Vrije Universiteit Brussel – Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes – France

2003

ANALYSIS AND IMPLEMENTATION OF ASYNCHRONOUS COMPONENT MODEL

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Kaiye Xu

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoters: Jean-Claude Royer (Ecole des Mines de Nantes)

ACKONWLEDGEMENTS

Thanks first to all the people for their all kinds of help to make me finish my master thesis successfully. Thank you very much.

I want to say thanks to my advisor Jean-Claude Royer who gave me a lot of useful instructions and good suggestions on my thesis development during all my work in 6 months. The regular meeting which I had with Jean-Claude Royer once a week, and also more usual communication by email is the guarantee and indicator to bring my thesis to the goal.

I am also grateful to all the staffs of the Information Department of EMN, who provided good support and help to let me concentrate myself on the master thesis to get the best product. Especially thanks to Object Group, which gave me chance to listen to the different voices related to my work.

Dr.Hong Zheng is the lady I am willing to appreciate since she introduced me to EMOOSE to have a great time of my master study.

I would like to thank all my classmates and friends I met in EMN and in France, care and help given by them, especially the great party which made my abroad life so colorful and wonderful.

The final but special thanksgiving is devoted to my dear parents and my brother, for their strong encouragement and assistance to accompany me when it is my first time leaving them so far.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Component-based development is the popular approach in nowaday IT industry. It gives software developer a highly modular way to integrate all the necessary components as a whole system of the specialized application. Furthermore, component could be reused for a lot of different situation since every component is designed as an independent unit with its own function. Thus this reduces much time and money on software production, and makes the emphasis about software quality more on the system design in the software life cycle. How to get an elegant and efficient component model is a core issue in the methodology to build component-based software. Meanwhile, today is Internet time. All the application must be migrated or developed from local machine to the network. One important fact of network is that it need the component to work in distributed, concurrent and heterogeneous environments. Obviously synchronous and asynchronous communication glues all the components together to obtain some intended function. Correspondingly the component model is divided into asynchronous model and synchronous model. However asynchronous model get less attention on its analysis and implementation than synchronous one in the past. According to the new programming language ProActive using asynchronous communication, my master thesis is trying to give some rules to translate the asynchronous component model to its implementation of ProActive. Moreover I give one common algorithm to check the message buffer size in asynchronous model represented by ATAG. At last I'd like to discuss some interesting questions about some future work on component and asynchronous model.

KEY WORDS
Component, Architecture, Asynchronous, RMI, ProActive, MDA, ATAG, Mailbox

Introduction

Looking back on the software history, software development was usually faced to the certain issue in a given domain. This situation led to the specialized software system where each subsystem is designed, coded and tested only for the prescribed function of that system. In other words, it is development-specific. However, today since computer has swiftly changed our human life through an incredible speed and power, people become more and more greedy on what the new software should bring to the world. This raising need for software function in turn make the size and complexity of software exponentially increase. Additionally software is intended to be versatile as much as possible to make lazy life for people. Therefore to build fashionable and flexible software seems to need much more money and time than ever. On the other hand, large software usually means low productivity and high cost which hinder the IT industry going forward. Fortunately people already found some ways to overcome this bottle-neck, and the most classic solution is reusability. Reusability simply means developing once, using more than once in terms of application or problem domain.

In order to improve software reuse in the development, there must be a change in the way we construct the software system with reuse. Obviously integration is the answer to the question, and new software should have a mechanism to integrate the developed system instead of reproducing it by programmer. Component-based software engineering is the corresponding paradigm to describe this kind software process. Here component is the unit of reuse. A software system could be considered as a composition of existing components. In this case, the design of the whole system architecture, or say how to glue all the relative components, is much more important than the design of individual concrete component. UML and ADL is being used to address the problem of modeling component-based software, but both usually can't give the precise semantic description on the software architecture. Particularly, UML and ADL can't correctly feature inter-component communication that is quite dominant and difficult in the software design. In addition, nowadays all the software must be associated to the network. Parallel, concurrent, distributed application becomes the mainstream on network, where asynchronous communication is natural and common. According to these problems, more and more requirements are given out to find a good solution for component-base model with robust communication, not only on synchronous way but also on asynchronous way.

My thesis firstly is to find the proper implementation to the asynchronous component model. The architecture designer not only focuses on the abstract function of each component, but also on the integration and communication protocol among the different components. There must be a gap between the abstract asynchronous component model and concrete software implementation. Meanwhile, although many programming language like Smalltalk and Java, has built-in synchronization support, there are few languages like ProActive, which take care of asynchronization of message sending. This makes the implementation of asynchronous model more difficult on the programming level.

This paper tries to give some general rules to transform asynchronous component model to ProActive. It shows how ProActive does the asynchronous call on top of java RMI layer. All the transformation rules have UML-like style, in the MDA (Model Driven Architecture) way. The general diagram of asynchronous component model may be mapped to the ProActive diagram as the specification of the implementation. According to the transformation rules the paper also provides a simple example about the flight reservation system, and introduce some corresponding source code for asynchronous communication featured by ProActive.

The second research of this master thesis proposes an approach to analyze the asynchronous component model, mainly about its asynchronous communication and related component state. The asynchronous component model is represented by ATAG (Asynchronous Abstract Graphical data Type), which use abstract graphics with some predefined notions to illustrate the component and behavior. ATAG is based on the previous study of GAT (Abstract Graphical data Type) and Korrigan which describes the general component model with symbolic transition system (STS) and algebraic specification. Because the ATAG model could be easily translated into a set of rules with simple data structure, it is clear to make specific algorithm to automatically check the quality problems of component model with asynchronous communication.

Although there are a lot of important analysis for the asynchronous component model, this thesis restricts the work on the component message buffer, so-called mailbox in this paper. In fact mailbox is a classic way to simulate the asynchronous call between components to store the message temporarily for the future use. The question whether the mailbox is bound or not need to be solved to avoid message overflow. This paper gives an algorithm to estimate the mailbox of each component in the ATAG component model. The algorithm not only includes the bound question, but also the max number of each message that a component would receive, and the message cycle that leads to the unbound mailbox. The algorithm is built beyond depth-first search in ATAG component model. Meanwhile in order to realize this algorithm and test it in java, the corresponding parser of ATAG is also made to receive the correct input from ATAG file. The analysis tool and parser is integrated into a java GUI to make the analysis of mailbox in a more uniform and structured way.

Since the work of this master thesis only consider few parts of asynchronous component model, in the future some other analysis will be done to it to get more confidence in software design.

Structure of the Dissertation

Section one explains the main characteristics of software component, and also gives the short description of other relative aspect.

Section two shows the general information of component model, especially on its communication mode: synchronous communication and asynchronous communication, and gives the main difference between both.

Section three describes the programming language ProActive for asynchronous component model, and then lists the transformation rules to bridge between the asynchronous component model and ProActive, and finally uses a simple example to give more insight of ProActive from the implementation view.

Section four presents the ATAG to represent the pure asynchronous component model, and demonstrates the bound algorithm that is used to analyze the mailbox of component in ATAG model.

Section five concludes the thesis work, and describes some related research to improve the analysis of asynchronous component model, and also some other methods which could be used here.

CHAPTER ONE

COMPONENT IN SOFTWARE DEVELOPMENT

Although component was born to help developer reuse previous software, the idea of software reuse actually has been throughout the long history of the software development. Old form of reuse is upgraded with the progress of relative aspect of software or hardware development, for example programming language and standardized I/O interface. Thus component should be considered as the state-of-the-art version of software reuse. Many people get some confusions between the previous way of reuse and the component, so I would like to introduce some main steps in software reuse, and then explain what component is.

1.1 Software Reuse Before Component

Subroutine

The most original thing for reuse is the subroutine in the procedure programming, for example the early Fortran in 1950s [24]. The reuse of subroutine is on the source code level, so programmer can write some code to call a shared method or function in the same source file or another one. All the source code is compiled and linked to get the executable software.

Although this kind of software reuse looks very straightforward, it is limited only for programming. The scope of subroutine is in a source file. Usually the subroutine has its method name as the single interface, sometimes with a few parameters. Furthermore, subroutine has the explicit dependency on the programming language and operation system, and thus the development of subroutine should be designed carefully in a uniform environment. In addition, the test and maintenance of subroutine is difficult, because programmer should use other program in the same system to debug and examine the function of subroutine. The subroutine can’t be changed or overwritten solely, because it is embedded in the executable file. The user must replace the corresponding file with the upgraded version even if the change only happens in a subroutine.

Function Module

Since there could be a lot of subroutines in the software system, the better way to get reuse is apparently to group all the relative or similar subroutines as structured program. Thus the second mode of software reuse is function module, which is a separated unit composed of a set of shared functions; in other words, it could be called reusable function library. This idea was adopted in 1960s [26], and the classic example is programming in C language born at 1971.

The scope of function module is in a particular application of software system [14]. For example, the date/time function module could give the current date/time in various formats, or change the system date/time if necessary. Although the function module is kept alone from other programs of software, it usually needs to be compiled with other program in the software system and linked together. Actually each function module must be available locally for the method call from the other part the system. Therefore function module has the implicit dependency on the specific technology and operation system. Since function module is constructed only for a family of functions, the developer could easily create an initial software architecture composed of raw function modules. Then he can refine the architecture gradually by designing and analyzing each underlying function module. According to that software architecture, the developer could decide whether to reuse the existing function module or to implement it by hand. Because the function module is formed by the traditional subroutines, the testing of it is same as subroutines. The programmer has to run other program to examine the function module. The maintenance of function module sometimes may be simplified to just replace the function module with the new one, with no change to the rest of system.

Class

The powerful object-oriented programming paradigm brings the new software reuse to the world. That is class, as we know. A class is "*a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind*" [29]. Thus the class could be thought as object factory to create new object including predefined variables and methods. Meanwhile class represents a concept of the real world, and object is just the instance of the concept. Undoubtedly, class is implemented by OO language, like Smalltalk [27] since 1970s [26].

The scope of class is in a particular problem domain [14], and developer can produce class hierarchy by inheritance to precisely describe the abstraction of the problem and solution. Because class belongs to the object-oriented category, it needs the relative aspect to support its implementation and deployment, such as language and class loader. Thus class still has the implicit dependency on the specific technology and operation system. Moreover, the class could also be grouped in class library like function module. Often we can reuse class with the same design pattern to solve the commonly occurring problem.

