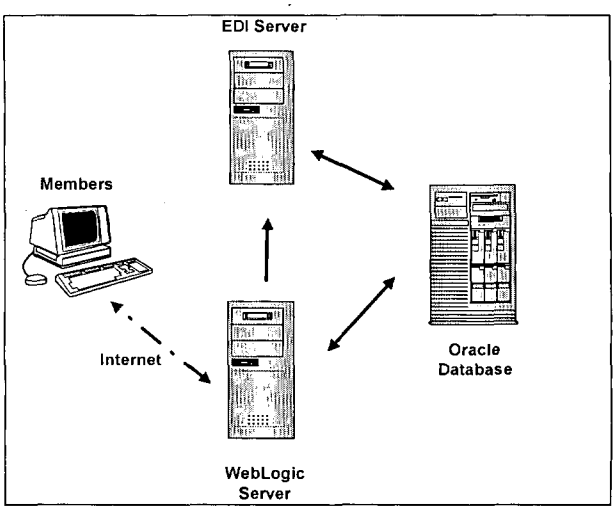
We chose to use the WebLogic server because of previous familiarity, its place as market leader for Java application servers, as well as the fact that it runs on Windows NT (the initial target platform) and Sun Solaris (a possible follow-on platform if more powerful servers are needed). The WebLogic server turned out to be an excellent choice. However, the Personalization Server and the Commerce Server were unknowns that presented major developmental challenges.

[3] An appropriate metaphor is an unconscious consideration in most modern interface design; it is worth noting here because, in developing for neophyte users within a new domain, initial assumptions may need re-examination.

The Personalization Server. The Personalization Server supports customization of Web content for the individual user through Java Server Pages and specialized EJBs. The Personalization Server allows the developer to manage users, tailor Web content, define and manage rules for the content, and create and control "portlets". A portlet is a specialized content area that occupies a small 'window' in the *portal* page. The portlet can contain content quite independent of anything else on the page.

The Commerce Server. Much of our expectations for leveraging the capabilities of the WebLogic eCommerce components rested with the Commerce Server. The Commerce Server is a framework of Java classes with considerable automation support for generating eCommerce applications. The name of the framework is "eBusiness". We discuss in detail our efforts in utilizing this framework within eSource in Section 4.

The Platform Topology. The application architecture and platform topology is shown in *Figure 1*. It consists of BEA's WebLogic Server, running WebLogic 5.1 [2] that provides Web support to Members (users). The additional components that enhance the functionality of the WebLogic server for eBusiness and run on the same machine are the Commerce Server 2.0 and the Personalization Server 2.0[4]. The WebLogic server sends data on purchase orders to the EDI Server. Both the WebLogic server and the EDI Server store data onto an Oracle 8i database, running on a separate machine. Each of the machines is running Windows NT 4.0, configured with 2 gigabytes of memory.



. *Figure 1: The eSource Platform Topology*

[4] Current versions of the BEA eCommerce components are at revision 3.5. They have been extended considerably from the 2.0 version.

4 Match and Mismatch in the eBusiness Domain

Electronic commerce—the interactive merchandising of consumer goods—is a relatively new area for computer support. Unlike other domains that have long profited from electronic technology, such as accounts receivable systems and billing systems, the domain of eBusiness is very recent. As a noted in the **Model and Metaphor** section, the questions we faced in evaluating the Commerce Server eBusiness framework focused on the following:

- **Is the metaphor for interaction effective?** Will the diverse AWMA community find the interface consistent with their intuitions regarding ordering business items?
- **Is the interaction style, defined by this metaphor, well supported in the design and implementation model?** To be perceived as useful, the system must demonstrate adequate response time and up-time [14]: implying that th implementation must support adequate throughput and scale.

Does the Framework Match the Domain: Is B2B ≡ B2C?

In one sub-domain of eBusiness, the Business-to-Consumer (B2C) domain, an effective metaphor has been the individual person shopping in a store. Everyone has had the experience of going to a store, examining goods, and purchasing those goods. Adopting this metaphor for the B2C experience has been very successful. Those Web sites that created a sense in the consumer of familiarity with the scenario, while supporting the essential functionality needed for eCommerce—have prospered; those that did not have failed (or changed). These Darwinian experiments in B2C metaphors have resulted in a quick evolution in creating effective models for these kinds of system.

Another sub-domain of eBusiness, the Business-to-Business (B2B) domain, is less well explored. One might guess that it is likely to have a great deal of commonality with the B2C domain because they are essentially about purchasing merchandise, but the immediate selection of a metaphor that captures the essence of the experience is likely to elude. Far fewer people participate in merchandise selection between business entities. There is also a sense that, with all the possibilities for relationships between businesses, there might be few universals. Except in specific cases where interactions are regulated or defined or conventionally performed in a given way, it might be difficult to articulate what is an effective support mechanism for B2B eCommerce.

In the eSource application, the Business Analysts who were driving the effort had narrowly defined the B2B interaction. The B2C experience of a single Consumer shopping in a store would be very similar to the B2B experience in eSource, except for a few differences. These differences seemed small:

- The Consumer did not buy goods from a store, but created purchase orders to be sent to the "store" for fulfilment;
- The Consumer was not a single individual, but an agent for a Company;
- The Consumer did not shop in one store, but in a mega-store, where he/she can choose products from multiple vendors all at once.

With more familiarity with the sub-domains, we might have seen that these "small" differences had big implications for the supporting model. Unfortunately, like everyone else in unexplored terrain, we had no such awareness, proceeding as we were without a "map." Our first finding was thus:

- **The Framework Employed Must Be a *Real* Match for the Domain**

The differences between the B2C and the B2B interactions are synopsised in Table 1 and Table 2. We characterize the mismatches between the framework needed and the framework available in subsequent parts of this section.

Table 1: A B2C Scenario

1. The B2C scenario begins with a Customer visiting a Web site for a virtual store where he/she is seeking merchandise to buy
2. The Customer examines available products on the site.
3. The Customer may select one or more products for purchase by placing them in a virtual Shopping Cart
4. The Customer can make modifications to these purchasing decisions by manipulating the contents of the Shopping Cart: deleting items, changing item quantities or properties; the Customer may decide at any point to abandon the shopping activity and leave the site.
5. If the Customer chooses to buy the products in the Shopping Cart, he/she proceeds to Check Out. At this point, Customer identification, credit information, and shipping information is collected. Most systems allow the consumer to save this information in a profile to be used later for the Customer's convenience and as data in developing marketing strategy for the virtual store.
6. When the purchase activity is complete, the customer receives a confirmation code whereby he/she can track the status of the order.

Table 2: A B2B Scenario

1. The Customer in a B2B is a Company, rather than an individual, as in the B2C. The Company may have one or more agents or Members who are authorized to trade (buy) for the Company. The

Member visits the Web site for the virtual store where he/she is seeking merchandise.

2. Unlike the B2C example where the Customer can begin shopping immediately by examining products, *the Member must present authorization to the systems that he/she is a valid Member of the trading alliance.* Unlike the B2C where all Customers are allowed to shop in the virtual store, in a B2B, only validated Members whose Companies have established trading agreements are allowed to enter the store. Another difference between a B2B and a B2C is that, in most cases, the relationship between Customer and virtual store is a many-to-one relationship: many Customers and one store. In a B2B, the relationship between Companies is a many-to-many relationship: many Companies can trade with many Companies. This is a more complex model to support electronically than is the many-to-one relationship and has development implications, which we will see later.

3. Steps 3 and 4 in the B2C activities covering selection and purchase of merchandise are very similar for B2B Members, but again there are differences that impact the domain model and the system implementation: *not all Members of a Company have equal capabilities*:

- *One or more Members may have administrative or managerial rights for the Company.* This may include extra privileges to allow them to audit buying activity of other Members, authorize or revoke permissions of a Member, including revoking their authorization.
- Members may be restricted to trading with only a subset of the trading partners of their Company.

In short, all the complexities for hierarchical authority that are unnecessary in a B2C may be part of a B2B.

4. *Check out for a B2B Member does not require identification or profiling* because this information has been collected when the Member was authorized to use the system. Customisation of the order may occur at this point, however, including indications of where to ship the goods, specifying a shipping cut-off date, etc.

5. When the purchase is complete, the Member receives one or more confirmation notices whereby he/she can track the order. Multiple confirmation codes may be generated if goods are requested from multiple partner Companies.

The eSource Domain Framework. A simple representation of the final framework for eSource is shown in

Figure 2. While `Company` is a primary element in a B2B framework, as noted in

Figure 2 most of the relationships are associations between the company `Member` and other framework elements.

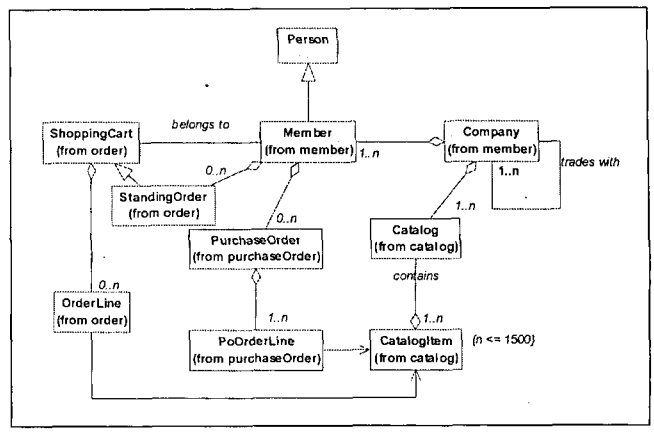


Figure 2: The Domain Framework for the eSource B2B System

A Member, who is a kind of Person, is an associate of a Company. A Company trades with one of many companies. Each Company has one or more on-line Catalogs. (Only one Catalog was allowed in the first release.) Each Catalog contains 1 to many Catalog Items, typically the number of items didn't exceed 1500. The Member can select items for purchase from Catalogs belonging to trading partners of his/her Company. Selected Items are placed into a Shopping Cart. The Shopping Cart then contains Order Lines that include Item information, Quantity ordered, and the Price for that Quantity. The Shopping Cart can be saved as a Standing Order, to be activated at another time when the Member wishes to purchase the same items. (The content of a Standing Order is then added to the Shopping Cart; one or more Standing Orders can be added to the Shopping Cart.) Upon checkout, Purchase Orders are created from the contents of the Shopping Cart. A Purchase Order contains PoOrderLines with an immutable Price for the Items and Quantities. (Purchase Orders are saved into the database and must never be change once created.) A separate Purchase Order is created for each Company's items in the Shopping Cart. All Purchase Orders generated remain the property of the Member and may be viewed on-line.

The eBusiness Framework from BEA's Commerce Server. *Figure 3* illustrates a portion of BEA's Commerce Server framework that was relevant to the eSource project. Showing this limited view of the framework probably is an injustice; it has a far richer structure than is illustrated. In addition to classes shown here, the Commerce Server provides packages that support a gift registry, inventory, invoicing, shipping, tax, trouble tickets, and a shopping advisor that matches customers to items.

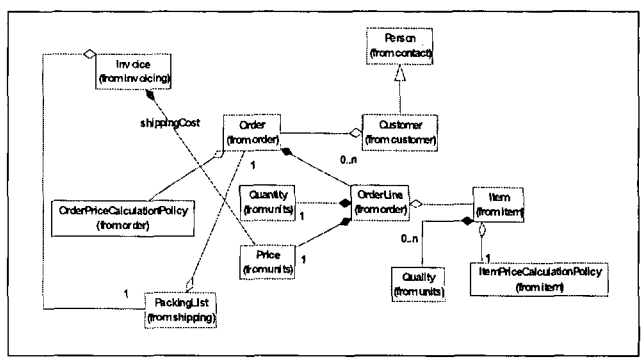


Figure 3: Some of the Classes from BEA's Commerce Server

The eBusiness Framework is a very rich structure for creating eCommerce applications. It is well modelled and makes few, if any, compromises with its "object-oriented-ness"; prices and quantities for goods are modelled as objects, as are policies for pricing. This approach promotes flexibility in areas where flexibility is important. Unfortunately, it may not promote optimal performance, particularly where such flexibility isn't needed. The next section discusses such issues in detail.

Differences in the Frameworks and Implications

This section covers the differences between the final framework for eSource and that of BEA's eBusiness. Design of the eSource framework was not even begun when we had to choose the architectural components of the system. The subsequent divergence of the two models was a result both of growing awareness of differences between B2B and B2C domains and the inevitable churn of requirements changes. These changes were particularly virulent with eSource, perhaps because of the number of stakeholders involved. Table 3 is a synopsis of the similarities and difference in the models.

Table 3: Differences in the Frameworks

eSource Class	eBusiness Class
`Person`	`Person`
`Company`	*No equivalent*
`Member`	`Customer`
`ShoppingCart`	`Order`
`StandingOrder`	*No equivalent*
`Catalog`	*No equivalent*
`CatalogItem`	`Item`
No equivalent	`Quality`
Default price or promotional price	`ItemPriceCalculationPolicy`
`OrderLine`	*Similar to eBusiness `OrderLine` but not identical*
Invariant attribute of CatalogItem	`Price`
`PurchaseOrder`	*Similar to `Order` and `PackingList`*
Hard-coded	*`OrderLineCalculationPolicy`*
`PoOrderLine`	*Similar to PackingList, but not equivalent*
Attribute of PoOrderLine	`Quantity`
Some similarities to PurchaseOrder	`PackingList`
Handled externally to eSource through billing system; information extracted through company relations	`Invoice`

Person. Fortunately, a `Person` is just a `Person` in both systems, with first and last name, etc. Many of the descriptive attributes of a `Person` were not necessary for eSource.

Company. Company information includes Company name, tax id or DUNS number, billing address, and default shipping address, trading partner relationships, among other information.

Member versus Customer. While a `Customer` is similar to a `Member`, a `Customer` is an individual `Person` in the eBusiness application but a `Member` is an agent for a `Company` in eSource. Company affiliation is extremely important in eSource because this is how an agent is authorized. Moreover, some Company Members have special privileges allowing them to change the Company profile or administer the rights of other Company Members.

ShoppingCart versus Order *(and secondary classes)* A shopping cart in most B2C applications is simply a means for holding items selected for purchase by the `Customer`. While also is true in eSource, there are crucial differences:

1. The shopping cart must hold items from multiple catalogues. This facility is needed because the `Member` might examine catalogues from multiple companies and buy from several.
2. The shopping cart could be loaded with items from Standing Orders, see below.

- **OrderPriceCalculationPolicy.** A class used to compute the total cost of the Order, including costs for shipping and tax. Not used in eSource because shipping was not relevant and tax was left out of the first release.

StandingOrder. A `StandingOrder` is a collection of goods routinely ordered by a `Member`. Standing orders might be goods ordered weekly or monthly, or they might be goods ordered only during a particular season, such as summer merchandise. Because a `StandingOrder` may reference `CatalogItems` whose availability and prices vary over time, operations on a `StandingOrder` differ from the simple add/modify/delete operations `Members` perform on a `ShoppingCart`. When adding the items in a `StandingOrder` to a `ShoppingCart`, the availability and current price must be checked to be sure that the addition is valid.

Catalog. A `Catalog` is an aggregation of `CatalogItems`. A single catalog per company is allowed for the first release. Subsequent releases may support multiple catalogs to allow for tier pricing.

CatalogItem versus Item *(and secondary classes)* A `Catalog` contains a `Vector` of `CatalogItems` are objects in eSource, but an `Item` is an Entity Bean in the eBusiness framework. (See **'Does the Persistence Model Fit the Interaction Paradigm?'** below.)

- **"Hard coded" Price Calculation versus ItemPriceCalculationPolicy.** In the eBusiness

framework the `ItemPriceCalculationPolicy` is an aggregate of policies that can be associated with an `Item` to determine its price. The `ItemPriceCalculationPolicy` implements the *Strategy Pattern* [15] allowing specialized pricing policies to be associated with an `Item` This supports fine-turning of pricing for different Items, quantities, occasions—a wide variety of situations. Using the *Chain of Responsibility Pattern* [15], "pluggable" policies can be selected appropriately for the circumstances.

We had expected considerable benefit from the `ItemPriceCalculation` class because pricing is a very complex aspect of eBusiness, particularly in B2B relations. Prices vary according to quantity ordered, promotions, and established relationships between businesses, among many other options. Allowing this level of flexibility to pricing and being able to encapsulate these policies in a pricing policy class seemed to be a good means of capturing the information. Unfortunately, in going to the AWMA members, we found that there was little consistency amongst their pricing policies; they varied widely. It became clear that encapsulating their policies would be impossible. Even getting them to articulate them for encapsulation would be an arduous process, prohibitively time consuming. We abandoned it in favor of a single price within a `Catalog`, with an optional promotional price valid for a specified range of dates. Price changes could only occur with Catalog updates. While this simplified approach did not fit the pricing strategies for any but the smallest of AWMA member companies, all agreed that they would adopt this approach in order to get started in using the system.

- **Quality.** Used to match `Customers` to `Items`. Not applicable to eSource.

Order versus PurchaseOrder . An `Order` (equivalent to a `ShoppingCart` in eSource) is not a `PurchaseOrder`. Multiple `PurchaseOrders` may be created from the `ShoppingCart` on checkout.

eSource OrderLine versus eBusiness OrderLine *(and secondary classes)* The aggregation of all ordered items in the `ShoppingCart`, the two `OrderLines` differ primarily in the representation of Price and Quantity.

- **Price**. Price is a single value in the `Catalog` within eSource, except where there is a promotional price for specified effective dates. Within the eBusiness model, `Price` specifies both a value and a currency, allowing for conversion between currencies. Although the eSource application is intended to become internationally available, requiring the facility for currency conversion, that capability was not foreseen in any anticipated release.
- **Quantity**. Like `Price`, `Quantity` stores a value and a unit, supporting unit conversion between differing measures. Unit conversion was considered too idiosyncratic for goods within the domain to be useful to eSource.

"Goodness of Fit" of the Persistence Model to the Problem. The persistence model for an application—how and what information in the application is stored—follows from the domain model. That is, there is a high correspondence between the domain model and the logical format of the data in the database.[5] Clearly, there should be a good "fit" of the problem to the persistence model. In hindsight, these are some issues we found we must consider in determining "goodness of fit":

1. Does the persistence model fit the interaction paradigm?
2. Are there implicit scale limitations in the persistence model that are in conflict with requirements?
3. Does the persistence model support sufficient "separation of concerns" between the logic tier and the data tier?
4. Does the framework support some degree of automation to ease the more rote coding tasks?

Does the Persistence Model Fit the Interaction Paradigm? In selecting a persistence model for an application, the interaction paradigm of the user must be carefully considered. Some of the many questions designers ask before solidifying a model include:

- What data does the user access frequently?
- What activities are most data-intensive?
- What data is modifiable and how frequently is it updated?
- What data is read-only?

All of these questions drive persistence modelling decisions:

- Data frequently accessed is made readily available; possibly this data is cached in the server to speed acquisition. This approach is modelled on techniques used in OODBMS [8,9]
- Data-intensive activities must be treated carefully because a poorly implemented persistence model could slow performance for all users.
- Data that can be modified by the user must be protected within transaction boundaries; composite objects must be ensured data integrity through atomic commit [9]. Performance issues also complicate modifiable data because the designer must balance loss of user edits, still uncommitted and residing in server memory, against too frequent transactions that might impact performance.[6]

[5] Where this isn't the case, there are likely to be performance problems. These may be unavoidable when grafting a new application onto a legacy database.

[6] As noted in the **Model and Metaphor Challenges** section, many members of the AWMA user community dialled in over communication lines that might

- Read-only, invariant data is the easiest to handle because complications involving the integrity of concurrently accessed data aren't an issue [13] The structuring of this data can often be fairly lightweight because the mechanism for modification and update need not be part of the persistence model.

When a framework incorporates a persistence mechanism into its capabilities, the designer must carefully assess the assumptions built into that mechanism against the reality of the user interaction model.

In answering the design questions about user interaction in eSource, we found the following: the user's `ShoppingCart`, `StandingOrders`, `Catalogs` of trading partners, and `CatalogItems` are the most frequently accessed data. Less frequently accessed information includes previous `PurchaseOrders`, limited to 250 that are "viewable" on-line, and `Company` and `Member` profile information. The most data-intensive activity engaged in by the user is search of the `Catalogs`. Although the `Catalogs` are limited initially to no more than 1500 items, a `Member` may search many `Catalogs` if his/her `Company` has many trading partners. A naïve search could result in the request for all items in all `Catalogs`—a possibly very data-intensive operation. Fortunately, the `CatalogItems` are not modifiable; the data in them is read-only to the searching `Member`. (eSource personnel do `Catalog` update through a separate process as an administrative function.)

The eBusiness framework supports a persistence model that turned out to be a poor match for this interaction paradigm. Much of the mechanism for persistence was auto-generated from a Rational Rose model [23] through the Smart Generator. (This is a valuable capability, discussed later under **'Is There Automation Support?'**) There also was a nice separation between the domain class, such as `Member`, and the Data Access Object or "DAO" [19] class that encapsulated the JDBC code for persistence management. There were some nice optimisations, similar to those described in [17,18] for handling EJB Entity Bean references to contained elements. These minimized memory usage through judicious loading of primary keys. The framework also supported key comparators to check equality relations between objects through their primary keys. However, the framework required that a persistent object be an EJB Entity Bean. This caused us initial concern about performance and these concerns were borne out. Prior to committing to a detailed design, we gathered performance metrics [7] using Entity Beans on our target hardware. We found it prohibitively slow. Moreover, because there is very little modifiable data within eSource, the full capabilities of Entity Beans were not needed. We found we could get very good performance and all the management capability we needed by modelling the `Catalog` using Bean Managed persistence and the `CatalogItems` as a collection of simple objects referenced by the `Catalog`.

Does the Persistence Model Fit the Requirements?. As discussed in Section 3, under The Platform Topology, the eSource application uses an Oracle database as a repository for both EDI information and company information (Company, Member, Catalogs, etc). One of the implicit requirements for the repository was that is transparent to SQL queries; by that we meant that access to and modifications of the data store could be performed, if necessary, completely through SQL commands. It was important that there be a separation of concerns between each of the system tiers (interface layer, logic layer, data layer); that is, each tier was independent of any other. Not only did we adhere to this as an architectural principle to promote plug-compatibility amongst layers, but also we needed to be able to extract data for the eSource billing system, independent of the logic layer[7]. The billing system required access to partner data so as to bill eSource customers for using the trading service.

Unfortunately, a late discovery was that the automation support provided by the Commerce Server stored aggregate elements as LONG RAW values in the database. The price of an item, for example, was stored as a serialized object in a LONG RAW format. This approach is commendable in that it minimizes the number of "little objects" that must be tracked when a primary, domain object is made persistent, while allowing for the object to be materialized when needed *as an object*. However, this approach conflicted with our requirement.

Given all the conflicts between our performance requirements and our data transparency requirements, we jettisoned the modelling approaches that supported automation and wrote our own DAO classes.

Are There Implicit Scale Assumptions? Some modelling decisions may inadvertently build in an assumption of limited scale. In considering the elements supported in the eBusiness model and the way in which they are made persistent, one gets the impression that the framework is targeted at an application with rather limited scale. If one considers the majority of shopping systems on the Web (with the exception of mega-portals like Amazon.com), a limited scale is perfectly appropriate. Most shopping systems don't carry hundreds of thousands of items. Probably, most shoppers don't want to see that many. A few hundred or a thousand items probably are sufficient. Our early performance metrics gathering suggests that this perception of limited scale is probably correct—at least for our target

unexpectedly drop; loss of user edits was a primary concern in application design.

[7] This was both a "separation of concern" issue, as well as a performance issue.

architecture. Anecdotal evidence from another eBusiness developer indicated that on a large Sun server, performance was quite acceptable.

Is There Automation Support? The eCommerce framework automates many of the tasks of creating a B2C application. These automation procedures have been mentioned in passing, but because they are such a good example of support, we highlight them here:

- A Rational Rose model is supplied that models all the framework elements and documents the elements and relationships;
- Stereotypes on the elements and relationships specify how the elements will be made persistent;
- Export of the model in the proper form for the SmartGenerator is supported as a Rose add-in tool;
- The SmartGenerator uses the exported model to generate the supporting interfaces and classes necessary to create Entity Beans; behavior for the Impl classes must still be hand coded.
- The DAO classes with the requisite JDBC code are generated automatically.
- The database code can be automatically generated (if using Bean Managed Persistence rather than Container Managed Persistence) from a mapping file of database properties.

These facilities stand as an example to be emulated by other framework providers.

5 Conclusions

Our experiences in using the J2EE horizontal component of the application were more successful than using the vertical component, the eBusiness framework. The reasons for this disparity seem to be:

1. A greater familiarity with the J2EE standard and the BEA WebLogic implementation;
2. A poor match of the B2B domain with eBusiness B2C framework.
3. A lack of familiarity with the rich and deep technologies supporting the eCommerce mechanisms.

Our difficulties with employing the vertical framework have lead to the following observations:

- **The Framework Employed Must Be a Very Close Match for the Domain**

A vertical component is a very specific perspective on a domain. Unless there is a near perfect match between your application's perspective and the framework, specialization of the framework probably won't win you much. It would be best to choose these large-scale vertical components after finishing detailed design. At that point you can determine how close the match between your design and the framework is and whether you can modify your design to get more utility from the framework—or live with the mismatches. Where this timing of choice isn't possible, prepare to factor into your schedule enough time to create your own framework if you must jettison the vertical component.

- **The Persistence Model Must Fit the Interaction Paradigm**

The designer must be aware of which data is most frequently accessed, which is modifiable or read-only, and which activities are most data-intensive in order to correctly choose a persistence model.

- **Implicit Scale Assumptions Must Be Carefully Considered**

Design assumptions within the framework may assume a small user base or may assume deployment on a *very* large server. Knowing the assumptions beforehand can minimize dashed expectations.

- **Automation of Rote Coding Tasks Is Very Helpful**

Much of the effort of creating the code to support EJB definition, domain object vs. DAO object separation and JDBC coding can be automated—and should be.

6 References

1. Paseman, W. (1989) "The Atherton Software BackPlane—Architecture for Tool Integration," Unix Review, April.

2. WebLogic Server 5.1, documentation available from BEA web site: http://www.beasys.com.

3. BEA Systems, Inc. (2000) BEA WebLogic Commerce Server Components Developer's Guide, BEA WebLogic Commerce Server 2 .0, Document Edition, 2.0.

4. BEA WebLogic Commerce Server Components Developer's Guide, BEA WebLogic Commerce Server 2 .0, Document Edition, 2.0.1.

5. Benbasat, I. and P. Todd (1993) "An Experimental Investigation of Interface Design Alternatives: Icons vs. Text and Direct Manipulation vs. Menus", *International Journal of Man-Machine Studies* 38:369—402.

6. Biggerstaff, T. (1994) "The library scaling problem and the limits of concrete component reuse," In *3rd International Conference on Software Reuse*, pages 102-109, Rio de Janeiro, Brazil, November 1994.

7. Carando, P. (2001) "Performance Assurance: An Architectural Approach to Meeting Server-Side Java Performance Requirements", *Java Report 6(4):36—43*, April.

8. Cary, M., DeWitt, D., and Naughton, J. The OO7 Benchmark, Computer Sciences Department, University of Wisconsin-Madison.

9. Cattell, R. G. G. (1991) Object Data Management: Object-Oriented and Extended Relational Database Systems, Addison-Wesley, Reading, MA.

10. Carnegie Mellon Software Engineering Institute (2000) Domain Engineering and Domain Analysis, Software Technology Review document.

11. Cox, B. (1986) Object-Oriented Programming, An Evolutionary Approach, Addison-Wesley.

12. Cox, B. (1990) Planning the Software Industrial Revolution, November 1990, IEEE Software Magazine.

13. Date, C. J. (1995) An Introduction to Database Systems, 6th Edition, Addison-Wesley, Reading, MA.

14. Davis, F. D. (1993) "User Acceptance of Information Technology: System Characteristics, User Perceptions

and Zbehavioral Impacts." *International Journal of Man-Machine Studies* 38:475-487.
15. Gamma, E., R. Helm R. Johnson, and J. Vlissades (1995) Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley.
16. Holbrook, M. B., and E. C. Hirschman (1982) "The Experiential Aspects of Consumption: Consumer Fantasies, Feelings, and Fun." Journal of Consumer Research 9(September):132-140.
17. Holland, G. (2001). "Entity Bean Relationships in EJB 1.1, Part 1: The Basic Technique", *Java Report, 6(4):*72.
18. Holland, G. (2001). "Entity Bean Relationships in EJB 1.1, Part 2: Robust and Practical Techniques", *Java Report, 6(6):58—61.*
19. Kassem, N. and Enterprise Team (2000). Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition, Addison-Wesley, and at http://java.sun.com/j2ee/blueprints.
20. The Object Management Group (2000). The Common Object Request Broker: Architecture and Specification, CORBA 2.4.2, available through the Object Management Group web site: http://www.omg.org.
21. The Object Management Group (2000) Individual service specifications available at the Object Management Group web site: http://www.omg.org.
22. Long, E. and F. Morris (1993) An Overview of PCTE: A Basis for a Portable Common Tool Environment, Ed Long and Fred Morris, Technical Report CMU/SEI-93-TR-1, ESC-TR-93-175.
23. Rational Software Corporation (2001) Rational Rose: Model Driven Development with UML, data sheet, available through the Rational web site: http://www.rational.com.
24. Shannon, B., M. Hapner, V. Matena, J. Davidson, E. Pelegri-Llopart, L. Cable, et al. (2000) Java™ 2 Platform, Enterprise Edition: Platform and Component Specifications, Addison-Wesley.
25. Sun Microsystems Inc. (1999) JDBC™ 2.1 API, final specification, available at: http://java.sun.com/products/jdbc.

Management process for supporting the component development

Haeng-Kon Kim[1]
Dept. of Computer Engineering, Catholic University of Daegu ,KyungSan, TaeGu, 712-702, Korea.
hangkon@cuth.cataegu.ac.kr

Roger Y. Lee
Dept. of Computer Science, Central Michigan University, Mt.Pleasant, MI 48859, U.S.A.
lee@cps.cmich.edu

Keywords: Component-Based Development, management process, independent extensions, cost effective, component repository

Received: May 28, 2001

CBD (Component-Based Software Development) has rapidly become a substantial and interesting field in the development of business applications. Because CBD represents a new development paradigm composing applications from software components, increasing requirements for productivity of the flexible systems development can be solved by CBD technologies. It must be recognized that CBD does not mean acquiring parts from anywhere. CBD is a new discipline and there are many associated problems which remain unsolved

In this paper, we try to clarify the CBD-related theories as practical techniques to be applied to real systems. So, we suggested technical theories for guidelines management for supporting the development of the CBD process, especially for component development. We focus on setting standards for components and address the impact that CBD has on managing component development. This includes the model management strategy, development and delivery of components, adoption by an organization and the capability to add new releases of components or parts of components.

1 Introduction

We have pressure for bringing new products to the market, but we don't think the software development life cycle is becoming shorter. To be able to provide required functionality to the customer, the use of standard components and components developed by a third party supplier are becoming more and more important.

CBD(Component-Based Development) as a vision and an approach offers many exciting possibilities in terms of reducing application development costs, providing greater software reuse, and facilitating maintenance and evolution of systems[1,2,3]. However, to achieve this vision in practice requires a number of hurdles to be overcome. How does one pull off this feat of architecture? Where do the components come from? What must you do to ensure you get the advertised benefits? If you are in the hot seat labeled 'architect', what should you do?

The following appear to be the most significant factors of CBD:

Reduced time-to-market. The availability of components of the sort just described also promises to drastically reduce the time it takes to design, develop and field systems. Design time is drastically reduced because key architectural decisions have been made and are embodied in the component model and framework. Component families such as those found in the Theory Center obviously contribute to reduced time to market. Even if such component families are not available in an application domain the uniform component abstractions will reduce development and maintenance costs overall.

Component markets. Component models prescribe the necessary standards to ensure that independently developed components can be deployed into a common environment, and will not experience unanticipated interactions such as resource contention. The integration of support services in a framework also simplifies the construction of components, and provides a platform upon which families of components can be designed for particular application niches.

Independent extensions. One problem that plagues legacy software is lack of flexibility. Components are units of extension, and a component model prescribes exactly how extensions are made. In some cases the framework itself may constitute the running application into which extensions (components) are deployed. The component model and framework ensure that extensions do not have unexpected interactions, thus extensions (components) may be independently developed and deployed[4,5,6].

Improved predictability. Component models and frameworks can be designed to support those quality attributes that are most important in particular application

[1] This research was supported by the Research Grants of Catholic University of DaeGu in 2001.

areas. Component models express design rules that are uniformly enforced over all components deployed in a component-based system. This uniformity means that various global properties can be "designed into" the component model so that properties such as scalability, security and so forth can be predicted for the system as a whole. For example, EJB ™ [7] is touted as promising scalable, secure, and distributed transactions by virtue of its component model and framework services. It might be argued that there are other benefits that accrue from a component-based approach to systems.

It must be recognized that CBD does not mean acquiring parts from anywhere: they are unlikely to be compatible without a great deal of integration. Instead, components must be designed to work together, and designed for a particular domain or business: interoperability standards for components must be agreed upon within a project, company, or industry.

These standards involve much more than interconnection and container technology such as CORBA[8], COM[9] or EJB[7]. A distinction between the tasks of the product builder, component designer, and component composer and product line architecture must be made.

In this paper, we try to clarify the CBD-related theories as practical techniques to be applied to real systems. So, we suggested technical theories for standard management in supporting the CBD process especially implementation and delivery phases. We focus on setting standards for components and addressing the impact that CBD has on managing component development. This includes the model management strategy adopted by an organization and the capability to add new releases of components or parts of components.

2 Related Studies

2.1 Component Repository

Component repository is a library system that supports finding, providing and managing components for building a business application. So it is a kind of tool to store, register and manage all of the artifacts produced in the component life cycle based on component architecture, and support a "Reuse with component" in the CBD process through performing advanced retrieval and browsing of information. Most of all, component repository is a central mediator for component generation and utilization. So, analyzing and applying consistent meta and user feedback information can establish the CBD process including creation, verification, configuration management and circulation of component[3][4].

2.2 CBD Process

CBD promises cost-effective productivity assuring a high flexibility and maintenance by assembling the components as independent business processing. The CBD environment is divided into two aspects according to process evolution level. That is, we consider the CBD process as a supply process producing and providing the commercial components into a repository, and consume process supporting component utilization for constructing business solutions[1,2,3,4,5,6,7]. The big picture represents essential works for realizing the CBD process, subjecting the basic principles for component reuse that is acquisition-understanding-applying, is shown in figure 1.

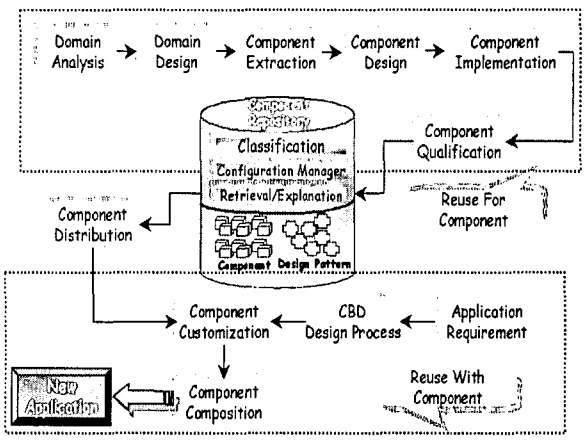


Fig. 1. CBD Process

2.3 Current States in CBD

CBD has rapidly become a substantial and interesting field in business applications, especially since CBD is primarily used as a way to assist in

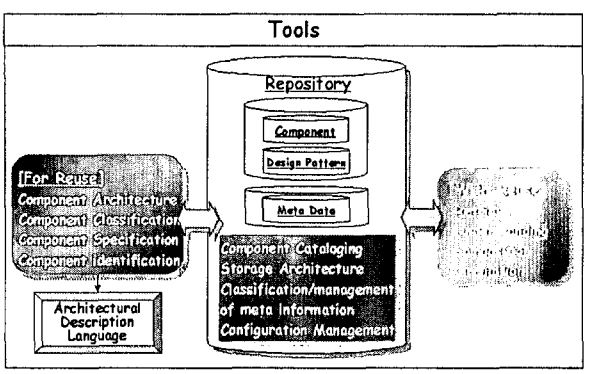


Fig. 2. Basis techniques for CBD process

controlling the complexity and risks of large-scale system development, providing an architecture-centric and reuse-centric approach at the build and deployment phases of development. So now, many vendors and researchers have tried to establish the CBD maturity by involving the following strategies [9,10]:

1) efficient building of individual components,
2) efficient building of development solutions of in a new domain effectvely,
3) efficient adapting of existing solutions to new problems and efficient evolution of sets of solutions.
But, by the lack of standardization and clearness for the CBD approach method, we can't expect a practice benefits in business solutions. So, we need the approach techniques in each step for organizing and practicing the CBD process like figure 2 [6].

3 Management Process for Components Development

CBD software systems are developed on an underlying process different from that of traditional software, so their management and quality assurance models should address both the process of components and process of the overall system. It is also essential that standards for components be in place before the components are built. Also needed are standards defining the contract between the component consumer and the component provider. Two types of standards are critical:

- Implementation
Because components can be delivered with or without the implementation, standards for the implementation are critical for component builders and for the component consumers who may modify the implementation at some other time. Implementation standards cover building the component, as well as its naming, identification, versioning, error handling, and security.
- Delivery
Delivery standards define a component's operations, specification type, and interfaces for the consumer.
Delivery standards should cover at least error handling, naming conventions, consumer information, and test plans and procedures.

3.1 Component Development Management

Component development is the process of implementing the requirements for a functional, high quality component with multiple interface. The objectives of component development are the final component products, the interface, and developments documents. Component development should lead to the final components satisfying the requirements with correct, well-defined behaviors and flexible interfaces. When developing and designing components, we recommend the following criteria:

- Provide test-suites with the component so that the customer can test your component in their environment. It is extremely important to test an imported component in the environment in which it will operate.
- Provide source code if possible, it might help the application developer to understand the semantics of your component.
- Make the components so they easily integrate into existing components. Describe what the component works with and describe how to make it work with other components as well.
- Components need to be carefully generalized to enable reuse in a variety of contexts. However, solving a general problem rather than a specific one takes more work.

Make sure that the application developers can adopt the component to their requirements. This can be done with sink interfaces where the user adds its own interface to the component so that the component can use that interface to communicate with the user. In our project, we studied managing the component development process for CBD software development paradigm as the following phases: 1) Requirement analysis; 2) CBD development; 3) Certification 4)

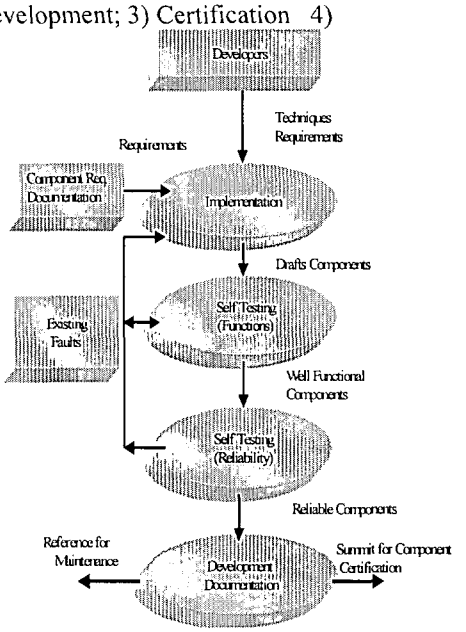


Fig. 3. Component implementation process

Customization. We suggest the component implementation process as shown in figure 3. Component implementation process consists of four sub processes: implementation, function testing, reliability testing and development document.

The input to this phase is the component requirement document. The output should be the developed component and its document, ready for the following phase of component certification and system maintenance, respectively.

3.2 Defining a Model Management Strategy

Each component is described in a component model. *Component model* is used to differentiate it from an application model. Component model is a type of support model. Component Models, on their own, may or may not provide a business solution. If they do not provide value on their own, they should do so when combined or assembled with several other components. This allows the organization to understand how to create, use, and maintain components. A model management strategy is an approach set up to organize and work within the component repository. The strategy is defined in terms of model types and their interrelationships. The model management strategy also includes procedures for creating, maintaining, retaining, synchronizing, and versioning models. The model management strategy is based on an organization's development coordination strategy. The types of models and the number of models per model type are dependent on:

- Diversity within the organization
  This is both business and technical (location or department, user organization, target production environments, security requirements).
- Complexity of development efforts
  Large efforts should be segmented into natural clusters to make the model more manageable and understandable. Implementation of business functionality should be incremented to minimize the impact to the user organization.
- Evolutionary development efforts
  Concurrent projects in various life-cycle phases are difficult to manage within a single model. Incremental and manageable releases into production should be supported.
- Technical capabilities
  This addresses the number of encyclopedias, performance of encyclopedia functions, and contention for reusable objects.

This refers to the amount of reuse through copying and sharing that the organization plans, as well as the degree of integration desired. Model management is the primary means by which an organization realizes its development coordination strategies in actual operation. An organization may define a model management strategy that has one or more of the component model types described in table 1. We suggest the model for management strategy as in figure 4.

With reuse of components, new types of models will be introduced into the model management strategy. These model types are described CBD model type with component, component catalog which contains the specification for component, and component pattern which will contain pattern to be used for developing component in the future.

Table 1. CBD Common Model Types for Management in our work

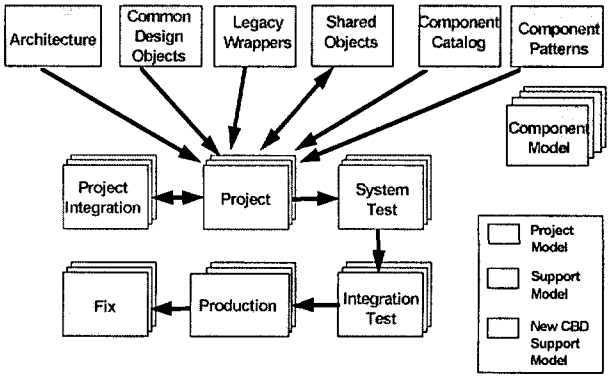


Fig. 4. Model for management strategy

The arrows between the models in figure 4 represent likely flows of information between the models. For example, the method of information flow may be copy, migrate, and create model from a subset.

Model Type	Description
Architecture	Holds a high-level model for an organization. It can be created as a result of a strategic planning or an information needs structuring activity.
Common design objects	Holds business system design standards. This model can be created or purchased and customized by an organization to propagate application design standards.
Legacy wrappers	Holds logic to enable access to commonly used legacy applications. Wrappers can be developed to reasonably insulate the using application from changes to the legacy application.
Shared objects	Holds definitions for common objects that are being used by multiple projects.
Project	Holds the documentation about the requirements and design efforts for the project.
Project integration	For large projects or programs, where there is a need to distribute development or "divide and conquer," a project integration model may exist. This model facilitates holding, synchronizing, and propagating the common objects across the sub-project models.
Test	A project may have one or more test levels, for example unit, system, integration, and acceptance. For each level of testing, there may be a separate supporting model: • System – testing the functionally testing application flows • Integration – testing application within the planned production environment • Acceptance – user acceptance test These test levels are examples of model types that can be used to progress a project from development through test.
Production	Represents the source of the last released application.
Emergency fix	Represents the fixes that have been applied to production since the last release. This may be a whole model that mirrors the Production model Rx, or it may be created when needed based on the production model. Production Rx+1 is the next release of production.

3.3 Management Process for Delivering, Publishing and Integration of Components

With our work, we suggest the management process for delivering, publishing and integrating of component. Ideally, there is a model for each components. The component model can hold the component specification, interface, and implementation design. There are three options for delivering a component:
(1)Deliver the whole component in which the complete component model with executable software modules is delivered. (2) Deliver the executable component in which the component model is delivered with specification, interface and executable software modules. (3) Deliver the component implementation design is which the component model is delivered with specification, interface and implementation design.

A software vendor who intends to sell a component is likely to protect the component implementation and use the second option. It will be important to have a central model, such as the component catalog model, to publish the component specifications. If this approach is used, it will be important to define and enforce component naming standards so there is no conflict when the specifications are migrated into this model or other models. Assuming the component consumer is an application model, the component specification is replicated in the application model, which then links to the generated executable software module.
After a component is built, its operations must be published or made public. The component catalog model

is used for this purpose. Publishing Component Specifications is the process for publishing the specifications of components. To make browsing more manageable, a model referred to as the *component catalog model* can be introduced to centrally hold the component specifications. After the component specifications are migrated into the model, they may need to be edited to ensure that the specification contains only the operation specifications. The operations action diagrams must contain only the import and export views, notes for the pre-/post-conditions, return codes, and its source member name. The component catalog model will then be the source for the component specifications.

When an application is initiated, there will be tasks to analyze or determine what is available for reuse that matches the requirements through component browser process as in figure 5. The component catalog model, if it exists, can be used to browse what is available as well as the source of the specifications. Alternatively, each of the component models can be reviewed. The required component specifications will be migrated into the application models. Applications may still reuse objects by replicating them in their models. The application integrate in figure 6 shows that a new application model has several sources to start assembling the application. If the application reuses software components, the source of the specification can be the component catalog model. When an application reuses in this manner, it is important to remember that the software executable modules have been generated by its source component model. Therefore, during test and construction, it is important that the using application be linked to the correct software module.

At this point, the components must have a specification defined and should be tested. For example, the component must have been used and tested in two or three applications. Component consumers can then obtain the required interface from the component model.

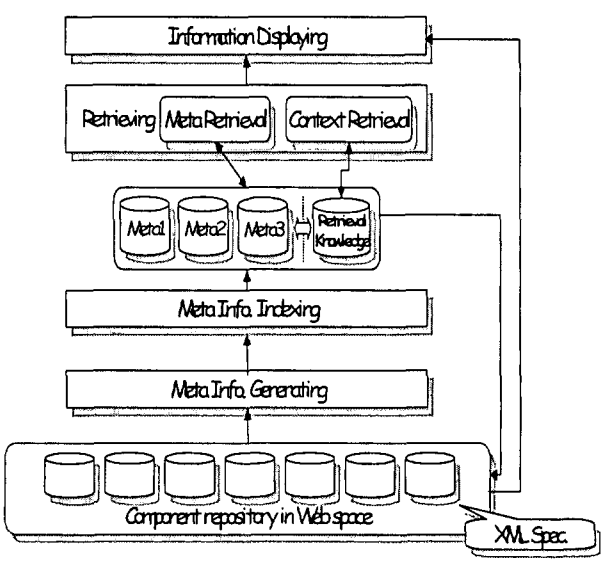


Fig. 5. Component browser process

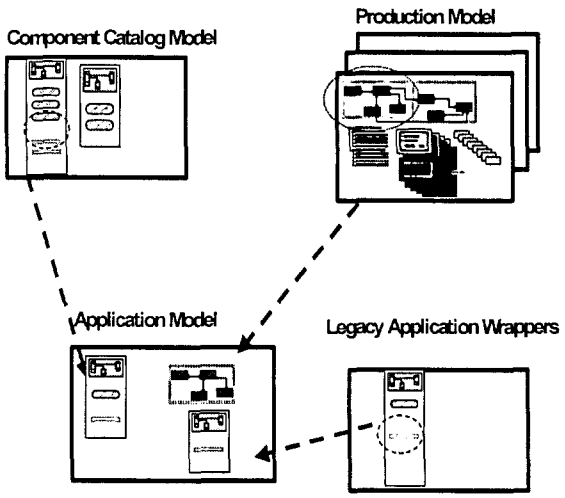


Fig. 6. Process for Application Integration of Component

Before acting and making decisions on how to build applications from components, we recommend that the following questions and thoughts be considered:

- The time your product is off the market can be greater than the time saved getting your product to market if your component supplier drops the product. Can you accept this risk?
- The functionality provided by the component may not remain precisely what you need over time, forcing you to create wrappers that get around this. Things are getting even worse if you are not getting support from the vendor.
- The functionality of the component may be more than you actually need, requiring you to write restrictive wrappers for functionality that you do not want to be used. Use of unintended functionality may cause problems.
- If you succeed in getting the source code from the component vendor, can you really maintain it if something goes wrong?
- A malfunction in the component may cause an error in your product. Are you willing to have a certification strategy for this. Your customer wants your product to work without having to think about your internal design. You have to provide a fix for the problem even though the error is in the third-party component.
- If you ask the component vendor to customize the component for you, are you aware that you now are strongly dependent on the vendor? The vendor can manufacturers are sharply reduced because their over time, there can be many component models, and it may be difficult for an application assembler to determine what is available for reuse.

In some cases, the application may choose to replicate the software executable modules in production. Whichever approach is used, it is important to ensure that the right module is linked each time the calling module is constructed or the reused software module is re-generated.