As object-oriented development is different from the function-based development, the object-oriented analysis and design is adopted to decide the class reuse. That's to say, according to the problem statement people tries to get the object model of the problem domain. Then the object model of problem is cast to the solution domain to find the good architecture of software system. The objects in the analysis and design lead to corresponding class, which could reuse the previous class of same object. Since class is the encapsulation to the relative information, it is logically independent unit in the object-oriented paradigm. The test and maintenance of class is also convenient with the OO programming language like C++ and Java [29]. The developer could make the testing object to automatically examine the object of target class, and maintenance may work in similar way.

Middleware

As object-oriented paradigm changed a lot on the software development, the network and Internet play a more surprising role in the whole computer science. Consequently network produces the new progress on software reuse, called Middleware [30]. Here Middleware is referred to a set of reusable and expandable services above the transport layer of TCP/IP [31], and below the application level i.e. API layer. These services generally are used to finish the common functions in operating system and network environment, such as message conversion or network addressing. The Middleware is built on class, and some middleware technologies such as COM have appeared since late 1980s [23].

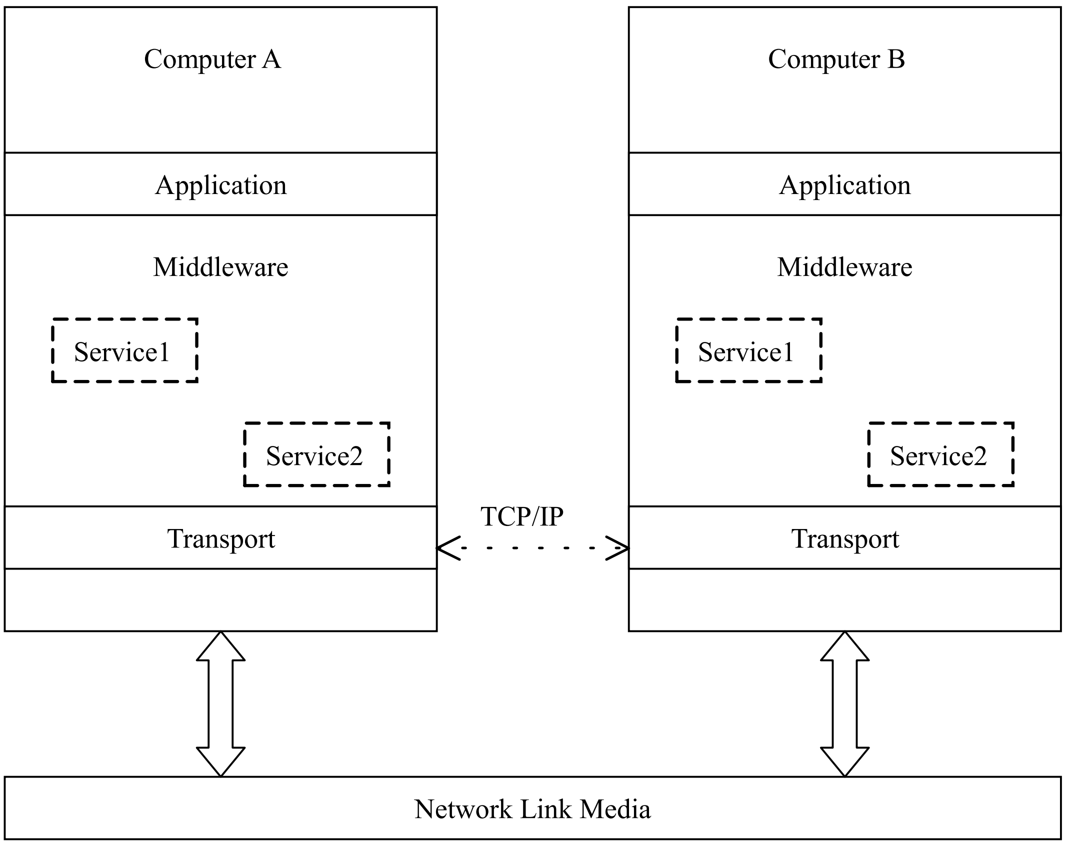


Figure 1: Middleware in Software Reuse

Since all the Middleware are developed in object-oriented style, the basic unit to reuse in Middleware is class. On the other hand, Middleware focuses on the common services for the application, so it emphasize on the quality of service by some class library more than concrete class and object. Like class, the scope of middleware is in the particular domain [14], where each service of middleware targets a smaller field. Meanwhile, the middleware is based on the class reuse in the object-oriented development, and thus it is still proper to apply the object-oriented approach to build the middleware. However, compared to the class reuse, the Middleware is not much restricted to the special technology and operating system. The Middleware is at the higher abstraction level than class. It may contain all kinds of services developed by the different OO tools in the heterogeneous situations, as long as the service is compatible to the run-time environment (operation system and network) and the front-end application.

1.2 Component and its Property and Rules

As the Internet is linking all kinds of application under various environments all over the world, the computing pattern is being changed from centralization to distribution. One large system often contains a lot of sub-systems, which could be cross-domain or cross-platform. Meanwhile, the system developer likes to put more intelligence and automata to the software reuse. It means to obtain the software reusability to a deeper and wider extent to build modern software system upon legacy code and experience. Thus component was born based on previous study of software reuse to realize the new requirements as expected.

Although the definition of component has multiple versions depending on the domain, purpose and individual, there is some agreement of the essence of component from most of the paper about component. Software component is the unit to be implemented for the composition to build a functioning system [13, 14, 16, 18, 20]. Software component could be considered as the deployment of its own interfaces [14, 16, 18]. Generally, the software component exposes its interfaces as the contract to the other things that want to use component. The component gives the explicit context dependency to enable it finish its work in the collaboration style. A component could be independently developed, deployed and updated [13, 14, 16, 18, 20], but it need to be subject to the composition by the third party [14, 18]. In other words, system developers need to make the component coincident to the particular component model or with the particular component platform.

Because the component is based on the previous experience about software reuse, it inherits the information encapsulation of the object-oriented technology since this schema has been proved to have many advantages for development. Therefore from the external view the component is graphically represented as some kind of box with clear interface specification. As reader can see in Figure 2, there is a component named Person to process the personal information. The component Person doesn't give the insight of its internal implementation and state. The client only knows the interface of Person component like IGetName and ISetName, which is the available service by Person component. As a matter of fact, this kind of hiding information is the same as the class, but software component is bigger and higher than what class usually means in the software architecture.

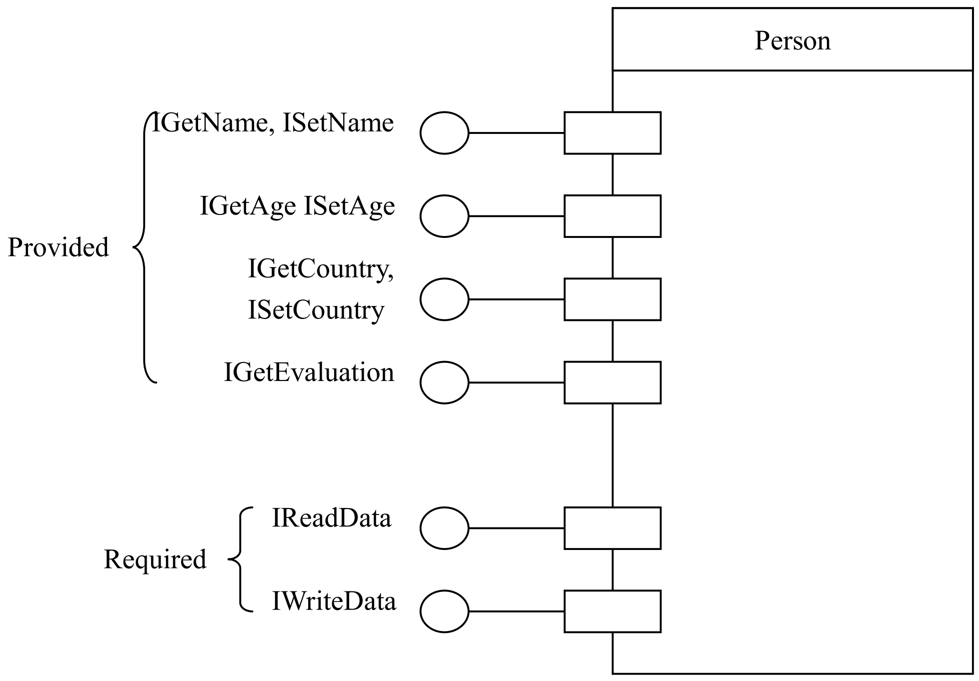


Figure 2: Person Component

Figure 2 also explains the composition of component interface. There are two kinds of services defined by the interface in terms of method caller. One is provided service, which means what the client could get if it calls the component. For example, a client could get the age of one person as provided service through interface IGetAge of component Person. Another is required service, which means what one component need when it calls other component. For example, the interface IReadData of component Person explains one required service that is called by Person to the other component to read the personal information from some database or file.

Although Figure 2 shows the same notion for both kinds of interface, people are trying to explicitly distinguish them in the formal representation and development. For example, there have already been some component languages to separately define the provided service and required service of one component

Using the Person component, the system developer could integrate it to some place in the software system as long as the functions of Person component, or more precisely its interfaces, comply with the design requirement. The following figure shows a component system with Person component, and the dash arrow link means the message sending for the service.