Table 2. Releasing meta-information of component

Category	Elements	
Registration	Vendor, Address, Partners, Tel-NO, Fax-NO, Mail-Address, URL, Component-Name, Classification-NO, Domain, Version-NO, Brief-Description, Assurance-NO, Registration-Date	
Management	Classification-Info(Hierarchy), Sale-Rate, Usage-Rate, Version-History, Quality-Info, Price, Priority	
Composition	Family-Classification-NO, Family-Component, Role-Relationship, Use-Case	
Functionality	Interface, Functionality-Description, Parameter, Constraint, Usage-Scenario, System-Requirement	
Environment	Platform, Middle, Database, Development-Tool, Description-Language, Reference-Model	
User	User-ID/Name, Password, User-Email, Visit-NO	
Repository	Management-Domain, Management-Component, Implement-Environment, Error-History	
Retrieval	General	Component-Name, Business-Domain, Description
	Functional	Interface, Method,
	Environment	Container, Family, Platform
	Circulation	Deliver-Status, Creation-Date, Price, Vender

The application integration process requires the following capabilities in the component repository:

- To browse available component specifications can be done by maintaining the component catalog model.
- To create an aggregate set for each component specification interface.
- To concatenate aggregate sets for component specification. This is necessary to migrate several component specifications at the same time.

We have the recommendations to the component integrator:

- Make a thorough evaluation of the component suppliers. Are they suitable as a supplier? Do they have good quality products and support? Check their financial condition so they don't easily bankrupt.
- Put a lot of effort into the legal agreement with the supplier. This may save you if the supplier goes out of business or if they refuse to support you.
- Have key persons that are assigned to supervise the component market. They shall keep track of new components and trends.
- Test the components in your environment.

One of the most important implications of this approach is that the application source will be fragmented across the application and a collection of source component models. Until the components are error-free, such an environment will make maintenance a challenge.

4 Management Process for Releasing a New Component

In this paper, we suggest the techniques apply to the publication of a new component that is an upgrade to a previously published component. The new release may:

(1) Include meta information of component such as registration, management, composition, functionality, environment, and retrieval information as table 2 addition to the specification of the previous release, such as additional specification types, attributes, relationships, and constraints, or additional operations. This would be done without removing anything from the original specification.

(2) Have an identical specification to the previous release but a different implementation.

(3) Be allowed to read and update data storage from the previous release.

(4) Include alterations to the previous specification; that is, the changes may not be simply additions to the original specification.

(5) Be a combination of any of the above.

In order to make a distinction between a change that impacts the consumer of a component and one that does not, two terms are being introduced. 'Version' is a new release of a component that impacts the consumer and 'revision' is a new release of a component that does not impact the consumer. It is important to make these distinctions to determine whether a component can be upgraded and the old release removed without impacting a consumer, or whether there needs to be a transition time where both the old and new releases co-exist.

The figures 7 and 8 are scenarios showing version and revision changes where an operation for an interface in a component is first built. The component, interface and operation version, and revision number all start at 1. At Key1, the operation is revised such that it does not impact the consumer of this component.

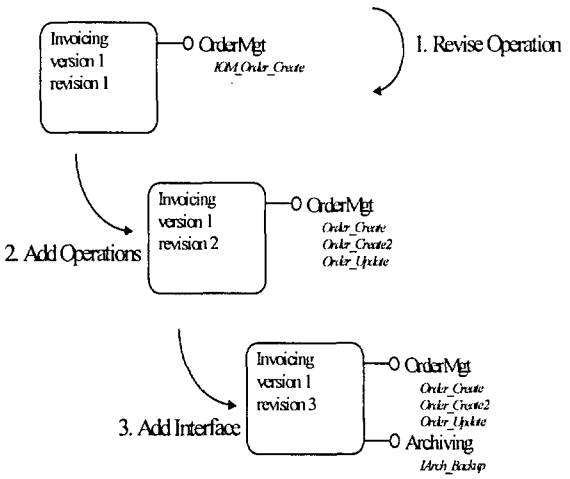


Fig. 7. The version/revision scenarios for 1st

The revision number on the operation, its interface, and the component are all incremented by 1.
At key 2, An operation is added. This does not impact the consumer. The operation is new so the version and revision numbers start at 1, but the interface and component revision numbers are incremented by 1. At key 3, An interface is added. Since it is a brand new interface with new operations, the version and revision number for the operation and interface are one, but the component has changed so the revision number for the component is incremented to 4

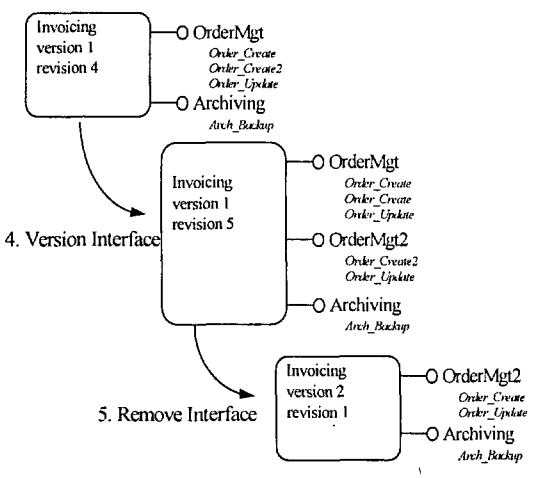


Fig. 8 The version/revision scenarios for 2nd

At key 4 in figure 8, a new version of the interface is published. This changes the versions of the operations and interface, but it is only a revision to the component. At key 5 in figure 8, An interface is removed. This changes the way a consumer can use the component.
Managing Versions:

Higher version numbers normally indicate improved functionality, which must be an extension to the previous release. Higher revision numbers within the same version number normally denote a fix or clean-up of the previous release.
Operation Version :
An operation version is created when the operation specification is changed. The operation version changes when:
(1) The operation "signature" changes are adding or removing import and/or export views, changing

Table 3. Scenarios Showing Version and Revision Changes

Component		Interface		Operation		
v.	r.	v	r	v.	r.	
1	1	1	1	1	1	
1	2	1	2	1	2	1. Revise op
1	3	1	3	1	1	2. Add op
1	4	1	1	1	1	3. Add interface
1	5	2	1	2	1	4. Version interface
2	1	2	1	2	1	5. Remove interface

specification type of import and/or export views and changing the order of import and/or export views.
(2) Changes are made to the existing specification types, subtypes, attributes, and/or relationships:
- Add, remove, or change the constraints such as cardinality, optionality, and uniqueness
- Change the length or data type of attributes
- Remove specification types, attributes and/or relationships
- A pre- or post-condition changes such that it no longer has the same implications as the old pre-/post-condition.
- Add or remove return codes
- A documentation change that no longer has the same implications as the old documentation. For example, the pre- or post-condition changes, and the documentation needs to reflect the change.
Interface Version:
The interface version changes when: An operation is removed. An operation version number changes and the new version of the operation is no longer compatible with the version of the interface that contains the operation.
Component Version :
The component version changes when an interface is removed.
A revision does not impact the consumer of the component; therefore revisions would be recorded in the documentation of the interface operations but would not effect the name of the interface or operation. Only the name of the component changes when the revision number is increased.
Operation Revision:
The operation is considered revised :

(1) When the operation specification is extended. Extensions to operation specifications are defined as: - Additional specification types, subtypes, attributes and/or relationships have been added.
- Constraints are added to new specification types, attributes, and relationships only.
- Constraints on existing types, attributes, and relationships must remain the same in order for this change to be a revision. Otherwise, it would be a new version.
- A change is made to pre- and post-conditions so they will conform to standards.
- Corrections and updates are made to documentation so it will conform to standards.

(2) When the implementation and executable is updated, such as "Bug" fixes, Performance improvements.

Interface Revision:
The interface is considered revised when an existing operation is revised and an operation is added. This includes an operation that leaves the interface compatible with the previous release. Interface revisions affect the interface description, which must be changed to reflect the revision.
Component Revision:
The component is considered revised when an existing interface revision occurs and an interface is added Component revisions affect the component model name.

5 Conclusion

Designing, developing and maintaining components for reuse is a very complex process which places high requirements not only for the component functionality and flexibility, but also for development organization. In this paper, we described the important issues to the development and management of components. We focus on setting standards for components and addresses the impact that CBD has on managing component development. This includes the model management strategy, development and delivery of components, adopted by an organization, and the capability to add new releases of components or parts of components.

We will put more effort to create an open and extendable architecture. We also will address the standard issues for component based software and CBD process, which covers component requirement analysis, component development, and component certification.

References

[1] George T. Heineman and William T. Councill, Component Based Software Engineering, Addison Wesley Publication Company, June,2001.

[2] Clemens Szyperski, *Component Software : Beyond Object-Oriented Programming*, January 1998, Addison-wesley.

[3] Mikio Aoyama, "New Age of Software Development : New Component-Based Software Engineering Changes the Way of Software Development," *1998 International Workshop on Component-Based Software Engineering, ICSE,* p.124~ 128,1998. http://www.sei.cmu.edu/cbs/icse98/papers/p14.html

[4] Robert C. Seacord, "Software Engineering Component Repository," *Proceedings of 1999 International Workshop on CBSE*, Los Angeles, at URL:http://www.sei.cmu.edu/cbs/icse99/cbsewkshp.html

[5] Luqi, Jiang Guo, "Toward Automated Retrieval for a Software Component Repository," *IEEE Conference and Workshop on Engineering of Computer-Based Systems*, March, 1999.

[6] Peter Herzum, Oliver Sims, *Business Component Factory : A Comprehensive Overview of CBD for the Enterprise*, OMG press, December, 1999.

[7]URL:http://java.sun.com/j2ee/tutorial/doc/EJBConcepts.htm

[8]URL:http://www.omg.org/gettingstarteded/specintro.htm/CORBA

[9]URL:http://www.effiel.com/doc/online/effiel45/papers/com/com/htm

[10] Haeng-Kon Kim, Jung-Eun Cha, Ji-Young Kim, Eun-Ju Park, Identification of Design Patterns and Components for Network Management System_, SNPD '00 International Conference, Vol. 1, NO. 1, pp. 426 ~ 431, May, 2000.

[11] Desmond Francis D'Souza, Alan Cameron Wills, Objects, *Components, and Frameworks With UML : The Catalysis Approach*, October 1998, Addison-Wesley Object Technology Series.

[12] Mark R. Vigder and W. Morven Gentleman and JohnC. Dean. components Software Integration: State of the Art. National Research Council of Canada, Institute for Information Technology report 39198, 1996.

[13] Alan W Brown and Kurt C. Wallnau "Engineering of omponent-Based Systems," *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute.* Los Alamitos, CA: IEEE Computer Society Press, pp.5-7, 1996.

[14] Oh-chun Kwon, "CBD Environment Model : An Integrated Approach of Object-Oriented Programming and Other Technologies", KIPS Software Engineering Review, 1999.

Image processing and becoming conscious of its result

Mitja Peruš
BION Institute; Stegne 21; SLO-1000 Ljubljana; Slovenia
Phone & fax: +386-1-513-1147; mitja.perus@uni-lj.si; http://www.bion.si/mitja.htm

Keywords: image processing, brain, vision, striate cortex = V1, consciousness, quantum associative net, Pribram

Received: March 7, 2001

Based on the Holonomic Brain Theory by Karl Pribram and related models, an integrated model of conscious image processing is proposed. It optimally incorporates contemporary limited knowledge starting from a systematic search for fit between existing computational models, and between available experimental data, and between data and models. Since we are not yet able to tackle qualitative conscious experience directly, processes for making an image (or result of image processing, respectively) conscious are discussed.

A quantum implementation of holography-like processing of images in the striate cortex (V1) is proposed using a computational model called quantum associative network. The input to the quantum net could be the Gabor wavelets, together with their coefficients, which are infomax-constrained spectral and sparse neural codes produced in the convolutional cascade along the retino-geniculo-striate visual pathway using the receptive fields as determined by dendritic processes. Perceptual projections are used as argument for holography-like and quantum essence of visual phenomena, because classically (neurally) alone they could not be produced in such a quality. Level-invariant image attractors are argued to be representations to become conscious in/by a subject, after a similar stimulus has triggered the wave-function collapse (i.e., recall from memory). Auxiliary representations for simultaneous subconscious processing, based on phase-information, for associative vision-based cognition are proposed to be Gabor wavelets (i.e., spectral codes in V1 receptive fields, or dendritic trees, respectively) and their coefficients (i.e., sparse codes in activities of V1 neurons).

1 Introduction

Aims and sources. This paper provides an *information-theoretic integrative model of conscious image processing* "having the kernel" in the *striate cortex* (named also the *primary visual cortex* or *V1*). Beside of an attempt to present a model that is an optimal compromise of biologically-plausible ingredients and relevant information-processing features needed for describing image processing in man, this study is interested in the problem how the *result* of image processing (the image representation) becomes conscious, i.e. how we become conscious of the perceived image.

The model is based on several earlier presentations of antecedent and accompanying physiological processes (Peruš, 2000b) and of information transfer and transformation along the visual pathway from the retina over the optic nerve and through the *lateral geniculate nucleus* (LGN) (Weliky & Katz, 1999) to V1 (Peruš, 2001). To provide a ground for the present study, a large body of neurophysiological, psychophysical, biocybernetical, neuropsychological, and other theoretical, experimental and simulation-based literature on vision (incl. reviews in: Kandel et al., 1991; Kosslyn & Andersen, 1992; Arbib, 1995) has been systematically studied, analyzed and compared in search for a synthesis (where possible). These data as well as several relevant models have been considered (Peruš, 2000a) in the context of Karl Pribram's (1991) *Holonomic Brain Theory*. Many informative complementarities were found (Peruš, 2000a, 2001). The present paper thus suggests a new comprehensive model of (conscious) image processing, while all the contextual processes – like visual attention and memory (Crick, 1984; Bickle et al., 1999; Vidyasagar, 1999; Wurtz et al., 1980; Desimone, 1996; Goldman-Rakić, 1996), stereopsis (DeAngelis, 2000; Porrill et al., 1999; Poggio et al., 1985), segmentation of figure from background (Sompolinsky & Tsodyks, 1994), perceptual binding (Roelfsema, 1998; Lee & Blake, 1999a,b) and imagery (Kosslyn, 1988) – have been integratively considered in auxiliary literature (Peruš, 2000a,b, 2001).

Early visual processing: infomax. Along the retino-geniculo-striate pathway (De Yoe & Van Essen, 1988; Livingstone & Hubel, 1988), a cascade of encoding / decoding processes, or convolutional processes, respectively, ensures optimal information pre-processing and encoding of images into various representations needed for visual cognition. Such preprocessing and encoding are realized, as psychophysical evidence (Wainwright, 1999; van Hateren, 1992) shows, so that *information is maximally preserved*, as is also imitated by the so-called *"infomax" models* of artificial neural net (ANN) processing. Many of them generate so-called sparse codes where an

oligarchy of units is active in encoding the entire image, but the majority is inactive.

It was realized (Peruš, 2001) that the infomax-models, like the *Independent Component Analysis* (ICA) by Bell & Sejnowski (1995, 1996, 1997) *and sparseness-maximization net* by Olshausen & Field (1996a,b; 1997), outperform the classical Hebbian or Principal Component Analysis (PCA) models (Haken, 1991, 1996), because they incorporate *phase information*, or higher-order statistics, respectively. Infomax-models were shown to give much more biologically-plausible outputs (receptive-field profiles[1]), but a biologically-plausible implementation on the "hardware"-level is possible (for now) only for the Olshausen & Field net, not for the Bell & Sejnowski net. Relations between the Olshausen & Field (1996a,b) net and MacLennan's (1999) dendritic field computation model were found (Peruš, 2001), which indicate a possibility of dendritic implementation of the Olshausen & Field net. However, dendritic processing "following the Olshausen & Field algorithm" would be strongly constrained by sparseness-maximization process which could originate from the lateral inhibition or from top-down (i.e., corticofugal) influences (e.g., Pribram in Dubois, 2000b; Montero, 2000; McIntosh et al., 1999; Moran & Desimone, 1985).[2]

Gabor wavelets. Since the oscillatory-dynamic phase-processing is experimentally supported (Gray et al., 1989; Baird, 1990; Pribram, 1971, 1991; Wang, 1999; Mannion & Taylor, 1992; Schempp, 1993, 1994, 1995; Sompolinsky & Tsodyks, 1994), a question arose whether ICA infomax-processing, or at least the sparsification process, might be realized virtually, i.e. on a "software"-level (higher-order attractor dynamics). ICA-like infomax processing shapes the receptive-field profiles into *Gabor wavelets*, and these are then convoluted with the sensory inputs (Pribram & Carlton, 1986). The infomax processing is thus viewed as an information-saving preprocessing procedure for optimal encoding into Gabor wavelets (also by other ICA models like: Harpur & Prager, 1996; Hyvärinen & Oja, 1998; Lewicki & Olshausen, 1999; cf., van Hateren & Ruderman, 1998).

As will be shown, infomax-based (appropriately weighted) Gabor wavelets are spectral image-representations (van Hateren & van der Schaaf, 1998) which are involved in convolution (during perceptual processing), or in interference, or in other *phase-Hebbian* processes (during pictorial cognitive processing and associations). Phase-Hebb learning rule, i.e. the Hebb correlation-rule with phase-differences (because complex-valued activities are correlated or convoluted) (cf., Sutherland, 1990; Peruš & Dey, 2000; Spencer, 2001), is a name I coined for the following expression for "holography-like" memory-storage into so-called connections (or weights, or interactions, respectively) J_{ij} between "units" i and j:

$$J_{ij} = \Sigma_k A_{ik} A_{jk} exp(i(\varphi_{ik}-\varphi_{jk})).$$

A is the activity-amplitude of a "unit", φ is its phase of oscillation; k is the eigenstate which represents a pattern or image.

Quantum implementation. In Peruš (1996, 1997a, 1998a) mathematical analogies in holographic, associative artificial-neural-net, spin-system and quantum-interference processes which could be harnessed for parallel-distributed information processing were systematically presented. Possible (biological) implementations of these processes were indicated. Furthermore, the *Quantum Associative Network*, an original computational model by Peruš (in Wang et al., 1998), was presented as a possible core-model for holonomic associative image processing in Peruš (2000a). Possibility for such a quantum image processing implies that the image, which is recognized by the quantum associative net, becomes the "object of our conscious experience". This hypothesis results from numerous indications (e.g., Goswami, 1990; Hameroff et al., 1994—1998; Lockwood, 1989; Rakić et al., 1997; Stapp, 1993; Peruš, 1997b) that consciousness is essentially related to quantum processes.

The comparative neuro–quantum study, and original derivation of the model of quantum associative net from the simulated neural-net formalism, are presented in detail in Peruš (2000c). Some resulting novel suggestions for flexible image processing (e.g., "fuzzification" harnessing "quasi-orthogonal" structure of data) are described in Peruš & Dey (2000) (cf., Kainen, 1992; Kainen & Kurkova, 1993; Kurkova & Kainen, 1996).

2 Attempt of an integrated model of image processing in V1, and beyond

Introduction to the model. The holonomic theory (Pribram, 1991) of the retino-geniculo-striate image processing (Pribram & Carlton, 1986; Peruš, 2000a) uses *Gabor wavelets* as "weighting"- or "filtering"-functions while performing convolution with the retinal image. The result of this Gabor transform is a spectral image-representation in V1. This, roughly hologram-like, representation in V1 is then reconstructed by an inverse Gabor-transform into the spatial representation in V2, probably. Thus, the topologically-correct "image" (cf., Tootell, 1998) is recovered in inverted form in V2.

The overlapped Gabor wavelets, which are used in image processing in V1, describe the receptive-field

[1] A *receptive field* of a neuron is everything (or the whole surrounding space or network, respectively) that influences its output *after* all the inputs have entered it along its own dendritic tree. The receptive-field's *profile* is a mathematical function describing the effect of transformations upon neuron's inputs (the "weights" of inputs) before the axonal output is "calculated".

[2] A "sparsification pressure" is imposed on dendritic (and maybe also on neuronal) processing in order to get maximally sparse codes. Biological realization of sparsification is unknown. It might originate in virtual higher-level attractor structures (the "software" level), maybe in a similar way as in Haken (1991). The second hypothesis, i.e. that lateral inhibition forces sparsification, is reflected in Pribram's (1998a) words: "[...] As the dendritic field can be described in terms of a spacetime constraint on a sinusoid – such as the Gabor elementary function, the constraint is embodied in the inhibitory surround of the field."

profiles of V1 neurons (Daugman, 1985, 1988) which realize the Gabor transform using their dendritic trees (cf., Berger et al., 1990, 1992; Artun et al., 1998). Gabor wavelets were shaped by an ICA-like process. Peruš (2001, 2000a) stated the reasons why it is good to prefer the Olshausen & Field (1996a,b, 1997) sparseness-maximization process over the ICA-variant of Bell & Sejnowski (1995, 1996, 1997) for the implementation-model of shaping the Gabor wavelets g (i.e., the independent components Y) and especially determining their coefficients s (i.e., the amplitudes or sparse codes of the independent components of input-images). The Gabor coefficients are updated much more rapidly than the Gabor wavelets. Coefficients change with each new input image, but the Gabor receptive fields adapt in a longer term – after a lot of different images have been presented. In fact, they adapt slowly all the time, but substantial change is seen after a while.

From Gabor receptive fields to wavelets. A Gabor elementary function g has two roles in modeling vision which seem to be somewhat different. First, it is used as a description of the receptive-field profile, i.e. of the "weighting" which the synapto-dendritic net imposes on all the inputs to a neuron. Second, it is used as a Gabor wavelet which represents the encoding of an independent component of input-images. Of course, both roles have the same origin (a sort of ICA) and "are two sides of the same coin". However, the consequence of infomax-processing *manifests in different places*: in the receptive-field profile which "lies hidden" in the synapto-dendritic net, *and* in the Gabor wavelet which propagates to other brain areas and gets involved in further holography-like processing there. Namely, if the receptive field is Gabor-shaped, then it gives Gabor-shaped outputs, or at least something similar or generalized, on a later stage. These Gabor wavelets might be of another sort – e.g., time-dependent or spectral.[3]

This effect (i.e., Gabor wavelets produced because/out of Gabor receptive fields) is more clearly evident if the retinal input is made uniform (i.e., "white noise" or Ganzfeld). This is related to well-known observations that an uniform stimulation triggers a system's response which is a sort of internal "expectation" (or a "hallucination", respectively) of the filtering system (e.g., MacLennan in Pribram, 1993, p. 189).

Sorts of representations. In principle, there are two sorts of representations in V1 available for further brain processing: the spectral "compound of images", or equivalently, the sparse assembly of codes (the so-called sparse codes) representing or "weighting" the independent components extracted from a collection of images. The first representation (independent component) is hidden in V1 dendritic fields; the second representation (sparse coding) is encoded in activities of V1-neurons (cf., Pribram et al., 1981). After the spectral representation of V1 has been inverse-Gabor-transformed in the connections between V1 and V2, the retinal image re-emerges in V2. Thus, the usual image, as once originally fallen on the retina, should be reconstructed (turned upside-down, maybe also somewhat "deformed") in V2 or nearby. This "image" is then the third available representation.

Why three representations? (Cf., Kirvelis in Dubois, 2000b.) We could suppose that the spectral (Gabor wavelet) representation is for perceptual image processing in V1. The sparse-code (Gabor coefficient) representation is for robust, rough encoding needed for automatic, immediate, reflex actions — they are unconscious and probably realized by neural circuits alone (dendrites just transmit the signals, do not process them). The "image" representation in V2 is used for the usual phenomenal conscious experience.

Let me explain. When a person is, say, involved in conversation, (s)he sees another person and at the same time processes a lot of information — e.g., "decodes" the other-person's body-language, not to mention more multi-modal and symbolic cognitive processes like understanding of words and thinking about the topic. The person sees the other one in full phenomenal integrity and quality all the time, without interruptions of the information processing (understanding body-language and spoken language, thoughts, etc.) going on in the background. For the seemingly-direct "realistic" experience of the environmental image, as conscious process offers it, the V2 space-time "image" is needed (more in Pribram, 1998a). For the abundant accompanying apperceptual processing (e.g., Luria, 1983; Stillings et al., 1995), which is unconscious and abstract, the spectral representation is needed (Pribram, 1997b). I will suppose that the "image" is also needed for additional processing, mainly limited to visual cognition, which uses associative processing that is more similar to holography than the perceptual spectral processing is.