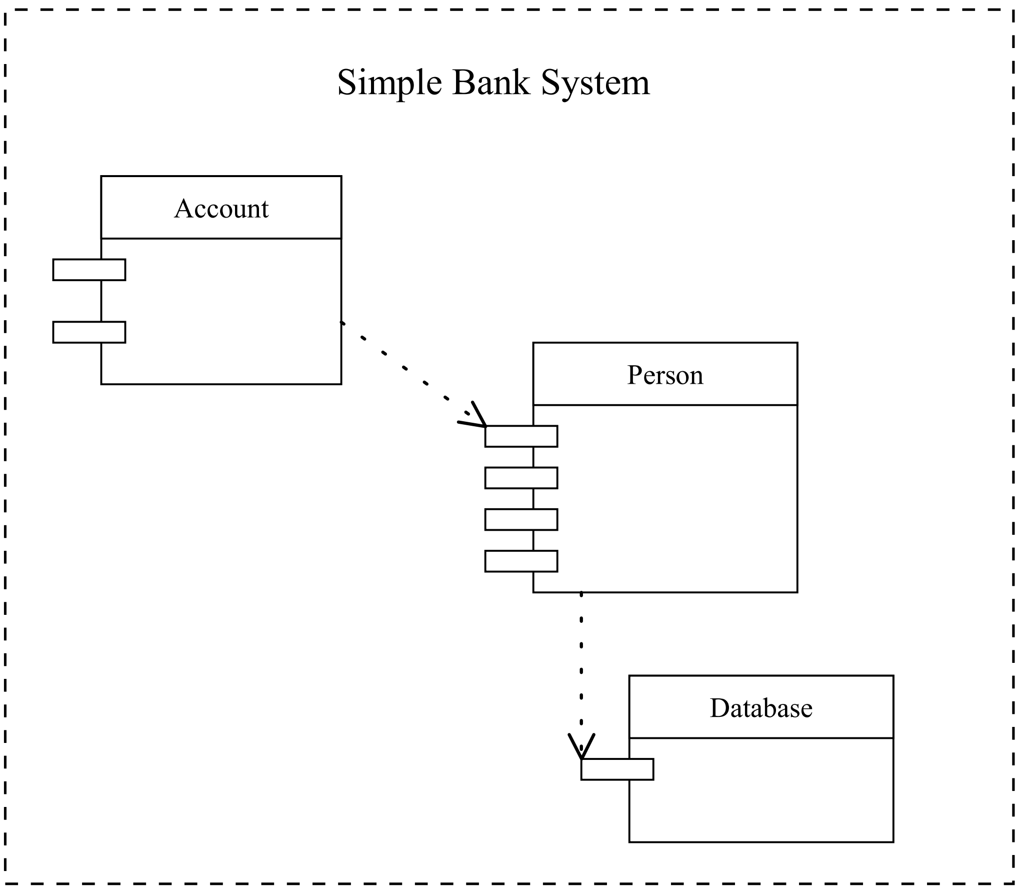


Figure 3: Simple Bank System with Components

The Figure 3 denotes a bank software system with three components: Account, Person and Database. The Account component has the dependency on the Person component as the arrow pointing to the interface of Person component. Meanwhile Person component need the function of Database component, no matter the database of the bank system is SQL server or Oracle, and so on. Therefore, Person component could be reused in the similar context of different application or domain, for example, the computer sales system.

Moreover, the interface and explicit context dependency of Person could be reused by other component, for instance, of flower information process. The component Flower could be represented as in Figure 4, and it has completely same interfaces as Person. Here the interface IGetAge and ISetAge of Flower component could means the life span of the flower since its sprout, and the semantics of other interfaces of Flower component is easy to figure out. Thus people only need to change the some implementation of Person component to realize the new component Flower.

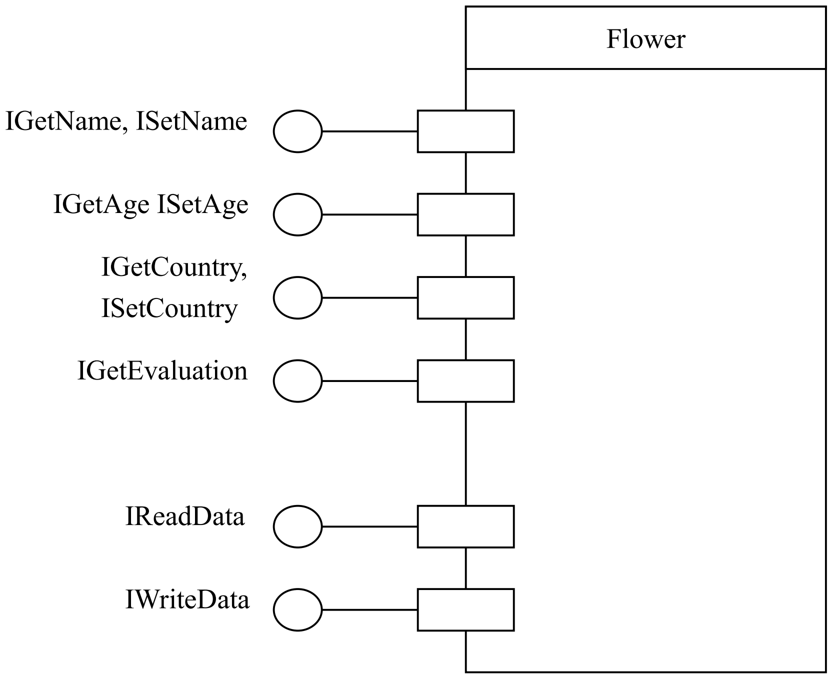


Figure 4: Flower Component

In addition, the component as the software unit is requested to be dynamic and mobile in the distributed, loosely-located and concurrent environment. This means that component could adapt itself to the change in its work situation, and it could be transferred by itself or the system.

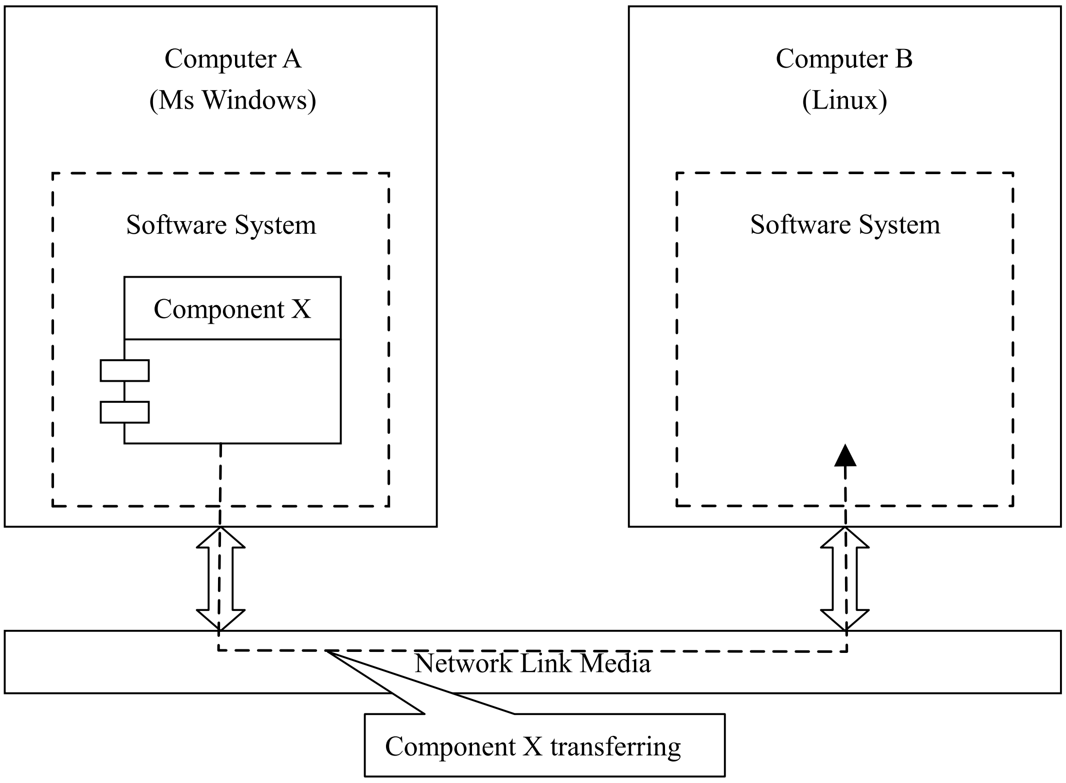


Figure 5: Mobile Component in Network

Figure 5 shows that the component X travels from computer A to computer B through the network media. Therefore, network makes the software reuse in a more autonomous way, especially for the large-scale distributed application.

From the description above, component is much more powerful than any previous form for reusable software. A component could be adopted in any domain if it follows a certain component model and its interfaces satisfy the function requirement [14]. The component should be independent from the specific technology and running environment as much as possible. The only dependency of component is the explicit requirement of relative support in the context since it would call the other software (element) to complete the work together. Moreover, the component is the basic reuse unit in the component-based system development [14].

Although usually the class currently is the implementation unit for almost all software components as the result of main commercial programming language, there is no restriction about what could be included to build the component. In other words, the component function is more important than the concrete realization. One can develop the prior Person component by C++ language, and another man may use this component in the web application with other component of Java classes.

The relative analysis and design is involved in the specific engineering process, so called Component-Based Software Engineering (CBSE) [13, 17, 18]. CBSE is introduced in section 2. The object-oriented analysis and design could be regarded as the sub-activity of CBSE.

Since the component is for composition to create some system, a composition of the some relative components may become a large component to be delivered to create more complex system. Internal components of compound component are invisible from outside, which follows the principle of information hiding. Therefore the software component also has the problem of granularity that would be the balance tradeoff between the flexibility and efficiency.

1.2.1 Characteristics in Component Category

There is not well-acknowledged formal specification of software component. According to the research and experience of component-based software development, some important characteristics have been extracted from all kinds of the component description. The main component characteristics should be the basis of uniform framework for component design and implementation.

The following listed characteristics of component is based on the discussion from a EMN component research group I have participated [32]:

Purpose

Give the purpose of the component, whether it is general component like I/O process component, or it is for the specific domain such as bank account component

Development Theory

Give the general way about how to build the software component, and indicate the proper infrastructure

Tools

Explain the correct tools to design, analyze, write and test the component, and maybe give the relative tutorial or website

Entities

The entity of component means the features of concrete implementation

Kinds

The classification of the software component, and this could be understood as the typed role in component-based system, like standard component, aspect, connector and agent.

Mixture

Describe how to implement the software component, whether it is for example pure OO programming, or it includes multiple way to get a compound result

Context

Point out the compatible environment or model, where the component should work well.