From edge- to object-perception. Edges of object-forms are the first level of invariance or perceptual constancy and can be detected by linear transformations, like those in ICA. Incorporation of phase processing essentially improves results, as the Bell & Sejnowski and Olshausen & Field simulations demonstrate. However, finding transformations that are invariant to shifting, scaling and rotation of object-patterns, are mainly an open problem for ICA (Lee after Bell & Sejnowski, 1997). These transformations were with certain success tackled with generalized Hebbian models using Fourier-preprocessing (e.g., Haken, 1991) and by other non-infomax specialized models. ICA seems to be a good model for image (pre)processing, but not necessarily for object perception (which is well distinguished from image processing by the holonomic theory and other models of vision, and this has psycho-physiological reasons) (e.g., see Wallis & Bülthoff, 1999). Object recognition, based on search for perceptual invariances, might need a combination of ICA and

[3] They retain the same form, but with different interpretation of coordinate-axes. The Gabor wavelet in spectral representation see in Daugman & Downing (in Arbib, 1995). This Fourier transform of the original Gabor profile is expressed in the same functional form as the original spatial Gabor "wavelet", but with the spectral (u) and spatial (x) variables interchanged.

associative processing (a successful example is: Bartlett & Sejnowski, 1997; cf. also Gray et al., 1997), probably in attractor-networks which manifest gestalt-like structures (Luccio, 1993; Peruš & Ečimovič, 1998).

Phase-Hebbian associations take over. Peruš (2001) thinks that visual associative processes of V1, after perceptual preprocessing has been finished (ICA has generated Gabor wavelets and Gabor coefficients, and the Gabor transform using convolution has produced spectral responses of V1 simple cells), could be well realized by a Hebbian (e.g., Churchland & Sejnowski, 1992; Gardner, 1993) or phase-Hebbian mechanism. The second one, which is most similar to holography, has much more chances for good performance.

These models may in some respect be less efficient than other phase-processing models, which are not phase-Hebbian, like ICA and MacLennan's (in Pribram, 1993) dendritic model as far as it has similarities with the Olshausen & Field (1996a,b) model. But the phase-Hebbian models have a peculiar symmetry which makes them fundamental and close to physics. So, I believe, there is a division of labor. Other models (ICA, convolution-models, Kohonen's Self-Organizing Maps) "do the hard job" first. After their processing with nonlinear moments (like sparsification) is finished, the phase-Hebbian associative dynamics start the "fine job". Phase-Hebbian models have an ability to construct rich multi-level attractor-structures. In this, they can go beyond Hebbian models which are already successful in flexible attractor-dynamics (Haken, 1991; Peruš & Ečimovič, 1998).

Attractor processing. For secondary visual processing (i.e., e.g., object perception, from V1 to V2, and beyond it), processing with attractors is unavoidable (Peruš, 2000b, 2001). Pribram (1971, 1991) also says that cortical processing uses largely parallel-distributed and redundant representations. The model, which realizes this most directly and is also the best ANN embodiment of holography (cf., Psaltis et al., 1990), is sketched as follows. In simple words, the whole network of units with their connections encodes numerous "images" simultaneously: In the weight-matrix (encoded in the array of connections / junctions), there is the content-addressable associative memory. In the configuration-vector (encoded in the set of units), there is the "image" which is currently processed (which "we are *conscious* of"). Each vector of activity-configurations, which represents an "image", acts as an attractor of network's dynamics, because it is at the minimum of its potential well — as in the Hopfield model (details in: Peruš & Ečimovič, 1998; Peruš, 2000d).

In the matrix of connections (or "hologram"), not the whole patterns or images are stored. Merely pair-wise *(auto)correlations* of all previously-input images are stored. With other words, condensed information about (dis)agreements among all image-parts of all the watched images is encoded so that it is parallelly-distributed across the whole matrix. This is sufficient for reconstruction of an image from the memorized image-traces if a recall-key (i.e., a new, similar input pattern) is presented to the array of connectionist units (formal neurons) of the network. The interaction of these units across the connections is modeled by multiplication of the vector of units' states with the matrix of inter-unit connections (details in: Peruš & Ečimovič, 1998).

Quantum net as the inner processor. The essential point is that processing of the *Quantum Associative Network* (Peruš in Wang et al., 1998; Peruš & Dey, 2000), as derived from ANN in Peruš (2000c) based on analogies of Peruš (1996, 1997a, 1998), realizes the attractor-dynamics, associative processing and image recognition in a "compact" and effective way. It is progressive that Hebbian processing is enriched with phase-processing. Because this model can be quantum-implemented in a natural way, it is, for now, hard to imagine anything more fundamental, more holographic, more effective, and hypothetically more directly linked to conscious experience, in the sense of associative processing. Processing similar to that of quantum associative net might take place in V1 and partially also maybe in V2.

The quantum associative net can process, by interference, various kinds of eigen-wave-functions (eigen-images). They can be Gabor wavelets. Gabor wavelets are very similar to the natural quantum wave-packets (MacLennan in Pribram, 1993). The Gabor wavelet, originally proposed by the "discoverer" of holography D. Gabor in 1946, is a sequence of waves under a fixed gaussian envelope while the frequency of the wave inside the envelope varies for different cases. Such a wavelet is *equivalent to a family of Weyl-Heisenberg coherent wave-packets used in quantum physics* (Lee, 1996). This observation allows me to relate infomax image-processing with quantum-implemented holography-like associative processing with attractors – in the quantum associative net. Peruš & Dey (2000) present interference-processing in the quantum associative net using the plane-waves for image-bearing eigen-wave-functions — for simplicity, although efficient. This is the most usual / basic quantum-type holography. *Interfering Gabor wavelets* could enable more sophisticated, maximally information-preserving processing, in accord with the holonomic theory.

Associative basis for visual cognition. The uniformity of (neo)cortical structure (Ebdon, 1993; details in Burnod, 1990) allows the use of phase-Hebbian associative models for a rough (but maybe the best available) approximation of global (neo)cortical processing (cf., however, Ingber, 1998; Körner et al., 1999) which is at roots of visual cognition (cf., Clement et al., 1999; Pribram, 1997a). The proper modeling combination, I suppose, would be: *ICA-constrained convolutional preprocessing up to V1, followed by fractal-based associative processing in (neo)cortical neural, dendritic and quantum attractor networks – one within another inside V1.*

My hypothesis is that the multi-level phase-Hebbian associative processing, having the quantum associative net as the most deep/inner level (for now), is currently the most convenient one for *cognitive manipulations of images or*, rather, *object-forms*, as performed probably in the inferior temporal cortex (ITC)

(cf., Miyashita & Chang, 1988; Fuster & Jervey, 1982; Mishkin et al., 1983; Perret & Oram, 1998). ITC is specialized for prototype-converging recognition or *comprehension* of objects, including discriminations and choices resulting from it (Pribram, 1971). Global associations and context-searches are necessary during search of the right prototype. In accord with Rainer & Miller (2000), Riesenhuber & Poggio (2000) argue that the prefrontal cortex finishes the object-recognition started by ITC. They write on p. 1202: "In anterior ITC, invariances to object-based transformations, such as rotation in depth, illumination and so forth, is achieved by pooling together the appropriate view-tuned cells for each object." Then Riesenhuber & Poggio (2000, Fig. 3 caption) continue: "The stages up to the object-centered units probably encompass V1 to anterior ITC. The last stage of task-dependent modules may be located in the prefrontal cortex." These modules are needed for tasks like object-identification, -discrimination and -categorization, they say before.

Experiments of Rainer & Miller (2000) on object recognition in the prefrontal cortex showed that familiar objects activate fewer neurons than novel objects do, but these neurons are more narrowly tuned. Such a sparse representation of a familiar object is also more robust to degradation (made after the learning period) of a newly-posed stimulus-object. Based on ITC inputs (in vision), the prefrontal cortex is the region most important for the so-called working memory used in cognition. Present models use Hopfield-net-produced Hebbian[4] attractors for working-memory representations and attractor dynamics for (visual) cognition. I believe, generalization of these models (which were used in: Peruš & Ečimovič, 1998; Peruš, 2000d) by incorporating phase-processing (i.e., using the phase-Hebb rule) and implementing it in dendritic or/and quantum networks would be more appropriate.

It should be emphasized once again that image processing and subsequent object recognition could be possible only because of "the hard job done" by ICA and the perceptual convolutional cascade. They provided Gabor wavelets (cf., Pötzsch et al., 1996) and spectral representations of images which are still used in many higher cortical areas for more abstract processing (i.e., processing without the topologically-correct pictorial representation – the usual image). Volition-, "I"-based control- and symbolic processing are examples of abstract processing. On the other hand, operations of visual cognition — like imagery, mental manipulations of objects, visual modeling or planning (e.g., vividly imagining how to drive from A to B) (review in Baars, 1997) — could have cortical implementation based on the quantum associative net, especially if these processes are performed consciously. This fits the Crick and Koch (in Hameroff, 1996) hypothesis that we begin to be conscious of visual processing in V2 and beyond (encompassing visual cognition based on cooperation of ITC and the prefrontal cortex).

[4] Details on neurophysiological bases of Hebbian memory-storage see in: Gardner (1993), Abbot & Nelson (2000), Bear (1996).

3 Addition of conscious experience and quantum processes into consideration

From dendrites to conscious experience. The following citate from Pribram (1971, p. 105) summarizes the view which has been later elaborated by the holonomic theory:

"Neural impulses and slow potentials are two kinds of processes that could function reciprocally. A simple hypothesis would state that the more efficient the processing of arrival patterns into departure patterns, the shorter the duration of the design formed by the slow potential junctional microstructure. Once habit and habituation have occurred, behavior becomes "reflex" — meanwhile the more or less persistent designs of slow potential patterns are coordinate with awareness. This view carries a corollary, *viz.* that nerve impulse patterns per se and the behavior they generate are unavailable to immediate awareness. [...]

In short, nerve impulses arriving at junctions generate a slow potential microstructure. The design of this microstructure interacts with that already present by virtue of the spontaneous activity of the nervous system and its previous "experience". The interaction is enhanced by inhibitory processes and the whole procedure produces effects akin to the interference patterns resulting from the interaction of simultaneously occurring wave fronts. The slow potential microstructures act thus as analogue cross-correlation devices to produce new figures from which the patterns of departure of nerve impulses are iniciated. The rapidly paced changes in awareness could well reflect the duration of the correlation process."

Discussion on sorts of representation in section 2 seems to fit this citate. Sparse-coding assemblies of neurons (i.e., just few neurons fire, and this is enough for encoding entire images) serve in reflexes without awareness. The second representations of images, the Gabor wavelets, interfere in the (sub)dendritic microstructure. The correlation process, hidden in subcellular or quantum (as I prefer) interference, could also be accompanied by awareness. The final result of interference processing, the conscious image, would be reconstructed after the collapse of quantum (or at least quantum-like) wave-function.

In support for his hypothesis on junctional electric activity as the substrate for awareness, Pribram (1971, 1995) mentions that using biofeedback subjects can discriminate α-EEG waves in their brain by "feeling them as pleasantly relaxed awareness". He also cites Libet's findings that stimulation-produced awareness occurs in patients 0.5–5 seconds after the relevant brain-area has been stimulated — as if some electrical state would have to be built before the patients can experience anything.

Neural and quantum "sides" of dendrites. Infomax-processing is probably based in dendritic-*fiber* networks or/and neural circuits – on the (sub)cellular level, not quantum. V1 image processing and subsequent

visual associations are probably realized by quantum-based dendritic *micro*processes. Dendritic processing thus combines two levels. Its *macroscopic fiber-part* is involved, under some top-down influences probably, in shaping the Gabor receptive-field profiles by specific collective dynamics of dendritic trees of many neurons that criss-cross. Its *microscopic membrane–"bioplasma" part* (in the patches or "holes" in-between the criss-crossed dendritic fibers beneath their membranes) implements the holography-like image-processing, as will be described in section 6, but probably by interfering a sort of Gabor wavelets instead of plane-waves. Wavelets could be naturally rooted in quantum background.

Why fractal-like multi-level processing. Systematic observations show that brain structures repeat roughly on many levels or scales like in a fractal. Why such a (seeming?) redundancy? The answer is probably: flexibility, adaptability and universality.

Pribram (1971, 1991) observes that patterns shaped or learned in one part (area, level) can be transferred to another part (area, level) of the brain. One perceives an image, can recall it, manipulate it in imagination, one can use it to guide and control motor action directed toward its object (the image of achievement). In Pribram's example, one can draw a circle with a pencil on a paper or wall using fingers of a hand or even of a foot, or one can put the pencil in the mouth. *The same pattern* (circle) *can be produced* (drawn) *in different circumstances using different body-actions and different brain areas.* Even different levels of the tissue are needed: microscopic for processing, macroscopic for execution of action. To mobilize a muscle, amplification of (sub)neural signals is necessary – that's why neurons are needed also, not just dendrites and quantum systems. Neurons are cells – like the muscle cells are also. Since Nature is multi-scale, body and brain also have to be multi-scale to handle it. The many levels therefore have to cooperate fluently, so they must be compatible in information exchange. Patterns as global information therefore have to be able to travel from one level to another. This is realizable by *fractal-like* dynamics that is intrinsic to complex (bio)systems anyway. How the *inter-level or inter-scale transfer of patterns* or images is realized is much harder to find out than to realize that this is indeed happening (J. Anderson).

Attractors are very probably those emergent virtual structures which can "travel across the brain". They are the bioinformational or PDP (parallel-distributed processing) correlate of gestalts — each represents an invariant information-unit (percept). An attractor, a primitive "ghost in the machine", is rooted in a network-state, but *changes its substrate-elements* (Peruš & Ečimovič, 1998) like the electric current changes its underlying crystal structure or like (water) waves change (water) molecules while propagating.

4 Quantum associative network model

Essentials. A verbal (partially metaphoric) description of processing in the quantum associative net, in comparison to holography, will now be given (mathematical details in Peruš, 2000a,c). The processing of quantum associative net *is* a sort of holography, if one is allowed to use the term outside classical optics, since the net interferes quantum waves. In fact, the quantum associative net is a quantum-mechanics-based mathematical model which can be computationally simulated (cf., Zak & Williams, 1998). No reasons have up to now been found why the model could not be implemented in a real quantum-physical system. The model also needs *specific input–output transformations*, therefore it is an informational model as much as it is a physical one.

Interpretation of states. The quantum associative net is the core of basic quantum mechanics (in Feynman's interpretation) *which is put into an intelligible interaction* with the environment (visual field). This is new: the input–output dynamics. Another essential new thing is that eigen-wave-functions (i.e., the basic, natural quantum states – they are often particles–waves, but not necessary) are harnessed to encode some information like an image. An intelligent being must be there which interacts with the system in such a way that the input-, output- and internal (memory) states represent some *meaningful information for the being*. His *interpretation "transforms" an ordinary quantum system into an information-processing system* as soon as he is satisfied with the input-to-output transformations. Let us assume so. (It is like in the case of a round piece of wood "becoming" a wheel if put in a proper context – the axis, other wheels, upper plate, etc.)

Inputs. Image processing can be done during the holographic process (Pribram, 1991). It works perfectly and simply, as all physicists and opticians know (Hariharan, 1996). It is natural in holography (as well as photography and any other optics) that the light encodes the 3-dimensional form of an object by specific modulation (i.e., shaping) of amplitudes, frequencies and phases of its waves (rays). So, it is possible to encode complicated object-forms into usual electro-magnetic waves — even with perfect resolution when the code is being reconstructed. We thus have: objects, their codes or representations (in a medium), and we need object-to-code transformations (encoding) and, finally, code-to-object transformations (decoding or reconstruction).

Because holography works with all sorts of waves, the information-carrying waves can be quantum waves. This might bring new capabilities, but not eliminate the classical ones. Hence, the input-waves can be *plane-waves*, mathematically described by equation

$$\psi_k(\vec{r},t) = A_k(\vec{r},t)\exp(i\varphi_k(\vec{r},t)) \qquad (1),$$

or the input-waves can be *Gabor wavelets* (made of "increasing and decreasing waves under a gaussian envelope"). (ψ is the wave-function, A is its amplitude, φ

is its phase – at a specific location r in specific time t; k is the eigenstate index.)

We merely need means for proper manipulations of waves. Even for quantum waves, technology is so far today. Brain might also be able to do it. (Details see in Peruš, 1997a.) We thus "insert the inputs" by illuminating an object so that light-rays (or, in brain's complicated version, ICA-produced wavelets) "fall into" the quantum associative net.

Interference constitutes memory. We do that with different objects and let the waves, each belonging to an object, "mix together" while falling on a medium (the hologram-plate). This is interference of waves (like when two water-waves criss-cross) produced so that it leaves a trace (the hologram) on the medium. The wave-parts add or suppress each other (the constructive or destructive interference), and a criss-cross matrix of their relationships is recorded on the hologram. This hologram is a "frozen" *content-addressable associative memory* which becomes active when light is sent through it!

The quantum interference and quantum holography are when the waves and the hologram (but not necessarily the object) are quantum. The quantum hologram is the interference-pattern of quantum waves which leaves traces *in the quantum medium itself.* In quantum world, "parts are virtually a whole", so the waves and the hologram "inter-penetrate". Matrix $\boldsymbol{G}$, as given by the phase-Hebbian expression

$$G(\vec{r}_1,t_1,\vec{r}_2,t_2)=\sum_{k=1}^{P}\psi_k(\vec{r}_1,t_1)^*\psi_k(\vec{r}_2,t_2)=$$

$$=\sum_{k=1}^{P}A_k(\vec{r}_1,t_1)A_k(\vec{r}_2,t_2)e^{(i(\varphi_k(\vec{r}_2,t_2)-\varphi_k(\vec{r}_1,t_1)))} \qquad (2),$$

describes the quantum hologram. Its essential memory-"traces" are phase-differences in the exponent (cf., Ahn et al., 2000). Matrix $\boldsymbol{G}$ is at the same time the carrier and transformer of waves. $\boldsymbol{G}$ is the quantum-holographic memory which is active – performs associations "through itself".

$\boldsymbol{G}$ describes the "self-organizing internal restructuring" of the quantum system by "internal interactions between its (seeming) parts", i.e. by *self-*interference. It should be emphasized here that this is not an interaction in the sense of chemical or quantum-particle (nuclear, subnuclear) reaction, but in the sense of mutual mechanical (or electrodynamic / optical) influence or *re-arrangement* on a quantum level. In the language of quantum informatics, $\boldsymbol{G}$ describes a *compound.* (The "deeper, holistic" quantum fields incorporate *entanglements*, where parts which have once interacted cannot be really separated any more, but just seemingly. See experiment by Aspect et al., 1982.) Compounds can be "un-mixed" like the images can be reconstructed from the hologram (memory).

Associative processing. The matrix / hologram / propagator $\boldsymbol{G}$ describes phase-relationships between "infinitely"-small parts of the waves which were "mixed". This is associative memory, which also acts like a "turbine" for associative "computing". Each quantum wave ψ "flows through the $\boldsymbol{G}$-turbine", and this changes both the $\boldsymbol{G}$ and the wave. In mathematical description:

$$\Psi(\vec{r}_2,t_2)=\iint G(\vec{r}_1,t_1,\vec{r}_2.t_2)\Psi(\vec{r}_1,t_1)d\vec{r}_1dt_1 \qquad (3).$$

This implies, because equation (2) should be inserted into equation (3) to replace $\boldsymbol{G}$, that (and how, why) waves ψ change $\boldsymbol{G}$ and $\boldsymbol{G}$ changes waves ψ. This is called the coupled dynamics of the quantum system — it is a "self-holography" triggered by our inputs. We call it associative processing, because it is realized by "projecting" the quantum eigen-state or -wave "through the associative memory or hologram" $\boldsymbol{G}$. Initial quantum-encoded informational state (ψ_{in}) is thus transformed into an *associated* quantum-encoded informational state (ψ_{out}).

Image recognition by wave-function collapse. If we want to recall a memory, or to reconstruct a stored image out of $\boldsymbol{G}$, respectively, we have to present a part of the image or a similar image (the memory-"key") to the system (ψ_{in}). The similarity activates matching of relations, encoded in phases, and thus *selectively associates* the "key" with the most similar stored image which then "comes out of the mixture" (i.e., $\boldsymbol{G}$) in a clearly-reconstructed form. This is described by the following sequence of equations:

$$\Psi(\vec{r}_2,t_2=t_1+\delta t)=\int G(\vec{r}_1,\vec{r}_2)\Psi'(\vec{r}_1,t_1)d\vec{r}_1=$$

$$=\int[\sum_{k=1}^{P}\psi_k(\vec{r}_1)^*\psi_k(\vec{r}_2)]\Psi'(\vec{r}_1,t_1)d\vec{r}_1=$$

$$=\left(\int\psi_1(\vec{r}_1)^*\Psi'(\vec{r}_1,t_1)d\vec{r}_1\right)\psi_1(\vec{r}_2)+\ldots+$$

$$\left(\int\psi_P(\vec{r}_1)^*\Psi'(\vec{r}_1,t_1)d\vec{r}_1\right)\psi_P(\vec{r}_2)=A\psi_1(\vec{r}_2)+B \qquad (4)$$

where $A\approx1$ ("extracted image") and $B\approx0$ ("noise").

I can claim that this quantum process, called *"wave-function collapse"*, is *typically holographic* in the framework of quantum associative nets (details in Peruš, 1997a). It is also essential for all quantum measurements, where one "chosen" eigen-state ψ_k is realized in the quantum state ψ, all the other eigen-states "retreat" (into the implicate order). Thus, the input-triggered wave-function "collapse" is the *memory-to-consciousness transition.* An image is reconstructed from memory and simultaneously "appears in consciousness", because it has been associated with all the relevant contexts during this very process! Therefore, the image is also (consciously) *recognized* at the same time!

Remarks. Memory associations are encoded in correlations of wave-amplitudes A and additionally in differences of wave-phases φ. The latter encoding turns out to be more important and more fundamental, although both encodings are complementary (details in Peruš, 2000a; MacLennan, 1999; Sutherland, 1990).

In sum, the significance of the quantum associative net is in the fact that all the elements or aspects of an input-image are compared with all the elements or aspects of all the images, condensely stored in the system (as described by $\boldsymbol{G}$ or, alternatively, by the

so-called *probability-density matrix* ρ; cf., Alicki, 1997). An optimal, "compromise" output-image is then given as the result.

In the following sections, some related quantum or similar models will be presented, and they will be, together with the quantum associative net, discussed in the context of consciousness.

5 Holographic perceptual out-to-space back-projection and object—image match

Mathematical-physical description of holography. A hologram is a complex linear superposition of collective stationary interference fringe patterns. Storage of information (i.e., object-image) is usually made by global interaction (mixing) of a coherent information-bearing object-wave (reflected from the object) and a no-information-bearing coherent reference-wave under a particular angle. Information can be later retrieved by illuminating the hologram (no object needed any more) with the anti-wave of the original reference-wave used at storage. The anti-wave is an original-like information-bearing reference-wave (called phase-conjugate wave) in the opposite direction of the original wave.

Namely, *phase conjugation* refers to the change of sign (direction) of the phase (in exponent) and thus of the wave-vector: k to $-k$. Wave-vectors with opposite signs ($k=2\pi/\lambda$ and $-k$) indicate wave-propagation in opposite directions, but with the same wavelength λ. The phase-conjugate wave ($-k$) has, in the case of all local fields having the same frequency, an unique property to propagate back, in real or virtual form, along the path of the original wave (k). The advanced wave k and the retarded wave $-k$, which is *as-if* time-reversed, get superposed (giving $e^{2\pi i\nu t} - e^{-2\pi i\nu t} = 2\cos 2\pi\nu t$), due to precise timing. Thus, the *phase-conjugate wave (-k) propagates in the direction opposite to that of the original wave (k), similarly to propagation of the original wave backwards in time (as well as in space).*

Hologram's parallel-distributed organization is globally regulated by the local relative-phase variable implemented by the infinite-dimensional irreducible unitary linear representation of the Heisenberg nilpotent Lie group (Schempp, 1993, 1994, 1995; Marcer & Schempp, 1997, 1998, in Dubois, 2000b). The virtual "slices" (pages) of the hologram are frequency-organized, selective by incident angle of the page-oriented retrieval scanning reference-wave. Pattern/page-selection is executed by phase-conjugate adaptive resonance. The fractal self-identity is encoded in a hologram enabling reconstructional resolution proportional to any hologram's fraction where the total information is enfolded from. (Schempp, 1993)

Phase-Hebbian holographic associative memory has many concrete implementations, e.g. in photorefractive media — see Anderson (in Zornetzer, Davis & Lau, 1990, ch. 18).

Implementation of (bio)holograms. Hologram is realizable in fundamental (quantum)-physical media as well as in brain tissue. E.g., O'Keefe (in Oakley, 1985, p. 88-89) gives a concrete proposal of holographic processing in the hippocampus, and Nobili (1985) using damping-constant variations of local oscillators in the cortical glia-tissue. Neural holography could be realized by dendritic transmembrane-potential oscillations characterized by microwave-frequency coherence of the non-thermal excitation-states of biomolecules with high permanent dipole moments. The needed coherent "wave" could be *internally* incident. Holograms can be made also with *partly coherent* waves (Marcer & Schempp in Fedorec & Marcer, 1996; Hariharan, 1996), although usually coherent waves are used. Information between neurons is exchanged also independently of synaptic connections via glia-cells or non-anatomic coherent resonance coupling. Fundamental (sub)quantum holography could be realized with coherent overall wave-functions of dynamic quantum-vacuum (cf., Bohm & Hiley, 1993). A variety of holographic processes including single-state (e.g., single-photon) holograms are possible (why not, at least in a generalized sense, also in the brain?).