First-Class

Represent granularity and priority in building the software component, and show the possible reflection and meta-programming

Interface

Generally speaking, the interface is the contract between the client and the component. The interface is the key of the component which distinguish one from the others, so interface decides the type of software component. It also has some important factors to specialize

Syntax

Give the abstract syntax to explicitly express the interface part of the software component

Element

Describe the different parts in the interface, such as Port, Service, and so on

Communication

Explain the needed communication pattern or protocol with the software component

Attribute

Give precise semantics of the attributes in the interface, and also the condition where the attribute works

Life Cycle

Describe the active phase of component interface and how the component become active to the client, the inactive phase as well

Dynamic Behavior

Describe the different execution state of component interface according to the run-time environment, and the corresponding denotation through element

Other Properties

Give the other useful property about the software component, for example, the quality of response in the average workload, also the static or dynamic network port-number of access

Although the above characteristics list still has something to get further refinement, it is definitely the comprehensive specification for component development.

1.2.2 Component Development Principles

Since the component is the basic software unit for the system development, the development of reusable component is a big but important work. It is very important to carefully identify and assign the function to each component while doing architecture design. Moreover, the developer has to design the proper interface and relative context dependency to make the efficient and flexible interaction among the components. A correct design and implementation of component is the key for the quick and successful integration of component to build software system.

There is a common way to develop the reusable component for some software system [16], shown in Figure 11. The life step “Search relative resource” means that according to the problem and solution domain, the developer searches for the relative source about how to solve the problem. Here the resource could be in various forms like formal publication, design pattern, university lecture.

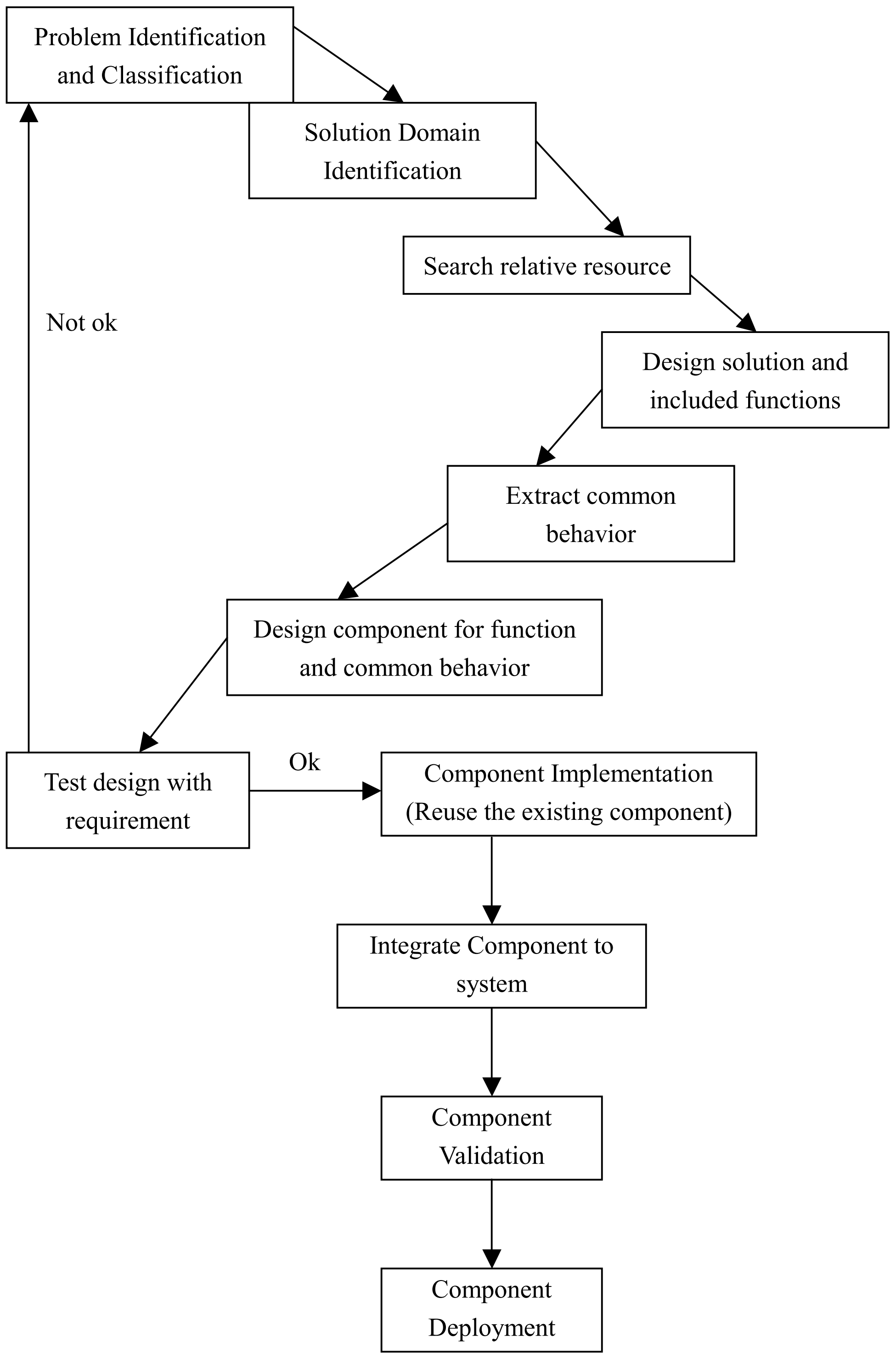

Problem Identification and Classification
Solution Domain Identification
Search relative resource
Design solution and included functions
Extract common behavior
Design component for function and common behavior
Test design with requirement
Not ok
Ok
Component Implementation (Reuse the existing component)
Integrate Component to system
Component Validation
Component Deployment

Figure 6: Life Steps of Component Development

The common behavior in Figure 11 is referred to the action happening in more than one component. Thus it should be separated and encapsulated into a new unique component. The common behavior could not only be used to design a common-functional component reused by other components, but also be treated as a common interface to be re-implemented by other components. The common interface looks like the Figure 12.

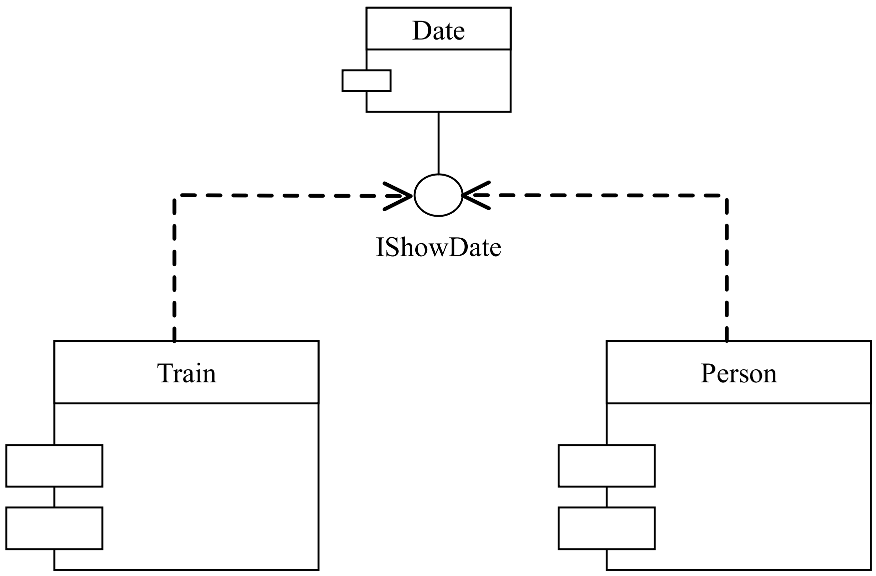


Figure 7: Interface Reuse by Components

In Figure 12, the interface IShowDate, which is defined in some component Date like a virtual class of C++, is reused by component Train and Person. It means that Train and Person both implement the interface IShowDate, or say write the code for the method of IShowDate. Component Train may use that interface IShowDate to show the date of train schedule while Person may use it for the birthdate of individual, even in the different format. Likewise, the new component could be created for the common behavior but delegate some parts or all of work to the different components according to the context.

As seen from the above life steps of the component development, it has an iterative process to analyze and solve the problem before getting the ultimate component design. The real operation in the component development always depends on the concrete application and development environment. Thus the developer would make subtle difference during component life cycle. However, there exist some basic principles of component development [13]:

· **Solution-specific component**

This principle is easy to be understood, since every component is born to serve the certain solution for a problem given by the client. The developer has two choices for the component to be used in the solution, one is domain-oriented, and another is domain-independent.

Domain-oriented component is to be developed as the common function unit for the software system in a certain domain. Since the domain-oriented component only solves the problem in a fixed and limited scope, usually it would get higher productivity in the system. Nevertheless, if the solution is going to be used for more than one domain, this kind of component has to be wrapped or composed with other components, to realize the reuse. For example, a bank-account component is excellent for all the bank system to provide the common function of financial operation on bank account, while it has to be wrapped into a new component to process the function of medical-insurance account.

Domain-independent, or say generic component, on the contrary, is developed as the general unit for the software system in multiple domains. The different domains usually have the different function requirements even to the same problem, and sometimes there would be the conflict among these functions. Therefore to develop domain-independent component would take more cost and time, but it would get more reusability for the various system developments. If the solution seems not to be reused cross the domain, the generic component development is not necessary and is an over-expensive work.

Therefore, all the components must be developed to work with the solution as the premise. The developer should think about the tradeoff between the domain-oriented component and generic component and make the rational estimation, according to applicable scope of that solution.

· **Separation of concerns**

This principle means that the developer should divide the function into smaller pieces for the diverse concerns in the application. The finer division leads to make the corresponding component of the function more simple and flexible, and thus more reusable in the component-based software development. As a matter of fact, it is a choice of component granularity. The effective degree of separation depends not only on the system function, but also on its quality requirement, for example the evolvability of the system.

· **Abstract Virtual Machine Interface**

This principle is to emphasize the polymorphic characteristics of components by component interface; in other words, the interface should be abstract enough to hide all the information of concrete implementation in the component. The interface is independent to the specific technology or platform, so different components could implement same abstract interface with different operation. This principle is similar to the single-interface-multiple-inheritance in OO programming language.