The fundamental Berry phase or geometric phase (Anandan, 1992) of the quantum system is promising for quantum memory and holographic (bio)information processing. A quantum system, which evolves so that it returns to its initial state, acquires a "memory" of this trajectory. This quantum "memory" is encoded in the Berry phase which is added to the phase of system's wave-function.

Holographic perceptual projections. We have an impression that an object we see is located "outside". The naive view is as follows: There is an object in the environment. Perception of it is produced in our brains in such a way that we see the object as it "really is – in external space". The perception appears to be somehow projected from the brain out to the original location. Pribram (1998b) mentions several cases of such perceptual or even cognitive projections: For example, one feels the paper, on which one is writing, at the tip of his/her pen, not at the tip of fingers holding the pen. A well-known case are also the so-called phantom-limbs — a patient feels the amputated limb which he has just lost. The pain is felt outside the remaining part of the limb — at the location where the former complete limb was or should be. In experiments, cited by Trstenjak, subjects spontaneously write on their own foreheads a letter (e.g., F) oriented as if they would read it from inside out (with their "mind's eye"), or as if they would write it on the internal side of their own foreheads. Projective nature and use of percepts are thus a part of human performance.

Let us illustrate how we could start to model holography-like perception and memory-recall. First, in stage 1, a subject faces an external, illuminated object. Light-waves reflect from object's surface toward subject's eyes. Image of the object is processed in his visual cortex, and gets memorized in a holography-like manner.

Later, in stage 2, the subject faces a similar object or the same one. Its light-wave indirectly interacts with holographic memory of the (original) object. A memory-image is remembered when the corresponding hologram-page is selectively reconstructed (as detailed in: Peruš, 2000c). By a *phase-conjugate wave*, the perception (a compromise of the stimulus and memory) is experientially projected back (from brain) into the surrounding space to the location of the original object. *The virtual (holographically back-projected) image of the remembered or perceived object coincidences in space with the original object.* This important and plausible hypothesis originates from: Marcer & Schempp (1998, in Dubois, 2000b). The idea of holographic back-projection by the (quantum) phase-conjugate wave is not yet finally proven, but it fits our *feeling* that the object and its brain-made and out-projected virtual image coincide out-there as if they are one!

The *perceptual* projection has been proven, for example, by the G. von Bekesy experiments (Pribram, 1971, pp. 168-171; 1991, pp. 90-91), but the *quantum-optical* phase-conjugate back-wave remains a question of quantum "reality" (i.e., theoretically it exists and is useful, experimentally there are indications, but there are different interpretations about their "reality"). The hypothetical back-projection waves would be *quantum* – not classical electro-magnetic waves like the input-waves to the retina. They would *not* exist *really* in the ordinary, i.e. classical-physical, sense. (These waves could "propagate backwards in time", "symmetrically" to the original input-wave, as in quantum field theory.)

To repeat: In stage 1, an original object is seen — a representation (a virtual "image") of it is produced in the brain. In stage 2a, a similar or the same object triggers remembering the original object of stage 1. In stage 2b, which follows stage 2a immediately, a joint perceptual image (usually a "compromise" of images of stages 1 and 2a) is projected back to the location of 1 or 2a, respectively. These stages of the dynamic holography-like process usually *iterate* and possibly converge to a maximal agreement of perception with phenomenal reality. Beside pragmatic advantages to the organism, this is also necessary for unambiguous, consistent communication between image-making subjects, since they share the perceived objects in positions relative to one another in the 3-dimensional "Cartesian theatre" they co-create (Marcer & Schempp in Dubois, 2000b).

Iterative matching loop. Thus, the image, which brain/mind creates, is perceptually coincident, maybe also quantumly coincident, i.e. phase-conjugate, in external space with the object imaged. In physical terms, there is a coincidence and annihilation ($\psi\psi^*$) of positive phases of forward-propagating waves (ψ, having wave-vector k) with negative phases of backward-propagating waves (phase-conjugate ψ^* with $-k$). Forward waves encode the original object and backward waves its perceptual image. *When they meet and match ($\psi\psi^*$) on the path they share, the perceptual transaction is completed, hence the wave-function collapse occurs* (Cramer, pers. commun.) *and so the image becomes conscious.* This adaptive-resonance hypothesis is best understood with the *transactional interpretation of quantum mechanics* by Cramer (1986).

One can ask: what is (or even: is there any) difference between the perceived object and the back-projected image of it (i.e., the image in the original location in the environment, not the image/code in the brain, say in V1). Disagreements lead to errors or misperceptions. The iterative matching process can also be led to creative generalizations and associations.

In *imagery*, or more plastically in hallucinations, the (possibly modified) back-projected image of the object replaces the non-existent object (which was present in stage 1, but not in stage 2 when the reconstructed reference-wave has some internal sources). In *perception*, the back-projected image phenomenally fits the real object.

Why (quantum) holography is necessary for spatial perception. Phase-conjugate projection of the image back into the space-location of the original object is an *exclusive characteristic of holography* (or at least of optics, if the image would perhaps be processed in another way, not holographically). Namely, *neural networks or other hard-wired subcellular networks, without having electro-magnetic or quantum embedding, **can definitely not back-project** their images into the external space on their own.* But we experience that perceptual projection is happening. Because holography (not photography) is involved, it is not directly the object-image that is back-projected (as in photography), but it is the *wave (-k)*, which carries the encoded object-image, that is back-projected by phase-conjugation during/after holographic retrieval (Marcer & Schempp, 1998, in Dubois, 2000b). So, *the external medium must be of the same or at least similar nature than the medium of the brain-hologram.* Hence, the *common medium can only be **quantum field**!*

Philosophical questions. Actually, there can be no plastic, geometrically-/ topologically-correct 3-dimensional perception of the object, which we experience and call "the real/true perception", without that fitting of the object with the back-projected image of it (which has been a moment earlier reconstructed from the "brain hologram"). This iterative fitting seems to need time, unless the Cramer (1986) transactional interpretation with "quantum waves backward in time" is adopted; but in fact space-time is co-created by ("in") our conscious experience "during" this visual processing. Objects and brain-states seem to be located in space and time, but conscious experience "has been / is / will be there all the time – as long as the Cartesian theatre is in the play". (E.g., Cramer (1986) even says that his interpretation, or a quantum transaction, respectively, is *a*temporal.)

Nothing is perceived outside mind: There are no perceptions, which we are aware of, without consciousness (i.e., conscious experience), and there are no phenomenal unconscious or subconscious sensations or "detections" (i.e., perceptions we are not aware of) without mind. A Kant's *Ding-an-Sich* (thing-in-itself) is

not perceived; just the co-created thing is, i.e. a "deformed shadow" of the hypothetical *Ding-an-Sich*. Thus, a perception is a Plato's "shadow" created by consciousness in cooperation with Nature, or "deformed" by brain-processing.

Various back-projections, like visual, tactile, auditory ones, match with the object and with each-other! Our vision "quantum-touches" the object successfully as well as our hands mechanically touch it simultaneously. Indeed a peculiar space-time fit. (But maybe this "resonance", and space-time incorporating objects, and conscious experience including objects, is the *same* thing/process... However, this symmetry or harmony (fit) may be broken, e.g. in hallucinations... The dilemma of reality thus remains.)

6 Dendritic holography-like image processing

Bioplasma dynamics. How can a dendritic network (physiology see in: Pribram, 1991, 1993; Damask & Swenberg, 1984; Koch, 1997; Koch & Segev, 2000) implement holography-like image processing of V1? Stimulus-specific waves of the dendritic field are produced, and these information-encoding waves interfere. A phase-Hebbian PDP is realized.

There are (at least, roughly) two related kinds of collective oscillations accompanying the dendrites criss-crossed in networks: first, the oscillations of dendritic membrane potential, and second, the oscillations of dendritic ionic "bioplasma", or of the electric polarizations within, respectively. The "bioplasma" "flows" and "waves" near the membrane-surface of the dendrite. It depends on the biomolecules (proteins, lipids, etc.) of high dipole moments located on (beneath, in, near, along, respectively) the dendritic membrane, and the ionic charge travels through it.

This membrane–"bioplasma" system of numerous coupled electric dipoles exhibits dynamic ordering which is determined by the distribution of phase-differences in oscillations of the corresponding dendritic potentials. The isophase-contours of oscillations in the polarization-field, extended over dendrites (especially their membranes) and the accompanying "bioplasma", determine the collective wave- and fluid-phenomena that are the *correlates of image processing at the subcellular level of V1* (after Appendix A of Pribram, 1991).

The Jibu, Yasue and Pribram model. The density (or concentration) of the ionic "bioplasma" $\rho(x,t)$ changes as a result of the dynamic pattern of hyperpolarizations and depolarizations across the space inside and outside dendrites. Yasue, Jibu & Pribram (in Appendix A of Pribram, 1991) defined a wave-function $\psi = \sqrt{(\rho)}\, e^{i\varphi}$ (φ is the phase). They derived (*ibid.*, pp. 282-286) a wave-equation for the membrane–"bioplasma" system:

$$i\nu\frac{\partial\psi}{\partial t} = \left(-\frac{\nu^2}{2}\nabla^2 + U_{ex}\right)\psi \qquad (5)$$

which has the same mathematical form as the Schrödinger quantum wave-equation (cf., Bonnell & Papini, 1997), but with different interpretations of variables. U_{ex} is the external static energy / potential, i.e. the external electric influence (stimulus). ν is a constant (a "relative" of the quantum Planck constant) equal to a quotient of "flow"-velocity v and the length of the so-called wave-vector k which is equivalent to spatial frequency, related to the change of phase: $k = \nabla\varphi$.

Successful derivation of such an equation, having a characteristic form for wave phenomena, for an idealized dendritic network / field shows that global polarization-waves, described here by ψ, emerge in the subcellular medium. As a consequence of these waves, "flows" and interference are also produced in the dendritic net. There are, of course, many variations of oscillations / waves / interferences in that complex medium. ψ (with different interpretations) could be chosen to approximate (m)any of them.

Phase-Hebb-like memory-storage in bioplasma. Wave-equation (5) is alone not enough for image processing. From wavelets

$$\psi(x,t) = \sum_n c_n\left(\frac{1}{L}\right)\exp\left[\frac{i}{\nu}\left(\frac{2\pi}{L}\nu x - \lambda_n t\right)\right] \qquad (6)$$

we obtain (details in Pribram, 1991, A, pp. 288-291) the density $\rho(x,t)$ of charge-distribution in "bioplasma":

$$\rho(x,t) = |\psi(x,t)|^2 = \psi(x,t)^*\psi(x,t) =$$

$$= \sum_n |c_n(t)|^2\frac{1}{L^2} + \sum_{n\neq n'} c_{n'}^*(t)c_n(t)\frac{1}{L^2}e^{\left[i\frac{2\pi}{L}(n-n')x\right]} \qquad (7).$$

c_n are the Fourier coefficients; L is a characteristic spatial extent of the dendritic system along the spatial axis x; λ_n is a constant; n, n' are integer numbers. The last term manifests interference, which is essential for holographic memory, since it has a phase-Hebb-like structure: $c_{n'}^*(t)c_n(t)$ represents "interference of amplitudes" and $exp[(2\pi i/L)(n-n')x]$ "interference of oscillations" – in the exponent one finds the phase-differences.

The flow of dendritic perimembranous "bioplasma" is driven by the phase-differences among isophase-contours (i.e., curves connecting all synchronized oscillations). The "density-based flow" toward an attractor at the centre of the concentric contour-family is regulated by the gradient (i.e., maximal change-rate) of phase ($\nabla\varphi$). Exactly:

$$\frac{\partial\rho}{\partial t} = -\nu\nabla\cdot(\rho\nabla\varphi) \qquad (8).$$

The wave-equation (5) and equation (8) describe the subcellular "fluid-dynamical" correlate of associative image processing in the V1 dendritic net.

Papers like Berg et al. (1996), Bray (1995), and those in Fedorec & Marcer (1996) support the possibility of biomolecular realization of holographic processing at

the subcellular level (cf., Nobili, 1985; Psaltis et al., 1990; Sutherland, 1990; Snider et al., 1999). These dendritic dynamics (Yasue, Jibu & Pribram in: Pribram, 1991, Appendix A) are in general principles equivalent to processing of the quantum associative net, but incorporate some sophisticated constraints imposed by characteristics of the subcellular tissue. The difference also is that the dendritic wave dynamics are quantum-like, but quantum associative nets are purely-quantum.[5]

7 Microtubules, coherent subcellular and quantum processes, and consciousness

Microtubules. *Microtubules*, cilindric / filamentary tubes, are the most important ingredient of the *cytoskeleton* which is a protein-made intracellular network. Microtubules extend along dendrites, come together at soma, and extend further along axons. They consist of oriented assemblies of electrical dipoles, or permanent electric polarization systems (electrets), respectively, which globally act as mega-dipoles.

A hypothesis (by Hameroff, 1994) with increasing support has been presented that cytoskeletal microtubules, constituting a network in cooperation with MAP (microtubule associated proteins), realize sub-cellular information processing based on coupled oscillatory collective dynamics. Since Hameroff & Penrose (1995) emphasize that mainly *dendritic* microtubules act such a role, this hypothesis might not be entirely incompatible with the holonomic theory, but complementary. Such dynamics emerges from conformational transitions of tubulin electric-dipolar molecules, which act as "bit flips", and from soliton-based, almost loss-less transfer of energy and information (phase!) along the paracrystalline microtubules. Tubulin states might encode the pixels of patterns which are processed (Hameroff, 1994; Nanopoulos, 1995). Globally, the processing is manifested in changes of concentration of electric polarization (polarization density), and moving of the concentration peaks from one side of the tubulin-web to another.

Long-range quantum coherence and related laser-like, thermal-noise-free (and information-loss-free) ordering phenomena, like super-radiance and self-induced transparency may take place in microtubules. The hypothesis that quantum coherence subserves binding of conscious perceptions is supported by an increasing number of authors (e.g., Hameroff et al., 1996). Microtubules are viewed by Jibu & Yasue (1997) as information-encoders into a coherent subcellular optical PDP network. Fröhlich (1968) has shown the first signatures of interdependence of biological and quantum oscillatory dynamics.

Fröhlich coherence. As proposed by Fröhlich (1968) and succesors, the so-called Fröhlich (long-range, microwave) coherence emerges from interacting oscillating (10^{11}-10^{12} Hz) dipoles of biomolecules. Electric polarization density serves as the biological order-parameter. Fröhlich coherent oscillations may lead to two sorts of extreme collective states: to the Bose-Einstein condensate, where all dipole-elements act as if they are one, or to loss-less solitonic polarization-waves (proposed by the Davydov model), where the dipole order propagates as one traveling condensate (Denschlag et al., 2000). This is analogous to superconductivity. Indeed, it was proposed by Jibu & Yasue (1995, 1996, 1997) that the experimentally-supported Fröhlich waves along the protein filaments can propagate without resistance, thermal loss and damping. Such superconductivity hypothetically occurs even at body temperature.

Subcellular automata. Many nanobiological systems could be represented as assemblies of dipoles: 1. cell membrane as a double sheet full of dipoles; 2. cytoplasmatic and extracellular water; 3. microtubules as systems of tubulins; etc. Systems of dipoles or spins can be arranged: 1. randomly; 2. ferroelectrically (i.e., aligned in parallel); 3. as spin-glass (i.e., in domains of frozen (dis)order, each with its own parallel alignment) (Mezard et al., 1987). The membrane bi-layer of dipoles might incorporate sandwich-like *Josephson junctions*, over which superconducting electrical currents with special effects would flow. They might be connected into a peculiar PDP "Josephson net-computer" (Rein in Pribram, 1993; Jibu & Yasue, 1995).

Quantum effects in synapse. Eccles (e.g., in Pribram, 1993) pioneered the idea that conscious mind, using attention, could influence the probability of discrete (quantal) release (*exocytosis*) of vesicles full with neurotransmitter-molecules at the hexagonal-paracrystalline presynaptic vesicular grid. Conscious mind would impose effect on probabilistic quantum processes (e.g., the wave-function collapse) underlying the probabilistic exocytosis in synapses. So, conscious process would selectively modulate, through quantum fields, the essential ingredients of memory-storage and associative processes – synaptic efficiacies (Rein in Pribram, 1993). To be precise, quantum influences should trigger electronic rearrangements resulting in movement of hydrogen-bridges which would effect vesicle-release from the presynaptic hexagonal grid (Hameroff, 1994).

Collapse and consciousess. The hypothesis of Hameroff & Penrose (1995, 1996, 1997) advocates microtubules and their nets as the main subneuronal substrate of consciousness. They are flexible, fast-changing and might allow retrograde signaling, thus mediating bi-directional subneuronal links between synapses. Hameroff (1994) argues, based on observations, that general anaesthetics cancel conscious experience by operating mainly on specific microtubular ingredients. He writes that an anaesthetized brain usually

[5] There is a different interpretation of the wave-function ψ which here corresponds to the "bioplasma-density" ρ – in contrast to Peruš, Bohm and the Copenhagen quantum interpretations. Namely, the "bioplasma-density" ρ does not exactly correspond to the quantum density matrix ρ, because the amplitude of the "bioplasma"-ψ is defined as $\sqrt{(\rho)}$ (not quantum!) to get the "quantum" $\psi^*\psi = \rho$.

remains quite active (as persistent EEG, evoked potentials and autonomic functions show), but this activity is neural, not microscopic quantum. So, quantum coherence, which gets disrupted by anaesthetics, should be essential for conscious experience.

In contrast to the environment-induced wave-function collapse of quantum theory (and of the quantum associative net), the wave-function collapse in microtubules is supposed by Hameroff & Penrose (1995, 1996) to depend on quantum gravity: Condensates which become larger than a threshold-size should cause their common wave-function to collapse "under their own mass". This would be thus a self-collapse called *orchestrated objective* collapse. Each such collapse is considered by Penrose and Hameroff to be a single conscious event. A temporal sequence of such conscious "nows" would constitute the "flow" of conscious experience, by this hypothesis.

According to the Hameroff & Penrose (1995, 1996) sketch, preconscious net-computing, when the classical (sub)neuronal PDP (parallel-distributed processing) is gradually replaced / complemented with quantum PDP, leads to emergence of a quantum coherent superposition. Each of its superposed alternative states has its own "competing" space-time geometry. When an instability-threshold is exceeded, the time-irreversible orchestrated objective self-collapse occurs, and this *is* the conscious experience (the "now"). This moment of maximal coherence "illuminates" the (results of) preconscious network-processing of images etc. (executed till the collapse) "by making it conscious" at the very moment of collapse which "chooses" one from many alternatives. The selected information-state (e.g., a recognized image) is further-on processed unconsciously in a classical way — until a new quantum coherence "consciously illuminates" the new mental state to make a qualitative experience of it, says the hypothesis.

Subcellular "holography". A number of oscillatory network-structures were mentioned: electric-dipole systems, microtubular nets, "bioplasma", extracellular matrix, protein nets, Josephson-junction nets, etc. They individually or in cooperation (as is usual) are able to exhibit holography-like interference processing (Pribram, 1993). However, molecular vibrational fields in these nets are just a sort of *interface between quantum networks and neural networks*. They all are very probably influencing, directly or indirectly, the synaptic efficiacies (e.g., whether they are inhibitory or excitatory, and how much) (e.g., Nanopoulos, 1995).

Subcellular coherence. What follows, is based on speculations by Jibu & Yasue (1995, 1996, 1997), derived from quantum field theory of Umezawa.

Beneath many levels of cell's biomolecular structure, many levels of sub-atomic or inter-atomic quantum particles, and their ensembles or condensates, can manifest collective dynamics capable of coherence and interference processing of holography-like sorts (reviews in Pylkkänen & Pylkkö, 1995; Pribram, 1993). They should mainly "live" in the intra- and inter-cellular *water* which composes more than 70% of the material composition of brain-cells like neurons and glia. These particles should collectively take part in water's rotational fields (or spinor-fields, emerging from spins of particles or from molecular spinning dipoles) and their interactions with the electro-magnetic field (i.e., with its quanta – photons).

Of special importance is supposed to be the Nambu-Goldstone boson (a sort of dipole phonon) which is a mass-less quantum of a long-range correlation-wave of the water rotational field created in an ordered vacuum-state.

Quantum binding. Macroscopic condensates of Nambu-Goldstone bosons are, after the hypothesis of Jibu & Yasue (1995, 1996, 1997), the fundamental carriers of perceptual memory and cues for reconstruction of the original stimulus-perception. Since they depend on the interaction of radiation (photons) and dipoles, they lead to evanescent (i.e., virtual, tunneling) photons which may collectively produce *Bose-Einstein condensates*. In such a condensate, many particles merge into a collective, unified, macrosopic quantum state. Particles (e.g., photons), which are able to unite into such a coherent condensate, are called bosons; particles (e.g., electrons), which never occupy the same quantum state, are called fermions. A Bose-Einstein condensate of evanescent photons is proposed to be the ultimate neurophysical correlate of an *unified conscious experience*. In Hameroff (1998, Box 1) and Jibu & Yasue (1995, 1996, 1997) suggestions are given how the Bose-Einstein condensates could be shielded enough by special biomolecular structures against the destroying thermal fluctuations.

The coherent dipole-field (i.e., having dipoles oriented in the same direction) of water might extend over the whole brain tissue or even whole body, not just over several cells. The coherence-length, i.e. a "diameter" of the region of coherent oscillations of all net-elements like dipoles, is calculated to be in the case of outer perimembraneous water about 20 to 50 μm (Jibu & Yasue, 1997) (more than cell-dimensions). Such ordered water with presumably laser-type processes is assumed to enable photonic holography in and around microtubules and in extracellular matrices (Jibu & Yasue, 1995; Hameroff, 1994).

Qualia unexplained. I can agree (Peruš, 1996-2000) with Hameroff and Penrose that the wave-function collapse seems essential for transitions from subconsciousness (or preconsciousness, or unconscious memory) to conscious experience. It also illustrates the classical–quantum (neuro–quantum, macro–micro) transitions. But saying (Hameroff & Penrose, 1996) that only and merely the "orchestrated collapse" (not any usual stimulus-induced collapse) provides the non-computable element necessary for consciousness, does not give any explanation of the qualitative experience.

Namely, the central characteristics of consciousness are *qualia* which are subjective, qualitative, phenomenal experiences ("how things seem to be to us") (Flanagan, 1992; Davies & Humphreys, 1993; Marcel & Bisiach, 1988). Examples of qualia are experiencing yellowness of a lemon, feeling pain in own elbow, and in general also *what it is like to be* a person,

etc. Qualia are felt in first person only, not in third person. A blind person cannot imagine precisely how it is to see; person A does not know precisely how person B feels. Qualia cannot be identified with their neural correlates — these are discussed, for example, in: Newman (1997), Frith et al. (1999); for color in: De Valois & Jacobs (1968), Schiller & Logothetis (1990).

Anyway, one might speculate that the usual stimulus-induced collapse is related to conscious perception of the stimulus, and that the orchestrated objective (if indeed induced by quantum gravity) collapse is rather related to (introspective) awareness. Although this hypothesis provides relations of conscious process to the origin of space-time, the problem of qualia remains. Qualia are only (with justice, I think) transferred to the most fundamental level (also). This could be concluded also for the suggestions of Jibu and Yasue: They might "explain" the origin of the unity of conscious experience, but not its qualitative phenomenal character.

8 Conclusions

This sketched integrated model based in the abundant literature of cognitive neuroscience (review in Kosslyn & Andersen, 1992), but transcended it by introducing fundamental informational (bio)physics. The latter seems to be needed (e.g., Bob & Faber, 1999) and promising (e.g., Dubois, 2000a,b; Pessa & Vitiello, 1999) for modeling quantum background of conscious processing (cf., Ezhov et al., 2000, 2001; Weinacht et al., 1999; Rabitz et al., 2000; Snider et al., 1999; Spencer, 2001; Wheeler & Zurek, 1983; Jones et al., 2000 – for hints from frontier technology).

According to the holonomic theory (Pribram, 1991), holography-like parallel-distributed processing in dendritic networks of V1 is essential for image processing. To be more specific, elèctric polarization fields or quantum fields and their wave phenomena are inside dendritic criss-crosses could be the central "medium" for processing. Here, it was proposed that the image-bearing eigen-waves, which interfere in the quantum associative net, are or at least could be infomax-produced, quantum-rooted Gabor wavelets. I thus suggest that the neocortex uses three types of image representations: the Gabor coefficients as sparse neuronal codes for automatic processing, the dendritically-implemented Gabor wavelets as spectral codes for associative visual cognition, and the V2-reconstructed spatial image used in our "direct" conscious experience.

Because the perceptual image seems to match precisely the original object in its external location, holographic back-projections by phase-conjugation have been argued to be necessary. Since neural or dendritic nets alone cannot realize such out-to-space projections, the only medium which is common to holography and to brain networks was declared ultimately responsible for conscious perception — the quantum system.

These ideas have been presented in the context of models by Pribram, Jibu and Yasue, Hameroff and Penrose, among others. Together they constitute, I believe, a view on systems-processing backgrounds of conscious image processing that provides an optimal integration of complementary proposals by the mentioned authors based on current knowledge. In accord with other views, the wave-function collapse was argued to be the physical correlate of becoming conscious of a selected image. The problem of qualia remains unsolved.

Concerning the kernel of the presented model, I can assume with much optimism that the quantum associative net, if really quantum implemented (as also, in a way, probably in brain), would realize efficient image recognition and related associative processing. Systematic comparison with extensive cognitive-neuroscientific literature allows me to assume that in cooperation with other brain structures, such an image processing would probably be conscious. A forthcoming paper will discuss results of the present paper in the context of experimental data on neural correlates of conscious visual experience and its impairments such as blindsight (Koch in Hameroff et al., 1996; Davies & Humphreys, 1993).

Acknowledgement

I am much honoured to be able to express cordial thanks to my great teacher Professor Karl H. Pribram (Stanford and Georgetown Universities) for his significant help and deep comments during our extensive discussions or as written notes into an earlier manuscript. Discussions with Professors D. Kirvelis, J. Bickle, J. Cramer, J. Gould, J. Glazebrook, G. Vitiello, A. Vogt, A. Železnikar, A. Župančič, D. Raković, with Drs. P. Marcer and R. Bogacz, and with A. Detela, are gratefully acknowledged. Many thanks also to Professors J. Musek, A. Ežov, P. Kainen, S. Dey and S. Pejovnik, to the National Institute of Chemistry and to MŠZŠ for financial support, to Drs. M. Škarja, A. Zrimec and R. Ružič for help, and to Mrs. Katherine Neville for hospitality.

References

[1] Abbot, L.F. & S.B. Nelson (2000): Synaptic plasticity: taming the beast. *Nature Neurosci. (Suppl.)* **3**, 1178-1183.

[2] Ahn, J., T.C. Weinacht & P.H. Bucksbaum (2000): Information storage and retrieval through quantum phase. *Science* **287**, 463-465.

[3] Alicki, R. (1997): Quantum ergodic theory and communication channels. *Open Systems & Information Dynamics* **4**, 53-69.

[4] Anandan, J. (1992): The geometric phase. *Nature* **360**, 307-313.

[5] Arbib, M.A. (Ed.) (1995): *The Handbook of Brain Theory and Neural Networks.* Cambridge (MA): MIT Press.

[6] Artun, Ö.B., H.Z. Shouval & L.N. Cooper (1998): The effect of dynamic synapses on spatiotemporal receptive fields in visual cortex. *Proceedings of the National Academy of Sciences of USA* **95**, 11999-12003.
[7] Aspect, A., P. Dalibard & G. Rogier (1982): Experimental test of Bell's inequalities using time-varying analyzers. *Physical Review Letters* **49**, 1804-1807 (and also: A. Aspect, P. Grangier & G. Rogier: *Phys. Rev. Lett.* **47** (1981) 460- &. **49** (1982) 91-.)
[8] Baars, B.J. (1997): *In the Theater of Consciousness*. New York: Oxford Univ. Press.
[9] Baird, B. (1990): Bifurcation and category learning in network models of oscillating cortex. *Physica D* **42**, 365-384.
[10] Bartlett, M.S. & T.J. Sejnowski (1997): Viewpoint invariant face recognition using independent component analysis and attractor networks. *Advances in Neural Information Processing Systems* **9**, 817-823.
[11] Bear, M.F. (1996): A synaptic basis for memory storage in the cerebral cortex. *Proceedings of the National Academy of Sciences of USA* **93**, 13453-13459.
[12] Bell, A.J. & T.J. Sejnowski (1995): An information-maximization approach to blind separation and blind convolution. *Neural Computation* 7, 1129-1159.
[13] Bell, A.J. & T.J. Sejnowski (1996): Learning the higher-order structure of a natural sound. *Network: Computation in Neural Systems* 7, 261-266.
[14] Bell, A.J. & T.J. Sejnowski (1997): The "independent components" of natural scenes are edge filters. *Vision Research* **37**, 3327-3338.
[15] Berg, R.H., S. Hvilsted & P.S. Ramanujam (1996): *Nature Lett.* **383**, 505-508.
[16] Berger, D., K. Pribram, H. Wild & C. Bridges (1990): An analysis of neural spike-train distributions: determinants of the response of visual cortex neurons to changes in orientation and spatial frequency. *Experimental Brain Research* **80**, 129-134.
[17] Berger, D.H. & K.H. Pribram (1992): The relationship between the Gabor elementary function and a stochastic model of the inter-spike interval distribution in the responses of visual cortex neurons. *Biological Cybernetics* **67**, 191-194.
[18] Bickle, J., M. Bernstein, M. Heatley, C. Worley & S. Stiehl (1999): A functional hypothesis for LGN-V1-TRN connectivities suggested by computer simulation. *J. Computational Neuroscience* **6**, 251-261.
[19] Bob, P. & J. Faber (1999): Quantum information in brain neural nets and EEG. *Neural Network World* **9**, 365-372.
[20] Bohm, D. & B. Hiley (1993): *The Undivided Universe (An ontological interpretation of quantum theory)*. London: Routledge.
[21] Bonnell, G. & G. Papini (1997): Quantum neural network. *Internat. J. Theoretical Physics* **36**, 2855-2875.
[22] Bray, D. (1995): Protein molecules as computational elements in living cells. *Nature* **376**, 307-312.
[23] Burnod, Y. (1990): *An Adaptive Neural Network: the Cerebral Cortex*. London: Prentice Hall.
[24] Churchland, P.S. & T.J. Sejnowski (1992): *The Computational Brain*. Cambridge (MA): MIT Press.
[25] Clement, B.E.P., P.V. Coveney, M. Jessel & P.J. Marcer (1999): The brain as a Huygens machine. *Informatica* **23**, 389-398
[26] Cramer, J.G. (1986): The transactional interpretation of quantum mechanics. *Reviews of Modern Physics* **58**, 647-687.
[27] Crick, F. (1984): Function of the thalamic reticular complex: The searchlight hypothesis. *Proceedings of the National Academy of Sciences of USA* **81**, 4586-4590.
[28] Damask, A.C. & C.E. Swenberg (1984): *Medical Physics*. Orlando: Academic Press. *Vol. III: Synapse, Neuron, Brain*; ch.4: Chemical and electrical properties of synapses; ch. 5: Neuronal integration and Rall theory. *Vol. I: Physiological Physics*; ch. 3: The nerve impulse.
[29] DeAngelis, G.C. (2000): Seeing in 3 dimensions: Neurophysiology of stereopsis. *Trends in Cognitive Sciences* **4**, 80-90.
[30] Daugman, J.G. (1985): Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by 2-D visual cortical filters. *J. Optical Society of America A* **2**, 1160-.
[31] Daugman, J.G. (1988): Complete discrete 2-D Gabor transforms by neural networks for image analysis and compression. *IEEE Transactions on Acoustics, Speech, and Signal Processing* **36**, 1169-.
[32] Davies, M. & G.W. Humphreys (Eds.) (1993): *Consciousness*. Oxford: Blackwell.
[33] Denschlag, J., *et al.* (2000): Generating solitons by phase engineering of a Bose-Einstein condensate. *Science* **287**, 97-101.
[34] Desimone, R. (1996): Neural mechanisms for visual memory and their role in attention. *Proceedings of the National Academy of Sciences of USA* **93**, 13494-13499.
[35] De Valois, R.L. & G.H. Jacobs (1968): Primate color vision. *Science* **162**, 533-540.
[36] De Yoe, E.A. & D.C. Van Essen (1988): Concurrent processing streams in monkey visual cortex. *Trends in Neurosciences* **11**, 219-226.
[37] Dubois, D.M. (Ed.) (2000a): Proceedings of CASYS'99. *Internat. J. Computing Anticipatory Systems* **5**, **6**, 7. Liege: CHAOS. Especially, in vol. 7: Proceedings of the Symposium "Quantum Neural Information Processing: New Technology? New Biology?" (espec. papers by Marcer, Sutherland, Farre, Mitchell, Dubois; also Citko, Luksza & Sienko).
[38] Dubois, D.M. (Ed.) (2000b): *AIP Conference Proceedings*, vol. **517**: *Computing Anticipatory Systems – CASYS'99* in Liege. Melville (NY): American Institute of Physics (especially papers by Marcer & Schempp, Kirvelis, Sienko, Hoekstra & Rouw, Santoli, Pribram, Dubois, Araujo).
[39] Ebdon, M. (1993): Is the cerebral neocortex an uniform cognitive architecture? *Mind & Language* **8**, 368-398.
[40] Ezhov, A.A. (2001): Pattern recognition with quantum neural networks. *Proceedings of ICAPR '01*, Rio de Janeiro, in press.
[41] Ezhov, A.A., A.V. Nifanova & D. Ventura (2001): Quantum associative memory with distributed queries. *Information Sciences*, in press.

[42] Ezhov, A.A. & D. Ventura (2000): Quantum neural networks. Ch. 11 in: N. Kasabov (Ed.): *Future Directions for Intelligent Systems and Information Sciences* (Series "Studies in Fuzziness and Soft Computing", vol. 45). Heidelberg: Physica-Verlag (Springer), pp. 213-235.
[43] Fedorec, A.M. & P. Marcer (Eds.) (1996): *Living Computers* (symposium proceedings). Dartford: Greenwich Univ. Press.
[44] Flanagan, O. (1992): *Consciousness Reconsidered.* Cambridge (MA): MIT Press.
[45] Frith, C., R. Perry & E. Lumer (1999): The neural correlates of conscious experience: an experimental framework. *Trends in Cognitive Sciences* **3**, 105-114.
[46] Fröhlich, H. (1968): Long-range coherence and energy storage in biological systems. *Internat. J. Quantum Chemistry* **2**, 641-649.
[47] Fuster, J.M. & J.P. Jervey (1982): Neuronal firing in the inferotemporal cortex of the monkey in a visual memory task. *J. Neuroscience* **2**, 361-375.
[48] Gardner, D. (Ed.) (1993): *The Neurobiology of Neural Networks*. Cambridge (MA): MIT Press.
[49] Goldman-Rakić, P.S. (1996): Memory: Recording experience in cells and circuits. *Proceedings of the National Academy of Sciences of USA* **93**, 13435-13437 (organizer's introduction to the symposium with the same title; whole proceedings in the same vol., pp. 13435-13551.).
[50] Goswami, A. (1990): Consciousness in quantum physics and the mind-body problem. *J. Mind & Behavior* **11** (1990) 75-96.
[51] Gray, C.M., P. König, A.K. Engel & W. Singer (1989): Oscillatory responses in cat visual cortex exhibit inter-columnar synchronization which reflects global stimulus properties. *Nature* **338**, 334-337.
[52] Gray, M.S., J.R. Movellan & T.J. Sejnowski (1997): Dynamic features for visual speech-reading: A systematic comparison. *Advances in Neural Information Processing Systems* **9**, 751-757.
[53] Haken, H. (1991): *Synergetic Computers and Cognition (A Top-Down Approach to Neural Nets).* Berlin: Springer.
[54] Haken, H. (1996): *Principles of Brain Functioning.* Berlin, Springer.
[55] Hameroff, S.R. (1994): Quantum coherence in microtubules: a neural basis for emergent consciousness? *J. Consciousness Studies* **1**, 91-118.
[56] Hameroff, S.R. (1998): "Funda-Mentality": Is the conscious mind subtly linked to a basic level of the Universe? *Trends in Cognitive Science* **2**, 119-127.
[57] Hameroff, S.R., A.W. Kaszniak & A.C. Scott (1996): *Towards a Science of Consciousness: Tucson I.* Cambridge (MA): MIT Press.
[58] Hameroff, S.R. & R. Penrose (1995): Orchestrated reduction of quantum coherence in brain microtubules: A model for consciousness. *In:* J. King, K. Pribram (Eds.): *Scale in Conscious Experience: Is the Brain Too Important To Be Left to Specialists to Study?* Mahwah (NJ): Lawrence Erlbaum Assoc., pp. 243-275.
[59] Hameroff, S.R. & R. Penrose (1996): Conscious events as orchestrated space-time selections. *J. Consciousness Studies* **3**, 36-53.
[60] Hariharan, P. (1996): *Optical Holography.* Cambridge: Cambridge Univ. Press.
[61] Harpur, G.F. & R.W. Prager (1996): Development of low entropy coding in a recurrent network. *Network: Computation in Neural Systems* **7**, 277-284.
[62] Hyvärinen, A. & E. Oja (1997): A fast fixed-point algorithm for independent comp. anal. *Neural Computation* **9**, 1483-1492.
[63] Ingber, L. (1998): Statistical mechanics of neocortical interactions: Training and testing canonical momenta indicators of EEG. *Mathematical & Comput. Modelling* **27** (no. 3), 33-64.
[64] Jibu, M., K.H. Pribram & K. Yasue (1996): From conscious experience to memory storage and retrieval: The role of quantum brain dynamics and boson condensation of evanescent photons. *Internat. J. Modern Physics* **10**, 1735-1754.
[65] Jibu, M. & K. Yasue (1995): *Quantum Brain Dynamics and Consciousness.* Amsterdam / Philadelphia: John Benjamins.
[66] Jibu, M. & K. Yasue (1997): Quantum field theory of evanescent photons in brain as quantum theory of consciousness. *Informatica* **21**, 471-490.
[67] Jones, J.A., V. Vedral, A. Ekert & G. Castagnoli (2000): Geometric quantum computation using nuclear magnetic resonance. *Nature Letters* **403**, 869-871.
[68] Kainen, P.C. (1992): Orthogonal dimension and tolerance. Tech. report IM-061592 (Industrial Math., Washington, DC).
[69] Kainen, P.C. & V. Kurková (1993): Quasi-orthogonal dimension of Euclidean spaces. *Applied Mathematics Letters* **6** (no. 3), 7-10.
[70] Kandel, E.R., J.H. Schwartz & T.M. Jessel (1991): *Principles of Neural Science.* London (UK): Prentice Hall Internat., 3rd ed.
[71] Koch, C. (1997): Computation and the single neuron. *Nature* **385**, 207-210.
[72] Koch, C. & I. Segev (2000): The role of single neurons in information processing. *Nature Neurosci. (Suppl.)* **3**, 1171-1177.
[73] Kosslyn, S.M. (1988): Aspects of a cognitive neuroscience of mental imagery. *Science* **240**, 1621-1626.
[74] Kosslyn, S.M. & R.A. Andersen (Eds.) (1992): *Frontiers in Cognitive Neuroscience.* Cambridge (MA): MIT Press.
[75] Körner, E., M.-O. Gewaltig, U. Körner, A. Richter & T. Rodemann (1999): A model of computation in neocortical architecture. *Neural Networks* **12**, 989-1005.
[76] Kurková, V. & P.C. Kainen (1996): A geometric method to obtain error-correcting classification by neural networks with fewer hidden units. In: *Proceedings of Int. Conf. on Neural Networks '96.* Washington (DC): IEEE, pp. 127-132.
[77] Lee, S.-H. & R. Blake (1999a): Visual form created solely from temporal structure. *Science* **284**, 1165-1168.

[78] Lee, S.-H. & R. Blake (1999b): Detection of temporal structure depends on spatial structure. *Vision Research* **39**, 3033-3048.
[79] Lee, T.S. (1996): Image representation using 2-dim. Gabor wavelets. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **18** (no. 10), 1-13.
[80] Lewicki, M.S. & B.A. Olshausen (1999): A probabilistic framework for the adaptation and comparison of image codes. *J. Optical Society of America A: Optics, Image Science, and Vision* **16**, 1587-1601.
[81] Livingstone, M. & D. Hubel (1988): Segregation of form, color, movement, and depth: Anatomy, physiology, and perception. *Science* **240**, 740-749.
[82] Lockwood, M. (1989): *Mind, Brain and the Quantum*. Oxford: Blackwell.
[83] Logothetis, N.K. (1999): Vision: A window on consciousness. *Scientific American*, Novem. 1999, 45-51.
[84] Luccio, R. (1993): Gestalt problems in cognitive psychology: Field theory, invariance, and auto-organisation. *In:* V. Roberto (Ed.): *Advances in Computational Perception.* Heidelberg: Springer.
[85] Luria, A.R. (1983): *Fundamentals of Neuropsychology*. Beograd: Nolit (Serb. transl.).
[86] MacLennan, B.J. (1999): Field computation in natural and artificial intelligence. *Information Sciences* **119**, 73-89.
[87] Majewski, A. (1999): Separable and entangled states of composite quantum systems – rigorous description. *Open Systems and Information Dynamics* **6**, 79-86.
[88] Mannion, C.L.T. & J.G. Taylor (1992): Information processing by oscillating neurons. *In:* J.G. Taylor & C.L.T. Mannion (Eds.): *Coupled Oscillating Neurons.* London: Springer, pp. 98-111.
[89] Marcel, A.J. & E. Bisiach (Eds.) (1988): *Consciousness in Contemporary Science*. Oxford: Clarendon Press.
[90] Marcer, P. & W. Schempp (1997): A model of neuron working by quantum holography. *Informatica* **21**, 517-532.
[91] Marcer, P. & W. Schempp (1998): The brain as a conscious system. *Internat. J. General Systems* **27**, 231-248.
[92] McIntosh, A.R., M.N. Rajah & N.J. Lobaugh (1999): Interactions of prefrontal cortex in relation to awareness in sensory learning. *Science* **284**, 1531-1533.
[93] Mezard, M., G. Parisi & M.A. Virasoro (1987): *Spin Glass Theory and Beyond*. Singapore: World Scientific.
[94] Mishkin, M., L.G. Ungerleider & K.A. Macko (1983): Object vision and spatial vision: Two cortical pathways. *Trends in Neurosciences* **6**, 414-417.
[95] Miyashita, Y. & H.S. Chang (1988): Neuronal correlate of pictorial short-term memory in the primate temporal cortex. *Nature* **331**, 68-70.
[96] Montero, V.M. (2000): Attentional activation of the visual thalamic reticular nucleus depends on 'top-down' inputs from the primary visual cortex via corticogeniculate pathways. *Brain Research* **864**, 95-104.
[97] Moran, J. & R. Desimone (1985): Selective attention gates visual processing in the extrastriate cortex. *Science* **229**, 782-784.
[98] Nanopoulos, D.V. (1995): Tech rep. ACT-08/95 – http://xxx.lanl.gov/abs/hep-ph/9505374
[99] Newman, J. (1997): Toward a general theory of the neural correlates of consciousness. *J. Consciousness Studies* **4**, 47-66 (part I), 100-121 (part II).
[100] Nobili, R. (1985): Schrödinger wave holography and brain cortex. *Physical Review A* **32**, 3618-3626.
[101] Oakley, D.A. (Ed.) (1985): *Brain and Mind.* London / New York: Methuen.
[102] Olshausen, B.A. & D.J. Field (1996a): Natural image statistics and efficient coding. *Network: Computation in Neural Systems* **7**, 333-339.
[103] Olshausen, B. & D. Field (1996b): Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature Letters* **381**, 607-609.
[104] Olshausen, B.A. & D.J. Field (1997): Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research* **37**, 3311-3325.
[105] Perret, D.I. & M.W. Oram (1998): Visual recognition based on temporal cortex cells: Viewer-centered processing of pattern configuration. *Zeitschr. für Naturforschung* **53c**, 518-541.
[106] Peruš, M. (1995b): Synergetic approach to cognition-modeling with neural networks. *In:* K. Sachs-Hombach (Ed.): *Bilder im Geiste.* Amsterdam/Atlanta: Rodopi, pp. 183-194.
[107] Peruš, M. (1996): Neuro-quantum parallelism in mind-brain and computers. *Informatica* **20**, 173-183.
[108] Peruš, M. (1997a): Mind: neural computing plus quantum consciousness. *In:* M. Gams, M. Paprzycki & X. Wu (Eds.): *Mind Versus Computer.* Amsterdam: IOS Press & Ohmsha, pp. 156-170.
[109] Peruš, M. (1997b): System-processual backgrounds of consciousness. *Informatica* **21** (1997) 491-506.
[110] Peruš, M. (1998a): Common mathematical foundations of neural and quantum informatics. *Zeitschr. für angewandte Mathematik & Mechanik* **78**, S1, 23-26.
[111] Peruš, M. (2000a): Tracing quantum background processing in visual cortex. *In:* A.V. Bataev (Ed.): *Proceedings 2nd All-Russian Sci. Conf. "Neuro-informatika 2000"*, Part 1, Moscow: MIFI, pp. 208-226 (also in Russian).
[112] Peruš, M. (2000b): Neural correlates of vision and attention (pp. 17-20) & Neuropsychology of vision (pp. 25-28). *Proceedings 3rd Int. Multi-Conf. "Information Society 2000"* – volume *"Cognitive Science / New Science of Cosnciousness"* (Ed.: I. Kononenko). Ljubljana: Institut Jožef Stefan, pp. 17-20 & 25-28.
[113] Peruš, M. (2000c): Neural networks as a basis for quantum associative networks. *Neural Network World* **10** (no. 6), 1001-1013.
[114] Peruš, M. (2000d): *Vse v odnom, odno vo vsem (Matematičeskie modeli associativnjih neironnjih setei).* St. Petersburg: KARO (originally Slovene book as translated to Russian by A. Ippa).
[115] Peruš, M. (2001): A synthesis of the Pribram holonomic theory of vision with quantum associative nets after pre-processing using ICA and other computational models. *Int. J. Computing Anticipatory Systems* **10**, 352-367.