· **Component hierarchy**

Generally speaking, the software architecture could be regarded as a kind of hierarchy structure. Every component should be developed to classify it to a certain level in the component hierarchy, in order to reuse it in a uniform style. In [13], it suggests a layered model for component hierarchy as below:

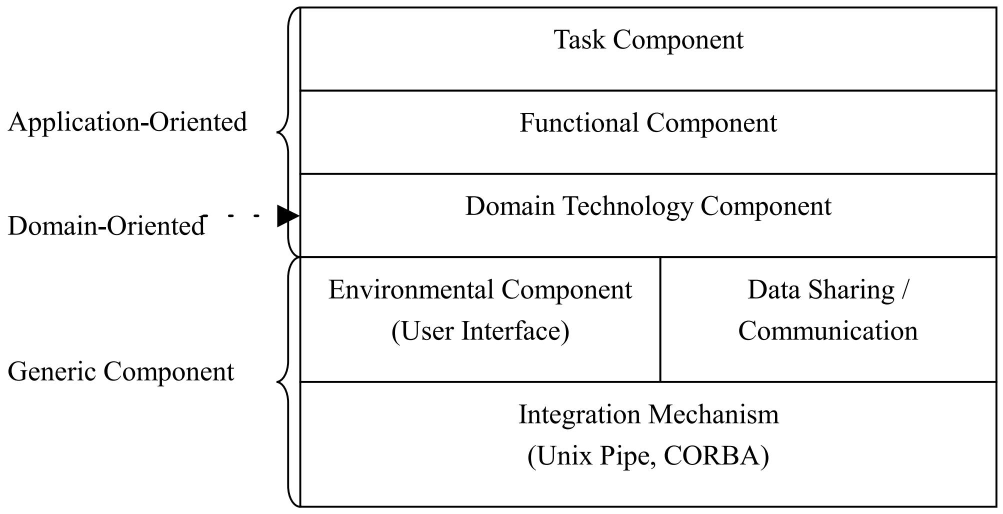

Application-Oriented
Domain-Oriented
Generic Component
Task Component
Functional Component
Domain Technology Component
Environmental Component (User Interface)
Data Sharing / Communication
Integration Mechanism (Unix Pipe, CORBA)

Figure 8: Hierarchically Layered Architecture of Component

· **Postponement of context binding**

In the architecture design process, all the details about the implementation of components are ignored. The work focuses on the basic function of component, communication and other structural aspects at a high abstraction level. Therefore, Binding the component to the property of its executable context like data type, poll-interval, database connection, is delayed to the phase of component integration in CBSE.

· **Design reuse**

In order to find the right component for the component-based architecture to be integrated in the system, an easier and feasible way is to reuse some previous design on component. In fact the design reuse is much popular nowadays. It addresses the similar problem or requirement occurring in many former system developments to reduce the cost and time. Therefore, the component would get more reusability with its shared design.

1.3 Current Status of Component Development

The promising component is playing predominant role in current software development, but it still does not satisfy people who always want to get the max benefit from the reuse of existing component. Software component is being given more study and support to overcome the emerging problems and detects in component-based software development.

The component model is always the basic issue for the component development, since there is no uniform way to build the component model. Therefore there would be some problems in the heterogeneous system with different model to reuse the software component.

As the result of the problem of component model, the interoperability among the component appears in front of the developer. Fortunately the main component company is making effort to have the component interface understood by the component from different component model.

Any component can’t foresee all the change in the future after it running, which means that every component has the doom if it never were upgraded. The problem of customizing and dynamically maintaining the component need more attention to make software reuse persistent and cheap.

The delivery of component is an interesting choice for the producer and client. It is difficult to define the proper granularity of the software component for the requirement of different client.

In addition, currently most of programming languages do not support component programming well, and there are few component languages like ComponentJ [33], to specialize the component in the design level or code level.

CHAPTER TWO

COMPONENT MODEL WITH COMMUNICATION

Today there are many components produced for the large system, such as applet, plug-in/add-in, framework and third-party control/widget. From the introduction of component in the Section 1, it is clear that component-based software is built in the process called component-based software engineering [13, 17, 18]. Here component-based software engineering (CBSE) is referred to the creation and deployment of certain system assembled from the proper components. CBSE also includes the component development and reuse of the existing components, like using Lego brick to get some useful shape in some way.

Thus from the view of accessory assembly, the component-based system would be often heterogeneous. For example, the components inside may be produced with different programming language, even the same language but of different version. Moreover, the whole system as a large compound component could be reused for other complex application.

It is more evolvable for a system to replace the single component with the new version, as long as the interface of substituted component is compatible with or same as before in term of semantics and behavior.

In addition, according to the property of component, the component-based system would get good distributed operation by the components that are scattered in the network. The software system also could be smartly mobile and adaptive in the network due to its components.

However the ultimate power source exists in CBSE. The key in CBSE is the software architecture involving component model. Software architecture defines the global behavior of the system, and the individual action of each component [15, 18, 20, 21]. The relative synchronous or asynchronous communication among the components is also decided in the architecture design as well. All of these of CBSE are explained in the following of this section.

2.1 Component-Based Software Engineering

The software engineering simply means to find the executable solution for the questions in the given domain, graphically like below:

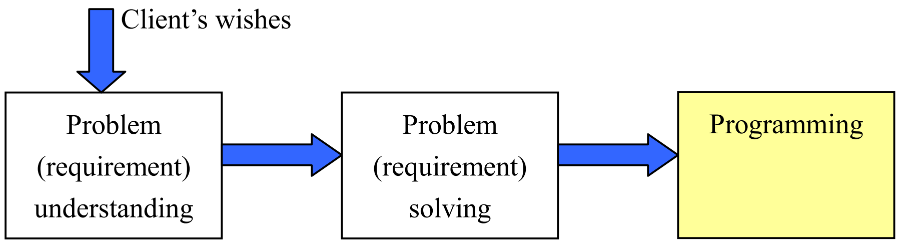

Client's wishes
Problem (requirement) understanding
Problem (requirement) solving
Programming

Figure 9: Overview of Software Engineering

Traditional software engineering typically is done by a group or a single organization [17]. Traditional software engineering usually matches a phased development process to make the milestone version released as the schedule [17]. The main life steps of traditional software engineering looks like below [17]:

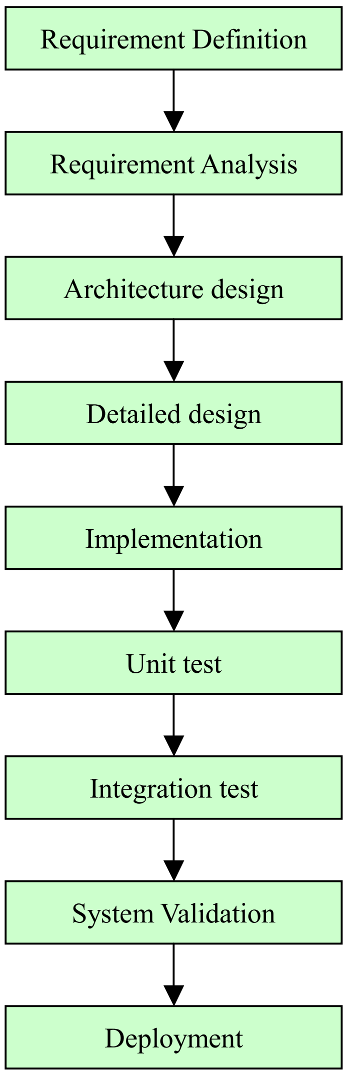


Figure 10: Life Steps of Traditional Software Engineering

There are some limits of reusability in traditional software engineering. Only the user interface of the software is shown to the client. The programming interface or design interface is known to the software developers or owners. Therefore, there just exists very small scope of people who can make reuse of the parts of software. This kind of software engineering usually leads to the efficient solution for a special problem in a certain domain. On the other hand, it often brings the expensive evolution of software if the corresponding problem evolves or that software needs to be changed or customized for the new user or requirement.

The component-based development always wants to reuse the existing component to maximize the productivity and minimize the cost and time on component implementation. Thus there are some unique features in CBSE to stand out component reuse. The main life steps of CBSE looks like below [17]:

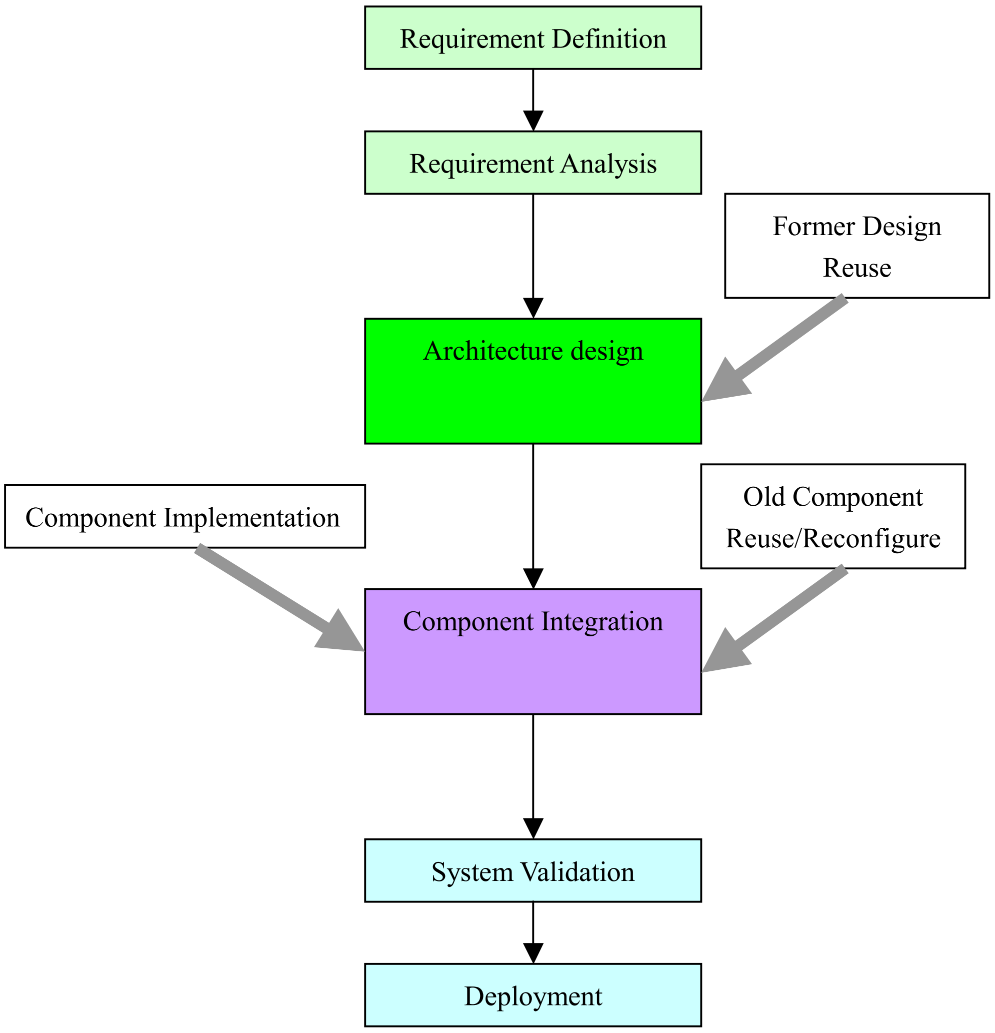


Figure 11: Life steps of CBSE

According to the life steps of CBSE and traditional software engineering, reader could find that in CBSE the work on design, programming and test is decreased a lot due to the reusable component. Instead the integration of component becomes the basic process in the component-based software development. In the component integration, if the developer can’t find the existing proper component to reuse, or to reconfigure it to subject to the system, he must implement the new component to complete the required function in the system.