[116] Peruš, M. & S.K. Dey (2000): Quantum systems can realize content-addressable associative memory. *Applied Mathematics Letters* **13** (no. 8), 31-36.
[117] Peruš, M. & P. Ečimovič (1998): Memory and pattern recognition in associative neural networks. *Internat. J. Applied Science & Computations* **4**, 283-310.
[118] Pessa, E. & G. Vitiello (1999): Quantum dissipation and neural net dynamics. *Bioelectrochemistry & Bioenergetics* **48**, 339-342.
[119] Poggio, T., V. Torre, C. Koch (1985): Computational vision and regularization theory. *Nature* **317**, 314-319.
[120] Porrill, J., J.P. Frisby, W.J. Adams & D. Buckley (1999): Robust and optimal use of information in stereo vision. *Nature Letters* **397** (1999) 63-66.
[121] Pötzsch, M., N. Krüger & C. von der Malsburg (1996): Improving object recognition by transforming Gabor filter responses. *Network: Computation in Neural Systems* **7**, 341-347.
[122] Pribram, K.H. (1971): *Languages of the Brain (Experimental Paradoxes and Principles in Neuropsychology).* Orig.: Englewood Cliffs (NJ): Prentice-Hall; 5th publ.: New York: Brandon House.
[123] Pribram, K.H., M.C. Lassonde & M. Ptito (1981): Classification of receptive field properties in cat visual cortex. *Experimental Brain Research* **43**, 119-130.
[124] Pribram, K.H. & E.H. Carlton (1986): Holonomic brain theory in imaging and object perception. *Acta Psychologica* **63**, 175-210.
[125] Pribram, K.H. (1991): *Brain and Perception (Holonomy and Structure in Figural Processing).* Hillsdale (NJ): Lawrence Erlbaum Associates.
[126] Pribram, K.H. (Ed.) (1993): *Rethinking Neural Networks: Quantum Fields and Biological Data.* Hillsdale (NJ): Lawrence Erlbaum Associates.
[127] Pribram, K.H. (1995): Brain and perception: From Köhler's fields to Gabor's quanta of information. *Proceedings of the 39th Congress of German Society for Psychology*, pp. 53-69.
[128] Pribram, K.H. (1997a): What is mind that the brain may order it? *Proceedings of Symposia in Applied Mathematics* **52**, 301-329 (Vol. 2: Proceed. of the Norbert Wiener Centenary Congress, 1994, eds. V. Mandrekar & P.R. Masani; Providence: Am. Math. Soc.).
[129] Pribram, K.H. (1997b): The deep and surface structure of memory and conscious learning: Toward a 21st century model. *In*: R.L. Solso (Ed.): *Mind and Brain Sciences in the 21st Century*. Cambridge (MA): MIT Press, pp. 127-156.
[130] Pribram, K.H. (1998a): Brain and the composition of conscious experience. *J. Consciousness Studies* **6**, no. 5, 19-42.
[131] Pribram, K.H. (1998b): Status report: Quantum holography and the brain. *In:* Proceedings of the ECHO conference, Helsinki. (Or see: Quantum holography: Is it relevant to brain function? *Information Sciences* **115** (1999) 97-102.)
[132] Psaltis, D., D. Brady., X.-G. Gu & S. Lin (1990): Holography in artificial neural networks. *Nature* **343**, 325-330.
[133] Pylkkänen, P. & Pyllkö P. (Eds.) (1995): *New Directions in Cognitive Science.* Helsinki: Finnish AI Soc. (especially articles by Chrisley, Globus, Gould, Hiley, Peruš, Revonsuo).
[134] Rabitz, H., R. de Vivie-Riedle, M. Motzkus & K. Kompa (2000): Whither the future of controlling quantum phenomena? *Science* **288**, 824-828.
[135] Rainer, G. & E.K. Miller (2000): Effects of visual experience on the representation of objects in the prefrontal cortex. *Neuron* **27**, 179-189.
[136] Rakić, Lj., G. Kostopoulos, D. Raković & Dj. Koruga (Eds.) (1997): *Brain and Consciousness — Proceedings of the First ECPD Int. Symposium* (vol. I) & *Workshop* (vol. II) *on Scientific Bases of Consciousness '97.* Belgrade: ECPD.
[137] Riesenhuber, M. & T. Poggio (2000): Models of object recognition. *Nature Neurosci. (Suppl.)* **3**, 1199-1204.
[138] Roelfsema, P.R. (1998): Solutions for the binding problem. *Zeitschr. für Naturforschung* **53c**, 691-715.
[139] Schempp, W. (1993): Bohr's indeterminacy principle in quantum holography, self-adaptive neural network architectures, cortical self-organization, molecular computers, magnetic resonance imaging and solitonic nanotechnology. *Nanobiology* **2**, 109-164.
[140] Schempp, W. (1994): Analog VLSI network models, cortical linking neural network models, and quantum holographic neural technology. *In:* J.S. Byrnes *et al.* (Eds.): *Wavelets and Their Applications.* Amsterdam: Kluwer, pp. 213-260.
[141] Schempp, W. (1995): Phase coherent wavelets, Fourier transform resonance imaging, and synchronized time-domain neural networks. *Procedings of the Steklov Institute of Mathematics* **3**, 323-351.
[142] Schiller, P.H. & N.K. Logothetis (1990): The color-opponent and broad-band channels of the primate visual system. *Trends in Neurosciences* **13**, 392-398.
[143] Snider, G., *et al.* (1999): Quantum-dot cellular automata: Review and recent experiments. *J. Applied Physics* **85**, 4283-4285.
[144] Sompolinsky, H. & M. Tsodyks (1994): Segmentation by a network of oscillators with stored memories. *Neural Computation* **6**, 642-657.
[145] Spencer, R.G. (2001): Bipolar spectral associative memories. *IEEE Transactions on Neural Networks,* to appear in May no.
[146] Stapp, H.P. (1993): *Mind, Brain and Quantum Mechanics.* Berlin: Springer.
[147] Stillings, N., S. Weisler, C. Chase, M. Feinstein, J. Garfield & E. Rissland (1995): *Cognitive Science.* Cambridge (MA): MIT Press. Ch. 12: Vision.
[148] Sutherland, J.G. (1990): A holographic model of memory, learning and expression. *Internat. J. Neural Systems* **1**, 259-267.
[149] Tootell, R.B., N.K. Hadjikhani, J.D. Mendola, S. Marrett & A.M. Dale (1998): From retinotopy to recognition: fMRI in human visual cortex. *Trends in Cognitive Sciences* **2**, 174-183.
[150] van Hateren, J.H. (1992): Real and optimal images in early vision. *Nature* **360**, 68-70.

[151] van Hateren, J.H. & D.L. Ruderman (1998): Independent component analysis of natural image sequences yields spatiotemporal filters similar to simple cells in primary visual cortex. *Proceedings of the Royal Society of London B* (Biol. Sci.) **265**, no. 1412, 2315-2320.
[152] van Hateren, J.H. & A. van der Schaaf (1998): Independent component filters of natural images compared with simple cells in primary visual cortex. *Proceedings of the Royal Society of London B* **265**, 359-366.
[153] Vidyasagar, T.R. (1999): A neuronal model of attentional spotlight: parietal guiding the temporal. *Brain Research Review* **30**, 66-76.
[154] Wainwright, M. J. (1999): Visual adaptation as optimal information transmission. *Vision Research* **39**, 3960-3974.
[155] Wallis, G. & H. Bülthoff (1999): Learning to recognize objects. *Trends in Cognitive Sciences* **3**, 22-31.
[156] Wang, D.L. (1999): Object selection based on oscillatory correlation. *Neural Networks* **12**, 579-592.
[157] Wang, P.P., *et al.* (Eds.) (1998): *Proceedings of the 4th Joint Conference on Information Sciences '98.* Research Triangle Park (NC, USA): Assoc. Intellig. Machinery. *Volume II*: Proceedings of the 3rd Internat. Conf. on Computatational Intelligence & Neuroscience (Ed. G. Georgiou); sections on neuro-quantum information processing: pp. 167-224.
[158] Weinacht, T.C., J. Ahn & P.H. Bucksbaum (1999): Controlling the shape of a quantum wavefunction. *Nature Letters* **397**, 233-235. (See also pp. 207-208 of the same vol.: Quantum control: Sculpting a wavepacket; by W. Schleich).
[159] Weliky, M., L.C. Katz (1999): Correlational structure of spontaneous neuronal activity in the developing lateral geniculate nucleus in vivo. *Science* **285**, 599-604.
[160] Wheeler, J.A. & W.H. Zurek (Eds.) (1983): *Quantum Theory and Measurement.* Princeton (NJ): Princeton Univ. Press.
[161] Wurtz, R.H., M.E. Goldberg & D.L. Robinson (1980): Behavioral modulation of visual responses in the monkey: Stimulus selection for attention and movement. *Progress in Psychobiology and Physiological Psychology* **9**, 43-83.
[162] Zak, M. & C.P. Williams (1998): Quantum neural nets. *Internat. J. Theoretical Physics* **37**, 651-684.
[163] Zornetzer, S.F., J.L. Davis & C. Lau (1990): *An Introduction to Neural and Electronic Networks.* San Diego: Academic Press.

Call for Papers

ADBIS'2002

Sixth East-European Conference on Advances in Databases and Information Systems

September 8-11, 2002, Bratislava, Slovakia

http://www.dcs.elf.stuba.sk/adbis2002

In co-operation with the Moscow ACM SIGMOD Chapter, Slovak Society for Computer Science, Faculty of Electrical Engineering and Information Technology STU Bratislava.
Co-operation with ACM SIGMOD is also envisaged.

AIMS AND SCOPE

The main objective of the ADBIS series of conferences is to provide a forum for the dissemination of research accomplishments and promote interaction and collaboration between the Database and Information Systems research communities from Central and East European countries and the rest of the world. The ADBIS conferences provide an international platform for the presentation of research on database theory, development of advanced DBMS technologies, and their advanced applications.

The Conference continues the ADBIS conferences held in St. Petersburg (1997), Poznan (1998), Maribor (1999), Prague (2000), and Vilnius (2001). The Conference will consist of regular sessions with technical contributions reviewed and selected by an international program committee, as well as of invited talks and tutorials given by leading scientists. The official language of the Conference will be English.

TOPICS

Original papers dealing with both theory and/or applications of database technology and information systems are solicited. The areas of interest include, but are not limited to, the following:

- database theory,
- data modeling and database design,
- physical database design and performance evaluation,
- database systems architectures,
- activity modelling, advanced transaction, and workflow management,
- advanced databases (object-oriented DB, web-based DB, multimedia DB, temporal and spatial DB, deductive and active DB, etc.),
- advanced information systems (GIS, intelligent IS, component-based IS, etc.),
- advanced database applications,
- heterogeneous databases interoperability and mediation,
- database and knowledge-base management systems and technology,
- text management,
- data mining, data warehousing, and knowledge discovery,
- data quality,
- XML and databases,
- e-business and e-commerce,
- web-based and distributed information systems,
- enterprise information systems,
- mobile computing and agents,
- information systems and software systems engineering,
- information system security.

SUBMISSION

We solicit contributions of the following form:

- full research papers describing research accomplishment (approximately 5000 words),
- short research papers that report interesting results and do not justify a full paper (2000 - 3000 words),
- communications, i.e. experience reports, surveys, project overviews, etc. which do not fully adhere to the standards of a firs rate scientific publication, but are nevertheless of interest and value for the participants of the conference,
- proposals for tutorials and panels.

Research papers should be original contributions, not accepted or submitted elsewhere. Research papers will be published in th LNCS series of Springer Verlag. Communications will be included in additional local proceedings.

There is an intention to publish a selection of outstanding papers in an internationally recognised journal (a subject of additiona review process).

We can only accept electronic submissions in Postscript, PDF, or RTF format. Already at the submission stage, authors ar encouraged to consider the final paper format requirements, as specified at http:// www.dcs.elf.stuba.sk /adbis2002/format/.

Additionally, for each paper send the first page in ASCII (text-format) containing title, authors, affiliation, contact information abstract and keywords.

Papers shall be submitted to: adbis@dcs.elf.stuba.sk

AWARDS

The best paper authored solely by students will be awarded. Please indicate in your submission, whether it is a student paper.

TIMETABLE

Submission of abstracts:	February 25, 2002
Paper submission:	March 4, 2002
Notification of acceptance:	April 30, 2002
Camera-ready papers:	May 28, 2002
Conference:	September 8-11, 2002

CONFERENCE ORGANISATION

General Chair:
Ludovit Molnar, Rector of the Slovak University of Technology in Bratislava, Slovakia

Program Committee Co-Chairs:
Yannis Manolopoulos, Department of Informatics, Aristotle University of Thessaloniki, Greece, manolopo@delab.csd.auth.gr
Pavol Navrat, Slovak University of Technology in Bratislava, Slovakia, navrat@elf.stuba.sk

Program Committee*:

Leopoldo Bertossi	Matthias Klusch	Guenther Pernul	Aphrodite Tsalgatidou
Omran A. Bukhres	Mikhail Kogalovsky	Jaroslav Pokorny	Vladimir Vojtek
Albertas Caplinskas	Karol Matiasko	Henrikas Pranevichius	Gottfried Vossen
Wojciech Cellary	Mihhail Matskin	Colette Rolland	Benkt Wangler
Bogdan Czejdo	Tomaz Mohoric	George Samaras	Tatjana Welzer
Johann Eder	Tadeusz Morzy	Klaus-Dieter Schewe	Viacheslav Wolfengagen
Heinz Frank	Nikolay Nikitchenko	Joachim W. Schmidt	Vladimir Zadorozhny
Remigijus Gustas	Boris Novikov	Timothy K. Shih	Alexander Zamulin
Tomas Hruska	Maria Orlowska	Myra Spiliopoulou	
Leonid Kalinichenko	Euthimios Panagos	Julius Stuller	
Wolfgang Klas	Oscar Pastor	Bernhard Thalheim	

* *PC is expected to be extended with additional members*

Organising Committee Chair:
Maria Bielikova, Slovak University of Technology (Slovakia), bielikova@dcs.elf.stuba.sk

ADBIS Steering Committee Chair:
Leonid Kalinichenko, Russian Academy of Science (Russia), leonidk@synth.ipi.ac.ru

ADBIS Steering Committee:

Andras Benczur (Hungary)	Mirjana Ivanovic (Yugoslavia)	Boris Novikov (Russia)
Radu Bercaru (Romania)	Mikhail Kogalovsky (Russia)	Jaroslav Pokorny (Czech Republic)
Albertas Caplinskas (Lithuania)	Yannis Manolopoulos (Greece)	Boris Rachev (Bulgaria)
Johann Eder (Austria)	Rainer Manthey (Germany)	Anatoly Stogny (Ukraine)
Janis Eiduks (Latvia)	Tadeusz Morzy (Poland)	Tatjana Welzer (Slovenia)
Hele-Mai Haav (Estonia)	Pavol Navrat (Slovakia)	Viacheslav Wolfengagen (Russia)

CONFERENCE VENUE

The Conference will be held in Bratislava, Slovakia, at the congress centre Družba located about 3 km west from the historica centre of Bratislava and reachable by regular tram and bus connection in about 10 minutes from the city centre.

Bratislava, the capital and the largest city in Slovakia, is an old central European city that has always gained attention through it location on the river Danube, its nearness to Vienna and Budapest, and the hospitality of its inhabitants. Throughout the history Bratislava has grown not only to the administrative, industrial, cultural and scientific centre of Slovakia but first of all to the lovel city worth of staying in. Today's visitors, sitting in restful cafés or walking through the streets and squares in immediate vicinity t the St. Martin gothic cathedral, the Town Hall, Primate's Palace, or other historical buildings, can fully enjoy and admire th ancient architecture of Bratislava's old city. The picturesque region around Bratislava, known for large vineyards and winemakin districts and for original ceramic manufacturing, impresses a visitor with hundreds of years of handicrafts, culture and traditions i this neighbourhood.

CONTACT ADDRESS

Pavol Navrat
Department of Computer Science and Engineering
Slovak University of Technology
Ilkovicova 3, 812 19 Bratislava
Slovakia

E-mail: adbis@dcs.elf.stuba.sk
navrat@elf.stuba.sk
Phone: (+421 2) 654 29 502
(+421 2) 602 91 548
Fax: (+421 2) 654 20 587

JOŽEF STEFAN INSTITUTE

Jožef Stefan (1835-1893) was one of the most prominent physicists of the 19th century. Born to Slovene parents, he obtained his Ph.D. at Vienna University, where he was later Director of the Physics Institute, Vice-President of the Vienna Academy of Sciences and a member of several scientific institutions in Europe. Stefan explored many areas in hydrodynamics, optics, acoustics, electricity, magnetism and the kinetic theory of gases. Among other things, he originated the law that the total radiation from a black body is proportional to the 4th power of its absolute temperature, known as the Stefan–Boltzmann law.

The Jožef Stefan Institute (JSI) is the leading independent scientific research institution in Slovenia, covering a broad spectrum of fundamental and applied research in the fields of physics, chemistry and biochemistry, electronics and information science, nuclear science technology, energy research and environmental science.

The Jožef Stefan Institute (JSI) is a research organisation for pure and applied research in the natural sciences and technology. Both are closely interconnected in research departments composed of different task teams. Emphasis in basic research is given to the development and education of young scientists, while applied research and development serve for the transfer of advanced knowledge, contributing to the development of the national economy and society in general.

At present the Institute, with a total of about 700 staff, has 500 researchers, about 250 of whom are postgraduates, over 200 of whom have doctorates (Ph.D.), and around 150 of whom have permanent professorships or temporary teaching assignments at the Universities.

In view of its activities and status, the JSI plays the role of a national institute, complementing the role of the universities and bridging the gap between basic science and applications.

Research at the JSI includes the following major fields: physics; chemistry; electronics, informatics and computer sciences; biochemistry; ecology; reactor technology; applied mathematics. Most of the activities are more or less closely connected to information sciences, in particular computer sciences, artificial intelligence, language and speech technologies, computer-aided design, computer architectures, biocybernetics and robotics, computer automation and control, professional electronics, digital communications and networks, and applied mathematics.

The Institute is located in Ljubljana, the capital of the independent state of **Slove**nia (or S♡nia). The capital today is considered a crossroad between East, West and Mediterranean Europe, offering excellent productive capabilities and solid business opportunities, with strong international connections. Ljubljana is connected to important centers such as Prague, Budapest, Vienna, Zagreb, Milan, Rome, Monaco, Nice, Bern and Munich, all within a radius of 600 km.

In the last year on the site of the Jožef Stefan Institute, the Technology park "Ljubljana" has been proposed as part of the national strategy for technological development to foster synergies between research and industry, to promote joint ventures between university bodies, research institutes and innovative industry, to act as an incubator for high-tech initiatives and to accelerate the development cycle of innovative products.

At the present time, part of the Institute is being reorganized into several high-tech units supported by and connected within the Technology park at the Jožef Stefan Institute, established as the beginning of a regional Technology park "Ljubljana". The project is being developed at a particularly historical moment, characterized by the process of state reorganisation, privatisation and private initiative. The national Technology Park will take the form of a shareholding company and will host an independent venture-capital institution.

The promoters and operational entities of the project are the Republic of Slovenia, Ministry of Science and Technology and the Jožef Stefan Institute. The framework of the operation also includes the University of Ljubljana, the National Institute of Chemistry, the Institute for Electronics and Vacuum Technology and the Institute for Materials and Construction Research among others. In addition, the project is supported by the Ministry of Economic Relations and Development, the National Chamber of Economy and the City of Ljubljana.

Jožef Stefan Institute
Jamova 39, 1000 Ljubljana, Slovenia
Tel.:+386 1 4773 900, Fax.:+386 1 219 385
Tlx.:31 296 JOSTIN SI
WWW: http://www.ijs.si
E-mail: matjaz.gams@ijs.si
Contact person for the Park: Iztok Lesjak, M.Sc.
Public relations: Natalija Polenec

CONTENTS OF *Informatica* Volume 25 (2001) pp. 1–595

Papers

ATKINSON, C. & T. KÜHNE. 2001. Stratified frameworks. Informatica 25:393–402.

BALANTIČ, Z. & M. BERNIK. 2001. Multimedia supported study of achieving high worker's efficiency in relation to his work. Informatica 25:371–374.

BARNETT, M. & W. SCHULTE. 2001. The ABCs of specification: asml, behavior, and components. Informatica 25:517–526.

BAVEC, C. 2001. Modelling of management decision making .... Informatica 25:375–380.

BEŽEK, A. & M. GAMS. 2001. An agent that understands job description. Informatica 25:99–105.

BINZ, E. & W. SCHEMPP. 2001. Information technology: The Lie groups definig the filter banks of the compact disc. Informatica 25:279–291.

BOSILJ-VUKSIC, V, & V. HLUPIC. 2001. Petri nets and IDEF diagrams: Applicability and efficasy for business process modelling. Informatica 25:123–133.

BRETT, A.C., T. MIYAMOTO & J.F. KESS. 2001. A nested combinatorial-states model of sentence processing. Informatica 25:107–122.

ARANDO, P. 2001. The need for speed: A practitioner's view of rapid application development in e-business. Informatica 25:555–564.

ČIZMAN, A. 2001. Computerized logistics information system—a key to competitiveness. Informatica 25:89–98.

COY, W. & U. PIRR. 2001. Change in learning and teaching. Informatica 25:149–153.
DESNOS, J.-F. 2001. A data warehouse for French universities. Informatica 25:177–181.

DIETRICH, S.W., S.D. URBAN, A. SUNDERMIER, Y. NA, Y. JIN & S. KAMBHAMPATI. 2001. A language and framework for supporting an active approach to component-based software integration. Informatica 25:443–454.

FEUSTEL, B., A. KÁRPÁTI, T. RACK & T.C. SCHMIDT. 2001. An environment for processing compound media streams. Informatica 25:201–209.

GALÁN M.J., F. GARCIA, L. ÁLVAREZ, A. OCÓN & E. RUBIO. 2001. 'Beowulf cluster' for high-performance computing tasks at the university: A very profitable investment. Informatica 25:189–193.

GALÁN MORILLO, F.J., V. DIAZ & J.M. CAÑETE VALDEÓN. 2001. Towards a rigorous and effective functional contract for components. Informatica 25:527–532.

GAMS, M. & M. BOHANEC. 2001. Intelligent systems applications. Informatica 25:387–392.

GREENBERG, J. 2001. A hybrid system for delivering web based distance learning and teaching material. Informatica 25:155–158.

GUIZANI, M. & M. ANAN. 2001. Fault-tolerant ATM switching architectures based on MINs: A survey. Informatica 25:69–81.

HAATAJA, A., J. SUHONEN & E. SUTINEN. 2001. How to learn introductory programming over the web? Informatica 25:165–171.

HUTCHINSON, D. & M.J. WARREN. 2001. Security authentication for on-line Internet banking. Informatica 25:349–356.

HYBERTSON, D. 2001. A uniform component modeling space. Informatica 25:475–482.

INDIHAR ŠTEMBERGER, M. & J. GRAD. 2001. The object model of splines. Informatica 25:135–142.

JI, K. & S. CHEN. 2001. DEPA (Design Pattern Application)—a component-based model for applying design patterns in software development. Informatica 25:455–463.

KANTARDZIC, M.M., P. FAGUY, A. GOLDBERG & T.E. HOWE. 2001. Artificial neural network approach and parameter estimation in experimental spectroscopy. Informatica 25:19–26.

KIM, H.-K. & R.Y. LEE. 2001. Management process for supporting the component development. Informatica 25:565–573.

KLENAK, S. & S. BAUK. 2001. Modeling shipping company information systems. Informatica 25:431–438.

KUITTINEN, M., E. SUTINEN, H. TOPI & M. TURPEINEN 2001. Learning by experience: Networks in learning organizations. Informatica 25:159–164.

KUMAR M.V.N., A., A.K. SINGH & R. BABU S. 2001. A security assurance framework for component based software development. Informatica 25:509–515.

LEE, C.-I. & C.-J. TSAI. 2001. An efficient approach to extracting and ranking the top K interesting target ranks from Web search e. Informatica 25:329–339.

LEIGH, W., M. PAZ, N. PAZ & R. PURVIS. 2001. A pattern recognition approach to the prediction of price increases in the New York Stock Exchange Composite Index. Informatica 25:261–269.

LINDEN, M., J. KANNER & M. KIVILOMPOLO. 2001. FEIDHE—integrating PKI in Finish higher education. Informatica 25:211–216.

LINKEVICH, A.D. 2001. Neural fields: An approach to infinite-dimensional systems for information processing. Informatica 25:235–246.

LOKE, S.W. 2001. An overview of mobile agents in distributed applications: Possibilities for future enterprise systems. Informatica 25:247–260.

MAGOULAS, G.D., K.A. PAPANIKOLAOU & M. GRIGORIADOU. 2001. Neuro-Fuzzy synergism for planning the content in a web-based course. Informatica 25:39–48.

MAHNIČ, V. & I. ROŽANC. 2001. Data quality: A prerequisite for succesful data warehouse implementation. Informatica 25:183–188.

MAIER, K.D., V. GLAUCHE, R. BLICKHAN & C. BECKSTEIN. 2001. Control of a one-legged three-dimensional hopping movement system with multi-layer-perceptron neural networks. Informatica 25:27–38.

MAIER, K.D., C. BECKSTEIN, R. BLICKHAN, D. FEY & W. ERHARD. 2001. A digital multi-layer-perception hardware architecture based on three-dimensional masively parallel opto-electronic circuits. Informatica 25:271–278.

MIKL, U. 2001. Electronic formation of a contract. Informatica 25:381–386.

MIZUNO, K., S. NISHIHARA, H. KANOH & I. KISHI. 2001. Population migration: A meta-heuristics for stochastic approaches to constraint satisfaction problems. Informatica 25:421–430.

MLADENIĆ, D., W.F. EDDY & S. ZIOLKO. 2001. Data mining of baskets collected at different locations over one year. Informatica 25:365–372.

MOISAN, S., A. RESSOUCHE & J.-P. RIGAULT. 2001. Blocks, a component framework with checking facilities for knowledge-based systems. Informatica 25:501–507.

MOYLE, S. & A. SRINIVASAN. 2001. Classificatory challenge-data mining: A recipe. Informatica 25:343–348.

NAEGELE-JACKSON, S., U. HILGERS & P. HOLLECZEK. 2001. Evaluation of codec behavior in IP and ATM networks. Informatica 25:195–200.

NOVAK, B. 2001. Soft computing on small data sets. Informatica 25:83–88.

OHLSSON, M.C. 2001. Evolution of fault-prone components in legacy systems: A case study. Informatica 25:545–553.

PAPRZYCKI, M., A. COSTEINES, W. DOUGLAS, P. GATLING, R. NIESS & L. SCARDINO. 2001. Applying neural networks to practical problems—methodological considerations. Informatica 25:11–17.

PARALIČ, J., M. PARALIČ & M. MACH. 2001. Support of knowledge management in distributed environment. Informatica 25:319–328.

PARHAMI, B. 2001. Approach to component based synthesis of fault tolerant software. Informatica 25:533–543.

PERUŠ, M. 2001. Image processing and becoming conscious of its results. Informatica 25575:–592.

RESLER, R.D. & J.M. BOYLE. 2001. Register allocation: A program-algebraic approach. Informatica 25:223–233.

RYJÁČEK, Z., J. RICHLÍK & P. JIROUŠEK. 2001. Information system supporting CATS. Informatica 25:173–176.

SATOH, I. 2001. MobiDoc: A mobile agent-based framework for compound documents. Informatica 25:493–500.

SIKICI, A. & N.Y. TOPALOGLU. 2001. Towards software design automation with patterns. Informatica 25:349–356.

ŠKRJANC, M., M. GROBELNIK & D. ZUPANIČ. 2001. Insights offered by data-mining when analyzing media space data. Informatica 25:357–364.

SOLOJENTSEV, E.D. & V.V. KARASEV. 2001. Risk logical and probabilistic models in business and identification of risk models. Informatica 25:49–55.

SOUNDARAJAN, N. & S. FRIDELLA. 2001. Understanding OO frameworks and applications: An increment approach. Informatica 25:297–308.

STRACHAN, A., T. SHAW & D. ADAMS. 2001. Information system delivery in a tiered security environment. Informatica 25:217–222.

TADEUSIEWICZ, R. & P. LULA. 2001. Neural network analysis of time series data. Informatica 25:3–10.

URBANČIČ, T. 2001. Learning and understanding human skill. Informatica 25:–.
ZHANG, S. & C. ZHANG. 2001. A model for compressing prababilities in belief networks. Informatica 25:409–420.

VITELA, J.E., U.R. HANEBUTTE & J.L. GORDILLO. 2001. Performance analysis of a parallel neural network training code for control of dynamic systems. Informatica 25:57–67.

WEINREICH, R. & R. PLÖSCH. 2001. An agent-based component platform for dynamically adaptable distributed environments. Informatica 25:483–491.

XIE, X. & S.M. SHATZ. 2001. An approach for modeling components with customization for distributed software. Informatica 25:465–474.

Editorials

ANDERSON, P.G., G. KLEIN, E. OJA, N.C. STEEL, G. ANTONIOU, V. MLADENOV & M. PAPRZYCKI. 2001. Introduction: Neural networks and their applications. Informatica 25:1.
JURIČ, M.B., I. ROZMAN & D. DEUGO. 2001. Component based software development. Informatica 25:441–442.

KNOP, J. & V. MAHNIČ. 2001. . Informatica 25:147–148.

INTRODUCTION. 2001. Informatica 25:295–296.

INTRODUCTION. 2001. Informatica 25:341–342.

Calls for Papers

Call for a forum discussing informational consciousness on the Internet. 2001. Informatica 25:143.

Informational society 2001. Infos, Cankarjev dom, Ljubljana, Slovenia. 22–26 October 2001. Informatica 25:144,293.

ADBIS 2002. Sixth East-European Conference on Advances in Databases and Information Systems. 2001. . Informatica 25:593–594.

Professional Societies

Jožef Stefan Institute. Ljubljana, Slovenia. 2001. Informatica 25:145,294,439,595.

INFORMATICA

AN INTERNATIONAL JOURNAL OF COMPUTING AND INFORMATICS

INVITATION, COOPERATION

Submissions and Refereeing

Please submit three copies of the manuscript with good copies of the figures and photographs to one of the editors from the Editorial Board or to the Contact Person. At least two referees outside the author's country will examine it, and they are invited to make as many remarks as possible directly on the manuscript, from typing errors to global philosophical disagreements. The chosen editor will send the author copies with remarks. If the paper is accepted, the editor will also send copies to the Contact Person. The Executive Board will inform the author that the paper has been accepted, in which case it will be published within one year of receipt of e-mails with the text in Informatica LaTeX format and figures in .eps format. The original figures can also be sent on separate sheets. Style and examples of papers can be obtained by e-mail from the Contact Person or from FTP or WWW (see the last page of Informatica).

Opinions, news, calls for conferences, calls for papers, etc. should be sent directly to the Contact Person.

QUESTIONNAIRE

☐ Send Informatica free of charge

☐ Yes, we subscribe

Please, complete the order form and send it to Dr. Rudi Murn, Informatica, Institut Jožef Stefan, Jamova 39, 1111 Ljubljana, Slovenia.

Since 1977, Informatica has been a major Slovenian scientific journal of computing and informatics, including telecommunications, automation and other related areas. In its 16th year (more than five years ago) it became truly international, although it still remains connected to Central Europe. The basic aim of Informatica is to impose intellectual values (science, engineering) in a distributed organisation.

Informatica is a journal primarily covering the European computer science and informatics community - scientific and educational as well as technical, commercial and industrial. Its basic aim is to enhance communications between different European structures on the basis of equal rights and international refereeing. It publishes scientific papers accepted by at least two referees outside the author's country. In addition, it contains information about conferences, opinions, critical examinations of existing publications and news. Finally, major practical achievements and innovations in the computer and information industry are presented through commercial publications as well as through independent evaluations.

Editing and refereeing are distributed. Each editor can conduct the refereeing process by appointing two new referees or referees from the Board of Referees or Editorial Board. Referees should not be from the author's country. If new referees are appointed, their names will appear in the Refereeing Board.

Informatica is free of charge for major scientific, educational and governmental institutions. Others should subscribe (see the last page of Informatica).

ORDER FORM – INFORMATICA

Name: .....

Title and Profession (optional): .....

.....

Home Address and Telephone (optional): .....

.....

Office Address and Telephone (optional): .....

.....

E-mail Address (optional): .....

Signature and Date: .....

Informatica WWW:

http://ai.ijs.si/informatica/
http://orca.st.usm.edu/informatica/

Referees:

Witold Abramowicz, David Abramson, Adel Adi, Kenneth Aizawa, Suad Alagić, Mohamad Alam, Dia Ali, Alan Aliu, Richard Amoroso, John Anderson, Hans-Jurgen Appelrath, Iván Araujo, Vladimir Bajič, Michel Barbeau, Grzegorz Bartoszewicz, Catriel Beeri, Daniel Beech, Fevzi Belli, Simon Beloglavec, Sondes Bennasri, Francesco Bergadano, Istvan Berkeley, Azer Bestavros, Andraž Bežek, Balaji Bharadwaj, Ralph Bisland, Jacek Blazewicz, Laszlo Boeszoermenyi, Damjan Bojadžijev, Jeff Bone, Ivan Bratko, Pavel Brazdil, Bostjan Brumen, Jerzy Brzezinski, Marian Bubak, Davide Bugali, Troy Bull, Leslie Burkholder, Frada Burstein, Wojciech Buszkowski, Rajkumar Bvyya, Netiva Caftori, Particia Carando, Robert Cattral, Jason Ceddia, Ryszard Choras, Wojciech Cellary, Wojciech Chybowski, Andrzej Ciepièlewski, Vic Ciesielski, Mel Ó Cinnéide, David Cliff, Maria Cobb, Jean-Pierre Corriveau, Travis Craig, Noel Craske, Matthew Crocker, Tadeusz Czachorski, Milan Češka, Honghua Dai, Deborah Dent, Andrej Dobnikar, Sait Dogru, Peter Dolog, Georg Dorfner, Ludoslaw Drelichowski, Matija Drobnič, Maciej Drozdowski, Marek Druzdzel, Jozo Dujmović, Pavol Ďuriš, Amnon Eden, Johann Eder, Hesham El-Rewini, Darrell Ferguson, Warren Fergusson, David Flater, Pierre Flener, Wojciech Fliegner, Vladimir A. Fomichov, Terrence Forgarty, Hans Fraaije, Hugo de Garis, Eugeniusz Gatnar, Grant Gayed, James Geller, Michael Georgiopolus, Jan Goliński, Janusz Gorski, Georg Gottlob, David Green, Herbert Groiss, Jozsef Gyorkos, Marten Haglind, Abdelwahab Hamou-Lhadj, Inman Harvey, Marjan Hericko, Elke Hochmueller, Jack Hodges, Doug Howe, Rod Howell, Tomáš Hruška, Don Huch, Alexey Ippa, Hannu Jaakkola, Ryszard Jakubowski, Piotr Jedrzejowicz, A. Milton Jenkins, Eric Johnson, Polina Jordanova, Djani Juričič, Marko Juvancic, Sabhash Kak, Li-Shan Kang, Ivan Kapustøk, Orlando Karam, Roland Kaschek, Jacek Kierzenka, Jan Kniat, Stavros Kokkotos, Fabio Kon, Kevin Korb, Gilad Koren, Andrej Krajnc, Henryk Krawczyk, Ben Kroese, Zbyszko Krolikowski, Benjamin Kuipers, Matjaž Kukar, Aarre Laakso, Ivan Lah, Phil Laplante, Bud Lawson, Ulrike Leopold-Wildburger, Timothy C. Lethbridge, Joseph Y-T. Leung, Barry Levine, Xuefeng Li, Alexander Linkevich, Raymond Lister, Doug Locke, Peter Lockeman, Matija Lokar, Jason Lowder, Kim Teng Lua, Ann Macintosh, Bernardo Magnini, Andrzej Małachowski, Peter Marcer, Andrzej Marciniak, Witold Marciszewski, Vladimir Marik, Jacek Martinek, Tomasz Maruszewski, Florian Matthes, Daniel Memmi, Timothy Menzies, Dieter Merkl, Zbigniew Michalewicz, Gautam Mitra, Roland Mittermeir, Madhav Moganti, Reinhard Moller, Tadeusz Morzy, Daniel Mossé, John Mueller, Hari Narayanan, Jerzy Nawrocki, Rance Necaise, Elzbieta Niedzielska, Marian Niedq'zwiedziński, Jaroslav Nieplocha, Oscar Nierstrasz, Roumen Nikolov, Mark Nissen, Jerzy Nogieć, Stefano Nolfi, Franc Novak, Antoni Nowakowski, Adam Nowicki, Tadeusz Nowicki, Hubert Österle, Wojciech Olejniczak, Jerzy Olszewski, Cherry Owen, Mieczyslaw Owoc, Tadeusz Pankowski, Jens Penberg, William C. Perkins, Warren Persons, Mitja Peruš, Stephen Pike, Niki Pissinou, Aleksander Pivk, Ullin Place, Gabika Polčicová, Gustav Pomberger, James Pomykalski, Dimithu Prasanna, Gary Preckshot, Dejan Rakovič, Cveta Razdevšek Pučko, Ke Qiu, Michael Quinn, Gerald Quirchmayer, Vojislav D. Radonjic, Luc de Raedt, Ewaryst Rafajlowicz, Sita Ramakrishnan, Wolf Rauch, Peter Rechenberg, Felix Redmill, James Edward Ries, David Robertson, Marko Robnik, Colette Rolland, Wilhelm Rossak, Ingrid Russel, A.S.M. Sajeev, Kimmo Salmenjoki, Bo Sanden, P. G. Sarang, Vivek Sarin, Iztok Savnik, Ichiro Satoh, Walter Schempp, Wolfgang Schreiner, Guenter Schmidt, Heinz Schmidt, Dennis Sewer, Zhongzhi Shi, Mária Smolárová, Carine Souveyet, William Spears, Hartmut Stadtler, Olivero Stock, Janusz Stokłosa, Przemyslaw Stpiczyński, Andrej Stritar, Maciej Stroinski, Tomasz Szmuc, Zdzislaw Szyjewski, Jure Šilc, Metod Škarja, Jiří Šlechta, Chew Lim Tan, Zahir Tari, Jurij Tasič, Gheorge Tecuci, Piotr Teczynski, Stephanie Teufel, Ken Tindell, A Min Tjoa, Vladimir Tosic, Wieslaw Traczyk, Roman Trobec, Marek Tudruj, Andrej Ule, Amjad Umar, Andrzej Urbanski, Marko Uršič, Tadeusz Usowicz, Romana Vajde Horvat, Elisabeth Valentine, Kanonkluk Vanapipat, Alexander P. Vazhenin, Zygmunt Vetulani, Olivier de Vel, Valentino Vranić, Eugene Wallingford, John Weckert, Michael Weiss, Tatjana Welzer, Lee White, Gerhard Widmer, Stefan Wrobel, Stanislaw Wrycza, Janusz Zalewski, Damir Zazula, Yanchun Zhang, Ales Zivkovic, Zonling Zhou, Robert Zorc, Anton P. Železnikar

INFORMATICA

AN INTERNATIONAL JOURNAL OF COMPUTING AND INFORMATICS

INVITATION, COOPERATION

Submissions and Refereeing

Please submit three copies of the manuscript with good copies of the figures and photographs to one of the editors from the Editorial Board or to the Contact Person. At least two referees outside the author's country will examine it, and they are invited to make as many remarks as possible directly on the manuscript, from typing errors to global philosophical disagreements. The chosen editor will send the author copies with remarks. If the paper is accepted, the editor will also send copies to the Contact Person. The Executive Board will inform the author that the paper has been accepted, in which case it will be published within one year of receipt of e-mails with the text in Informatica LaTeX format and figures in .eps format. The original figures can also be sent on separate sheets. Style and examples of papers can be obtained by e-mail from the Contact Person or from FTP or WWW (see the last page of Informatica).

Opinions, news, calls for conferences, calls for papers, etc. should be sent directly to the Contact Person.

QUESTIONNAIRE

☐ Send Informatica free of charge

☐ Yes, we subscribe

Please, complete the order form and send it to Dr. Rudi Murn, Informatica, Institut Jožef Stefan, Jamova 39, 1111 Ljubljana, Slovenia.

Since 1977, Informatica has been a major Slovenian scientific journal of computing and informatics, including telecommunications, automation and other related areas. In its 16th year (more than five years ago) it became truly international, although it still remains connected to Central Europe. The basic aim of Informatica is to impose intellectual values (science, engineering) in a distributed organisation.

Informatica is a journal primarily covering the European computer science and informatics community - scientific and educational as well as technical, commercial and industrial. Its basic aim is to enhance communications between different European structures on the basis of equal rights and international refereeing. It publishes scientific papers accepted by at least two referees outside the author's country. In addition, it contains information about conferences, opinions, critical examinations of existing publications and news. Finally, major practical achievements and innovations in the computer and information industry are presented through commercial publications as well as through independent evaluations.

Editing and refereeing are distributed. Each editor can conduct the refereeing process by appointing two new referees or referees from the Board of Referees or Editorial Board. Referees should not be from the author's country. If new referees are appointed, their names will appear in the Refereeing Board.

Informatica is free of charge for major scientific, educational and governmental institutions. Others should subscribe (see the last page of Informatica).

ORDER FORM – INFORMATICA

Name: .....

Title and Profession (optional): .....

.....

Home Address and Telephone (optional): .....

.....

Office Address and Telephone (optional): .....

.....

E-mail Address (optional): .....

Signature and Date: .....

Informatica WWW:

http://ai.ijs.si/informatica/
http://orca.st.usm.edu/informatica/

Referees:

Witold Abramowicz, David Abramson, Adel Adi, Kenneth Aizawa, Suad Alagić, Mohamad Alam, Dia Ali, Alan Aliu, Richard Amoroso, John Anderson, Hans-Jurgen Appelrath, Iván Araujo, Vladimir Bajič, Michel Barbeau, Grzegorz Bartoszewicz, Catriel Beeri, Daniel Beech, Fevzi Belli, Simon Beloglavec, Sondes Bennasri, Francesco Bergadano, Istvan Berkeley, Azer Bestavros, Andraž Bežek, Balaji Bharadwaj, Ralph Bisland, Jacek Blazewicz, Laszlo Boeszoermenyi, Damjan Bojadžijev, Jeff Bone, Ivan Bratko, Pavel Brazdil, Bostjan Brumen, Jerzy Brzezinski, Marian Bubak, Davide Bugali, Troy Bull, Leslie Burkholder, Frada Burstein, Wojciech Buszkowski, Rajkumar Bvyya, Netiva Caftori, Particia Carando, Robert Cattral, Jason Ceddia, Ryszard Choras, Wojciech Cellary, Wojciech Chybowski, Andrzej Ciepielewski, Vic Ciesielski, Mel Ó Cinnéide, David Cliff, Maria Cobb, Jean-Pierre Corriveau, Travis Craig, Noel Craske, Matthew Crocker, Tadeusz Czachorski, Milan Češka, Honghua Dai, Deborah Dent, Andrej Dobnikar, Sait Dogru, Peter Dolog, Georg Dorfner, Ludoslaw Drelichowski, Matija Drobnič, Maciej Drozdowski, Marek Druzdzel, Jozo Dujmović, Pavol Ďuriš, Amnon Eden, Johann Eder, Hesham El-Rewini, Darrell Ferguson, Warren Fergusson, David Flater, Pierre Flener, Wojciech Fliegner, Vladimir A. Fomichov, Terrence Forgarty, Hans Fraaije, Hugo de Garis, Eugeniusz Gatnar, Grant Gayed, James Geller, Michael Georgiopolus, Jan Goliński, Janusz Gorski, Georg Gottlob, David Green, Herbert Groiss, Jozsef Gyorkos, Marten Haglind, Abdelwahab Hamou-Lhadj, Inman Harvey, Marjan Hericko, Elke Hochmueller, Jack Hodges, Doug Howe, Rod Howell, Tomáš Hruška, Don Huch, Alexey Ippa, Hannu Jaakkola, Ryszard Jakubowski, Piotr Jedrzejowicz, A. Milton Jenkins, Eric Johnson, Polina Jordanova, Djani Juričič, Marko Juvancic, Sabhash Kak, Li-Shan Kang, Ivan Kapustøk, Orlando Karam, Roland Kaschek, Jacek Kierzenka, Jan Kniat, Stavros Kokkotos, Fabio Kon, Kevin Korb, Gilad Koren, Andrej Krajnc, Henryk Krawczyk, Ben Kroese, Zbyszko Krolikowski, Benjamin Kuipers, Matjaž Kukar, Aarre Laakso, Ivan Lah, Phil Laplante, Bud Lawson, Ulrike Leopold-Wildburger, Timothy C. Lethbridge, Joseph Y-T. Leung, Barry Levine, Xuefeng Li, Alexander Linkevich, Raymond Lister, Doug Locke, Peter Lockeman, Matija Lokar, Jason Lowder, Kim Teng Lua, Ann Macintosh, Bernardo Magnini, Andrzej Małachowski, Peter Marcer, Andrzej Marciniak, Witold Marciszewski, Vladimir Marik, Jacek Martinek, Tomasz Maruszewski, Florian Matthes, Daniel Memmi, Timothy Menzies, Dieter Merkl, Zbigniew Michalewicz, Gautam Mitra, Roland Mittermeir, Madhav Moganti, Reinhard Moller, Tadeusz Morzy, Daniel Mossé, John Mueller, Hari Narayanan, Jerzy Nawrocki, Rance Necaise, Elzbieta Niedzielska, Marian Niedq'zwiedziński, Jaroslav Nieplocha, Oscar Nierstrasz, Roumen Nikolov, Mark Nissen, Jerzy Nogieć, Stefano Nolfi, Franc Novak, Antoni Nowakowski, Adam Nowicki, Tadeusz Nowicki, Hubert Österle, Wojciech Olejniczak, Jerzy Olszewski, Cherry Owen, Mieczyslaw Owoc, Tadeusz Pankowski, Jens Penberg, William C. Perkins, Warren Persons, Mitja Peruš, Stephen Pike, Niki Pissinou, Aleksander Pivk, Ullin Place, Gabika Polčicová, Gustav Pomberger, James Pomykalski, Dimithu Prasanna, Gary Preckshot, Dejan Rakovič, Cveta Razdevšek Pučko, Ke Qiu, Michael Quinn, Gerald Quirchmayer, Vojislav D. Radonjic, Luc de Raedt, Ewaryst Rafajlowicz, Sita Ramakrishnan, Wolf Rauch, Peter Rechenberg, Felix Redmill, James Edward Ries, David Robertson, Marko Robnik, Colette Rolland, Wilhelm Rossak, Ingrid Russel, A.S.M. Sajeev, Kimmo Salmenjoki, Bo Sanden, P. G. Sarang, Vivek Sarin, Iztok Savnik, Ichiro Satoh, Walter Schempp, Wolfgang Schreiner, Guenter Schmidt, Heinz Schmidt, Dennis Sewer, Zhongzhi Shi, Mária Smolárová, Carine Souveyet, William Spears, Hartmut Stadtler, Olivero Stock, Janusz Stokłosa, Przemyslaw Stpiczyński, Andrej Stritar, Maciej Stroinski, Tomasz Szmuc, Zdzislaw Szyjewski, Jure Šilc, Metod Škarja, Jiří Šlechta, Chew Lim Tan, Zahir Tari, Jurij Tasič, Gheorge Tecuci, Piotr Teczynski, Stephanie Teufel, Ken Tindell, A Min Tjoa, Vladimir Tosic, Wieslaw Traczyk, Roman Trobec, Marek Tudruj, Andrej Ule, Amjad Umar, Andrzej Urbanski, Marko Uršič, Tadeusz Usowicz, Romana Vajde Horvat, Elisabeth Valentine, Kanonkluk Vanapipat, Alexander P. Vazhenin, Zygmunt Vetulani, Olivier de Vel, Valentino Vranić, Eugene Wallingford, John Weckert, Michael Weiss, Tatjana Welzer, Lee White, Gerhard Widmer, Stefan Wrobel, Stanislaw Wrycza, Janusz Zalewski, Damir Zazula, Yanchun Zhang, Ales Zivkovic, Zonling Zhou, Robert Zorc, Anton P. Železnikar

EDITORIAL BOARDS, PUBLISHING COUNCIL

Informatica is a journal primarily covering the European computer science and informatics community; scientific and educational as well as technical, commercial and industrial. Its basic aim is to enhance communications between different European structures on the basis of equal rights and international refereeing. It publishes scientific papers accepted by at least two referees outside the author's country. In addition, it contains information about conferences, opinions, critical examinations of existing publications and news. Finally, major practical achievements and innovations in the computer and information industry are presented through commercial publications as well as through independent evaluations.

Editing and refereeing are distributed. Each editor from the Editorial Board can conduct the refereeing process by appointing two new referees or referees from the Board of Referees or Editorial Board. Referees should not be from the author's country. If new referees are appointed, their names will appear in the list of referees. Each paper bears the name of the editor who appointed the referees. Each editor can propose new members for the Editorial Board or referees. Editors and referees inactive for a longer period can be automatically replaced. Changes in the Editorial Board are confirmed by the Executive Editors.

The coordination necessary is made through the Executive Editors who examine the reviews, sort the accepted articles and maintain appropriate international distribution. The Executive Board is appointed by the Society Informatika. Informatica is partially supported by the Slovenian Ministry of Science and Technology.

Each author is guaranteed to receive the reviews of his article. When accepted, publication in Informatica is guaranteed in less than one year after the Executive Editors receive the corrected version of the article.

Volume 25 Number 4 November 2001 ISSN 0350-5596

Informatica

An International Journal of Computing and Informatics

Introduction		441
A Language and Framework for Supporting an Active Approach to Component-Based Software Integration	S.W. Dietrich, S.D. Urban, A. Sundermier, Y. Na, Y. Jin, S. Kambhampati	443
DEPA (Design Pattern Application) - A Component-based Model for Applying Design Patterns in Software Development	K. Ji, S. Chen	455
An Approach for Modeling Components with Customization for Distributed Software	X. Xie, S.M. Shatz	465
A Uniform Component Modeling Space	D. Hybertson	475
An Agent-Based Component Platform for Dynamically Adaptable Distributed Environments	R. Weinreich, R. Plösch	483
MobiDoc: A Mobile Agent-based Framework for Compound Documents	I. Satoh	493
Blocks, a Component Framework with Checking Facilities for Knowledge-Based Systems	S. Moisan, A. Ressouche, J.-P. Rigault	501
A Security Assurance Framework for Component Based Software Development	A.M.V.N. Kumar, A.K. Singh, R.S.Babu	509
The ABCs of Specification: AsmL, Behavior, and Components	M. Barnett, W. Schulte	517
Towards a Rigorous and Effective Functional Contract for Components	F.J.G. Morillo, V. Diaz, J.M.C. Valdeón	527
Approach to Component Based Synthesis of Fault Tolerant Software	B. Parhami	533
Evolution of Fault-Prone Components in Legacy Systems: A Case Study	M.C. Ohlsson	545
The Need for Speed: A Practitioner's View of Rapid Application Development in eBusiness	P. Carando	555
Management Process for Supporting the Component Development	H.-K. Kim, R.Y. Lee	565
Image processing and becoming conscious of its result	M. Peruš	575
Reports and Announcements		593