Supposing that the reusable component is the previous work by the same developer, it is easy to integrate it to the new system since it is just the white-box artifact for that developer. On the other hand, if the component is given from other organization, the developer must carefully check the document of this component. The checking should include the important characteristics of component listed in the section 1. The developer should also find the practice information from the a few running system where that old component is being used.

Generally, there are two ways to get new component. One is to design and implement the component all by hand, and another way is to wrap or merge the similar existing component with some additional data and code [17]. The latter method means to make a partial-new component conform to the system architecture design. In order to wrap old component, the developer often needs to know more information about how the existing component does its function inside. Understanding more of existing component, the behavior of it could be well controlled and customized to build the new structural component.

After making every component work well, the important step is to validate the global behavior of the component-based system. Here system validation means to check whether the function through collaboration by multiple components is perfect as expected. It is difficult to test the black-box components that hide the detail by the strict encapsulation. Another problem of validation is that the original context for the development of existing component would have subtle difference to the current software environment.

Although the reusable components reduce much possible duplicated work in the development, it is still difficult to find the proper existing component corresponding to the component model as the result of architecture design. There is not a globally uniform way for system designer to build the software architecture with component. Meanwhile, it is much more expensive and complex to develop the reusable component than the simple program even for a single application. Thus, the component-based architecture design is being paid more and more attention to make it as a basis to well ease the subsequent development process in the CBSE.

2.2 Software Architecture and Component

Thanks to the previous development of reusable component, CBSE could put the less work-amount on the concrete implementation of software system, through the composition of the components. However this kind of composition must need a guideline to get the right component and put it in the correct position. This guideline is called software architecture in IT industry. Software architecture could be considered as the base-board of Integrate Circuit of all kinds of electronics.

Software architecture generally is the structural high-level abstraction to describe the main structure or say skeleton of the software system [15, 18, 20, 21]. It includes the relative rules and standards, static data and dynamic behavior in the system [15, 18, 20, 21]. Moreover, because the software architecture decides all the parts of the system and their relationship, it defines the main functions and qualities of the system. On the other hand, the property of the software system could be reasoned out according to its own architecture. However, the software architecture always focuses on the global performance, and ignores the detail of each part which is the center of detailed design process in traditional software engineering.

The core of architecture for component-based software could be simply represented as the components with their communication, also called component model. The component model is the blueprint for component integration, development, deployment and so on [14, 15].

One formal way to describe graphically the component model is to use UML language [34], as shown in Figure 3. It is called component diagram [6] in UML. Obviously, the component diagram at least shows each component which is assigned the abstract functionality. It also explicitly gives the interface and context of component to interact with each other in the system environment. Here the interaction between components should include the (remote) procedure call, event broadcast, channel etc. Moreover, some important architecture constraints to the components and their interaction are also expressed in the component model. For example, it could explain how the constraints of communication mode (asynchronous or synchronous), are designed with components.

An elegant component-based architecture could not only give the correct design to easily find the reusable existing one to be integrated in the system, but also be reused for the other system development at the architecture design level. This means that the different applications could share some common software architecture so that the relative components of that architecture get the max reusability for the development. Therefore the common component-based architecture improves the interoperability between multiple software systems.

2.2.1 General Requirement on Architecture

There are some important requirements for the component-based software architecture to be promising for the component-based development. The requirements also explain the direction of CBSE in the future, listed below [15].

· **Multiple component granularities**

This asks the architect to design the component architecture with different size of component. There may be atomic component and structural components in the architecture. Atomic component is the smallest function unit unable to split, while structural component is composed of several smaller components in a logical way. The member of structural component could be atomic component, or also structural component. It is obvious that the smaller component in structural component could be extracted out and reused for other things. There are already a lot of reusable components with different granularity now, for example, file reader as small component, word processor as large component.

The choice of component granularity depends on the system requirement and component characteristics. Usually the small component is more flexible and cheaper, whereas the large one is more versatile and expensive. In addition, the architecture should be able to deal with the compound component that is constructed by many small components.

· **Substitutability of component**

Component architecture should be able to allow the replacement to one or several components with the component, which has compatible interface and same function to continue the system running.

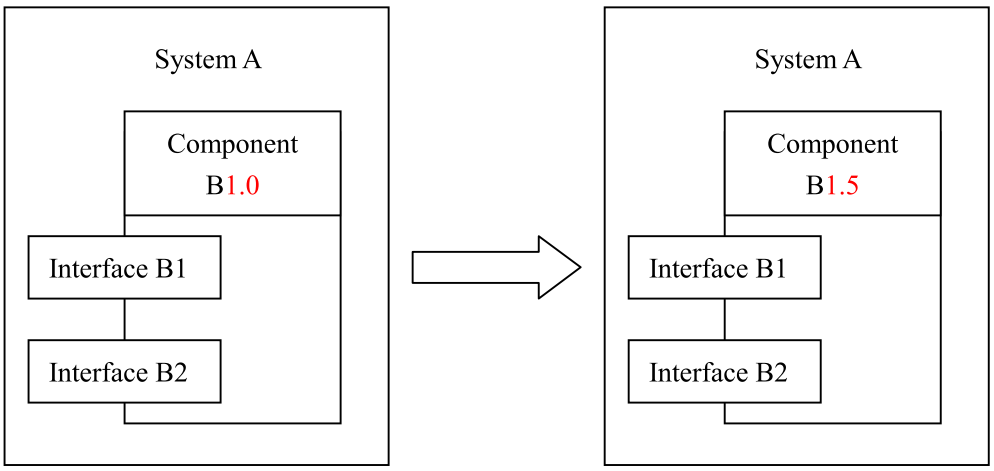


Figure 12: Substitute component by updating

Moreover, this kind of substitutability could be dynamic and made in the run-time.

The component-based system could perceive the needs of component replacement, and intelligently find the proper new component to replace the old one while system running. Therefore the maintenance becomes more convenient and this is particularly important for the real-time software system. The usual way to substitute the component is to replace the old version with new version to get better performance with fewer bugs, like Figure 9.

· **Parameterizable component**

The component in the certain software architecture has the confirmed interface and function. In order to reuse generic component which could be tailored by parameter, the architecture should be able to set the parameter to that kind of parameterizable component. The parameterizable component usually is the synthesis of a set of similar functions in a problem domain. For example, the picture-browsing component could be designed to show the different pictures according to the parameter of picture type. Therefore, the architecture enlarges the scope of reusability with parameterizable components.

· **Customizable component**

This requirement is close to the parameterizable component, but the customizable component is to be used by the end-user instead of developer. User can adjust the component to different environment or work, without replacement by the other component. In this condition, the architecture should make the additional support to the customizable component, for instance, having an easy-to-use GUI component to reflect or setup the customizable component.

· **Multiple programming language**

Since the component should be language-independent to be reusable, the corresponding architecture of component should also contain the component implemented by the different programming language varying from Fortran to C# [35]. However, although the component is allowed to be developed by the different language, the interface of component must be designed carefully to be understood by the other component. Additionally, the incompatibility between languages must be considered to avoid possible problems.

· **Component-specific help**

A lot of existing components are blackboxes with only interfaces exposed to the designer or user, but it is often necessary to get more information of the components. The component architecture should possess the ability to give the help of certain component to the architect or software client. This kind of ability also depends on the way the component show its help, such as the email address of the component author.

· **Component-specific changeable or integrated interface**

Besides the component being parameterizable for its specific function, there may be the needs to make the interface of component able to be tailored for the different application. Moreover, there could be an integrated interface from multiple components; in other words an interface is composed of several interface of multiple components to form a work group for the specific task or the typical user. Thus the architecture should support this kind of changeable or synthetic construction of interface in a structural way to provide the flexible reuse.

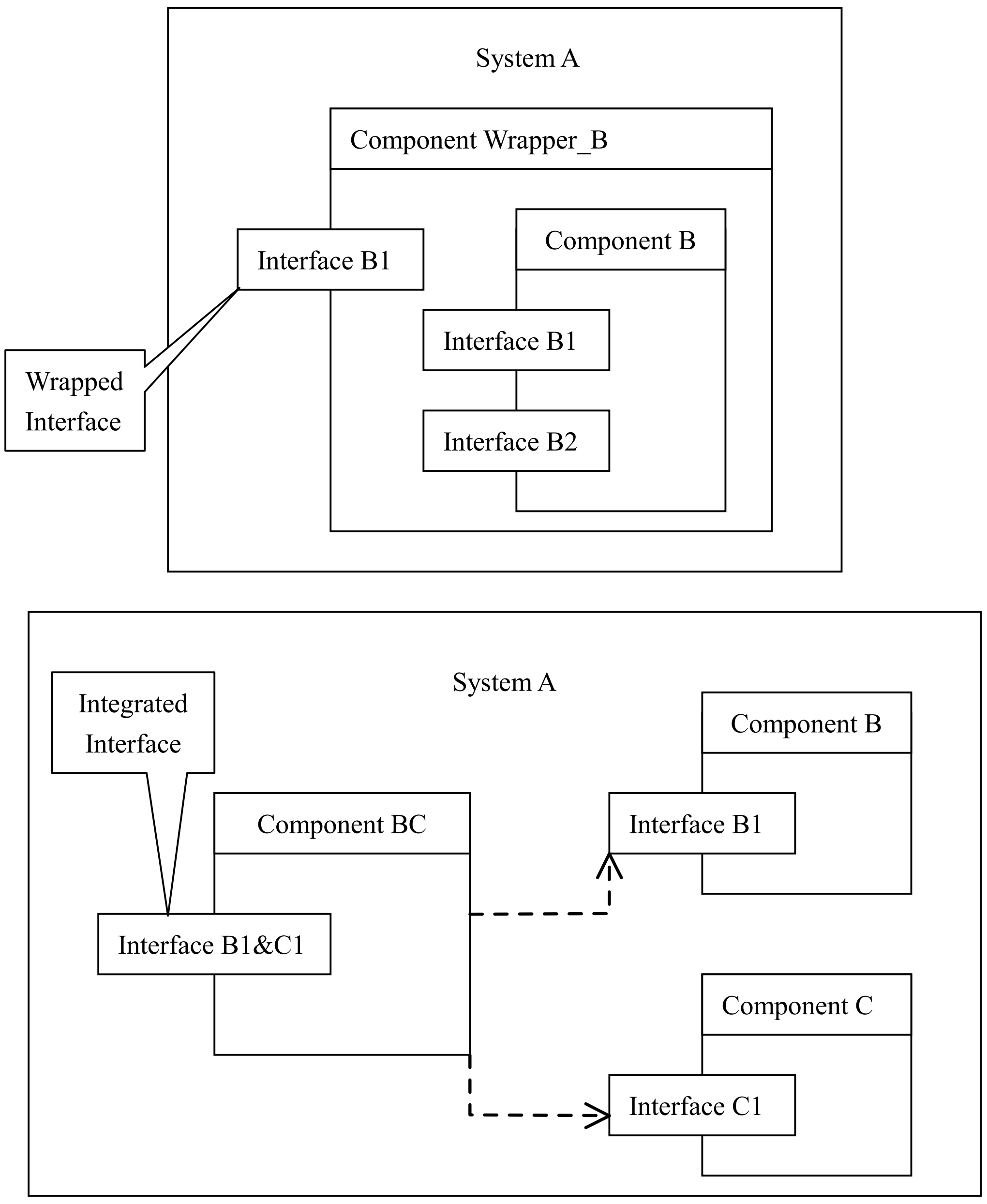


Figure 13: Wrapped and Integrated Component Interface

The usual trick to get the tailored interface is to wrap the relative components. Integrated interface could be done by creating a new component as the function proxy to the components that are responsible for the implementation. Both are shown in Figure 10.

· **Easy distribution of component**

One of the basic benefits of component is that it can be separately delivered to the designer or end-user. In addition, the component could move in the network by itself if necessary. Therefore the software architecture should make the independent component in the independent format, for example a unique Dll (Dynamic linked library) file. This development style would ease the deployment and maintenance of the component.

· **Support for sales**

This requirement is a non-technique request to the software architecture, but it is important for the commercial component-based software system. Since the company would have the different price policy for the different sales mode, the software architecture should be able to easily match with the various charge way. The usual method is to use a parameterizable component, which could be set up in the installation or deployment process to fix the sales mode for the certain version like time-limited usage. Therefore the software architecture would get more reusability in the CBSE for commercial software.

2.3 Communication between Components

Although the component is the basic unit in the software, it is the communication that glues all the components to form a complete system. Every "living" component needs communication as oxygen to exchange the information with other relative parts in the cyber-space. The communication between components could be simply described as that one component sends the proper message to another component according to the interface of both. Since every component conforms to the information-hiding principle, the communication only happens at the component interface level. In other words, one component can't directly call the internal sub-functions or sub-components of another component.

Emerging network and Internet bring more requirements on the communication of components. The space and time scope of component communication has been extended and enforced to execute the network-based application. The communication must be adaptive to different technology of the network, also the heterogeneous operation environment, to cope with the platform-independent and automatable component. In addition, because the network is becoming huger but more distributed in terms of scale and function, the software is difficult to strictly control its component communication in the network-based workflow. In this case, there could be more errors possibly occurring in the communication than on the single machine. Therefore, now the communication between components is the hotspot of the researchers for the component-based development.

There are a lot of study and analysis on the communication schema among the components. From the view on time phase of action steps in the communication between two components, it is divided into synchronous communication and asynchronous communication. Here the synchronization and asynchronization of communication is not referred to the low-level concrete transmittal of binary data, but to the high-level message transportation. The choice between synchronous communication and asynchronous communication directly affects the system performance and quality. Thus it is very important to carefully use both communication modes in the architecture design of CBSE.

2.3.1 Synchronous Communication

If a component A is blocked (waiting after it sending a request message to another component for some function) until the message receiver return the feedback message to the sender A to indicate the result of function corresponding to the request message (at least the acknowledgement to the sender A), this kind of communication is called as synchronous communication. The MSC (Message Sequence Chart) of synchronous communication between the components is looks like below:

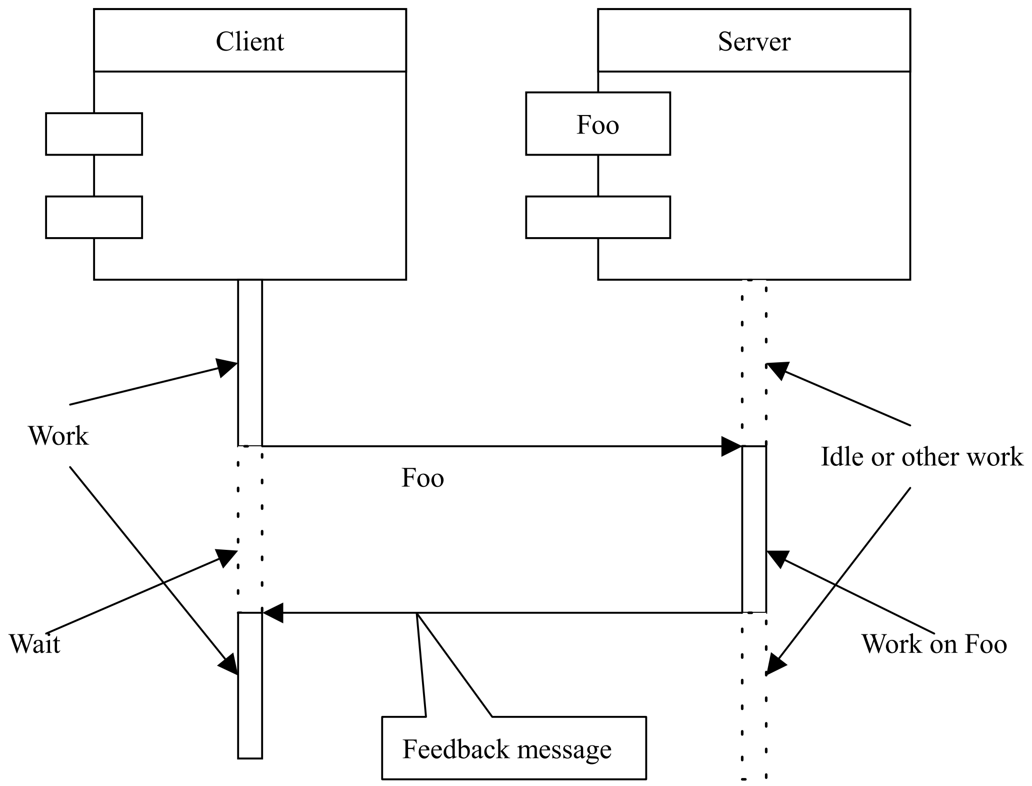


Figure 14: Synchronous Communication between Components

As seen from Figure 14, the Client component sends a message Foo to the Server component with corresponding interface for that message. Then the Client component suspends its work until the feedback message is sent from the Server component after its work due to receiving Foo message. After getting the feedback from Server, the Client component continues its execution until it sends the message again to another component in a synchronous way. Therefore, the function with synchronous communication could be seen as a set of sequential steps by the ordered collaboration by multiple components.

Briefly speaking, the synchronous communication couples the phase of message sending and relative execution as a tight pair. Thus it virtually produces a persistent link between the message sender and receiver, until all the steps in the communication are finished.

Currently, the most popular example of synchronous communication is online discussion through the Internet, such as chatting room. When one guy sends the message to another, the chatting component is blocked until the other side returns the information about receiving the message. The chatting component would display the current status or session record with that new message if successful, otherwise it will report some kind of error.

2.3.2 Asynchronous Communication

Obviously there are a lot of waiting state in the synchronous communication between components, but people wants to have a way to make use of that waiting time to do other thing to increase the productivity. Thus Asynchronous communication has been used to reduce the waiting cost, shown in Figure 15.

The Figure 15 explains the general way of asynchronous communication, where the Client component proceeds to do its work after it sending a message Foo to Server component. Actually in asynchronous communication, the Client component and Server component works in a parallel way without waiting for the finish of that interaction by Foo. If the Server component gets the result by the work due to receiving the message Foo, it would send that result to the Client component even if the Client were still running.

Note that although a component could continue to work after it asynchronously sends a message to other component, it is possible for that component to enter the waiting status. This is due to the fact that the work of sender may use the result of the previously sent message. Thus the sender has to wait if the result is not ready by the execution of the message receiver. Asynchronous communication could not completely avoid the waiting state in synchronous communication.

In essence, asynchronous communication decouples the message sending and relative execution to make both message sender and receiver in the communication independent to each other as much as possible, while synchronous communication links sender and receiver in a tight style.

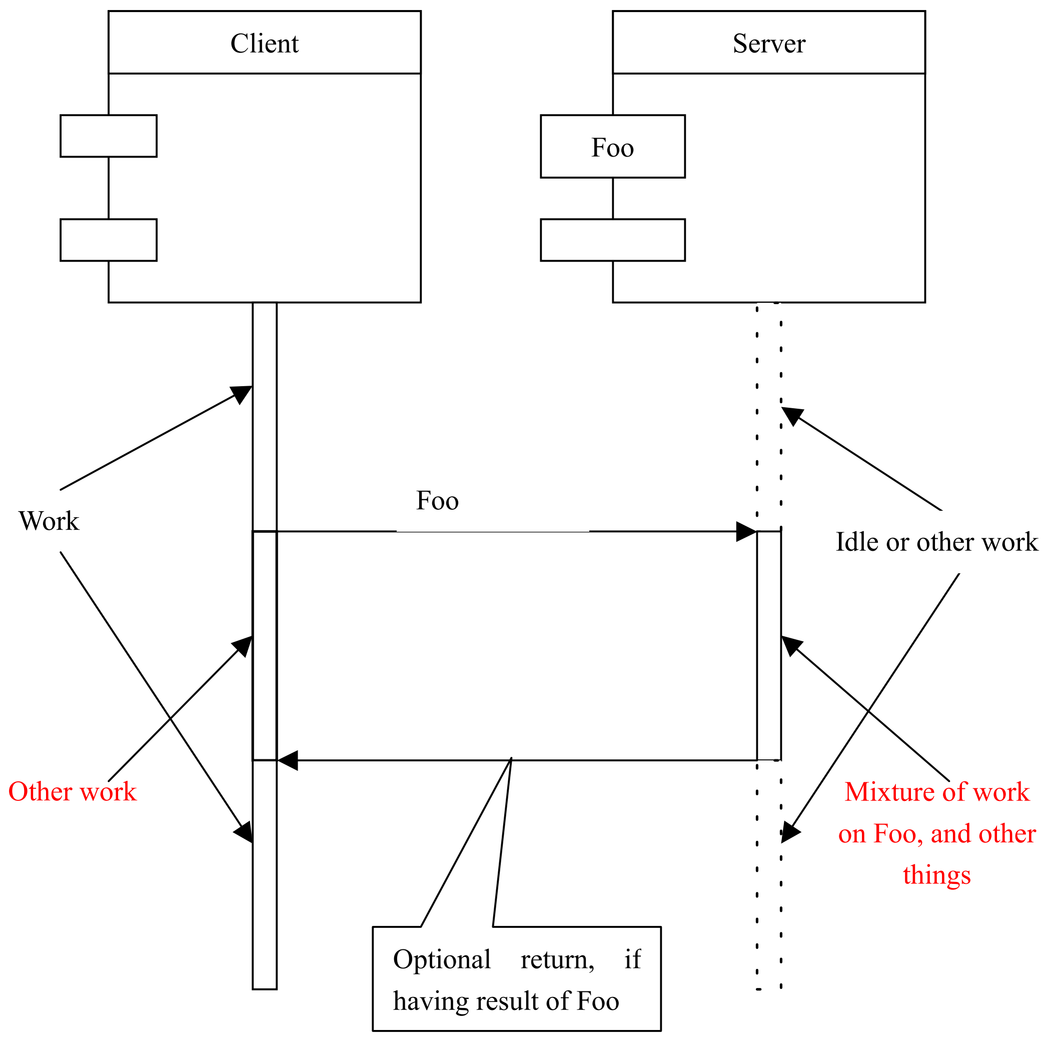

Client
Server
Foo
Foo
Work
Idle or other work
Other work
Mixture of work on Foo, and other things
Optional return, if having result of Foo

Figure 15: Asynchronous Communication between Components

There are two main methods by the message sender to handle with the result returned in asynchronous communication while the sender is doing other work:

1. The message sender is to use a thread to poll for the result of its message sending. That's to say, the message sender timely sends the poll message to the receiver to ask whether the action on previous message is finished or not. If it were finished the message sender would get the result from that message receiver. Therefore, this method would increase the communication cost in the interaction, but it is simple to implement.
2. The message sender is to adopt a callback function to the message receiver. In this way, the message sender gives an interface of a function (callback function) to the message receiver, when the sender sends the message. After the message receiver

finishes the execution on the received message, it would use that callback function which gets the result of original message. Then the callback function do some proper work like notification to the message sender, so this way would have the less message transferring, but it is more complex in the development.

Currently, the most popular example of asynchronous communication is information transmitting through multiple peers in Internet, such as email system. When one guy sends an email to another, the client-end continues to work no matter the transferring result of the email. The email-server would return the result if it fails to send to the email receiver after its action on that received email. At the end, the client-end would be notified for the failure of email while it is working on other stuffs for example writing new email.

2.3.3 Synchronous .vs. Asynchronous

Although synchronous and asynchronous communication obviously has some unique features based on the different messaging style, the distinction between them has been researched for a long time. People found that neither of them could be absolutely better than another one. However, The interesting thing of both kinds of communication is that they can simulate each other in the software development.

The classic way changing synchronous communication to be asynchronous is to insert the message buffer (message queue) and relative management in the middle of message transferring. The example of this kind of trick is channel, Publish/Subscribe pattern [4] and Mailbox in chapter 4.

As to make asynchronous communication be synchronous, one simple transformation is to give the acknowledgement to message sender as soon as the sender transmit the message to the receiver. Afterwards the system should use other mechanism to notify the true result of execution by message receiver.

The usage of synchronous and asynchronous communication depends on the real application and the execution context. Therefore there are a lot of software systems mixing both communication modes to make full use of their advantages and reduce the drawbacks. As for the architecture design in CBSE, the synchronous and asynchronous communication should be equally important for designer to carefully adopt for different components.

The quality of service in synchronous communication obviously is better than asynchronous one, since synchronous communication requests the response from the message receiver as soon as possible. That’s to say, the message receiver in synchronous communication should immediately do the operation according to the received message. Nevertheless, on the side of asynchronous communication, the message receiver is not much enthusiastic to give the action on the received message since it has its own operation schedule. The sender of asynchronous messaging would continue with other tasks if the result doesn't affect the afterward work. Actually the receiver in asynchronous communication may ignore the message for some time, putting it somewhere until the receiver think it possible to execute the function for that message. The difference on the quality service decides that synchronous communication is more proper to be used in sequential transaction system to keep the application with high efficiency and safety as expected by requirement. Meanwhile, it is difficult for asynchronous communication to estimate the time when the result is done. Therefore asynchronous communication would lead to the chaos of failure or success, and the choice of resending or ignoring.

Excellent service usually means expensive cost; in other words, the synchronous communication would take more overhead than asynchronous one. In synchronous communication, after the message sender transmit its message out, it has nothing to do but wait for the result. The waiting of component often makes some part of a software system idle for the CPU-time, especially when the message brings a time-consuming job to the receiver.

The worst thing in synchronous communication is that it could go to deadlock, since the message sender can't release the owned resource until it gets the result to continue its work. Furthermore, the recovery from the deadlock is a big loss to the whole working system. In addition, the synchronous communication asks all the involved parties to be active at the same time, which produces the long time and big load for the system to prepare for one large synchronous communication. Thus it is not realistic in distributed and concurrent systems.

On the other hand, the asynchronous communication gets more concurrency during the messaging. The message sender doesn't have to wait unless it need the result in its operation, and it could release the resource temporarily after it sends the message. Moreover, the message receiver could be inactive or busy with other stuff during the asynchronous communication, and to operate the message later. This kind of manipulation is more and more useful to the bandwidth in the Internet age to avoid network-congestion. Therefore, asynchronous communication obtains more attention from loosely-collaborating and distributed software system development for the long-time execution pace like database merging. The overhead of synchronous communication becomes an important factor to make designer consider the tradeoff with its required quality of service.

In the history of computer science, synchronous communication has been researched for a long time because it is easier to be understood simply as an ordered sequence of operations among the components in a single collaboration. Synchronous communication follows the general logical thinking of people. Today synchronization is a common part in the software system. There are many specific technologies used to help design, implement and optimize the synchronous communication, for example, process management by operation system. As for the component-based software development, the main component models and infrastructures (EJB [36], .NET (COM & DCOM) [37], CORBA [38]) all have the basic support to the synchronous communication between the components.

In contrast with synchronization, the asynchronous communication is difficult for people to obtain in the software production, because it has much higher parallelism not only on software but also on hardware, complex to concrete development. Asynchronous communication is more difficult to obtain the result in a rational time than synchronization which is based on presumptively successful communication. It is also harder to control and verify asynchronous communication according to its corresponding software architecture. In addition, there is less support to asynchronous communication from programming tools. Recently, some languages like ProActive [39] and Sharpie [41] claims to have default asynchronous communication, and EJB, CORBA also provides some way to get asynchronous communication.

However, now there are many new ideas that require more asynchronous communication than synchronous one. For example, distributed computing which is based on the network to assign the parts of one task to different machines, the advantages of asynchronous communication is very clear in that environment. Thus the developers need asynchronous communication to solve the big trouble in synchronous communication. Moreover, many kinds of component are rather asynchronously communicating entity in the system. The typical asynchronous component is intelligent agent, which always performs the tasks from the client while it sends message to delegate some sub-tasks to other components without suspending for the result.

CHAPTER THREE

IMPLEMENTATION OF ASYNCHRONOUS COMPONENT MODEL

Since component and communication become the basic keys to the modern software system, the communication modes: synchronization and asynchronization has to be chosen to compose the component to software system. However, there is less technique support to develop asynchronous communication between components than to make synchronous communication. Thus most of the software system only holds synchronous communication, although asynchronous communication is more useful at some time.

One of the basic things to support the usage of component-based software with asynchronous communication is the programming language. As a matter of fact recently some new experimental programming languages claim to have built-in mechanism for programming of asynchronous communication, like ProActive [39], Piccola [40], Sharpie [41]. If more precisely, these languages are just the extension of their host programming language from the view of implementation. Since ProActive is going beyond Java which is as industry-standard language, it would be more practicable for the software development. In this paper, I would explain how to implement the component model with asynchronous communication by ProActive.

3.1 ProActive

As known from the experience of programming in Java, it provides synchronous communication as default mode for the method call or messaging between the objects. In order to fix Java with another powerful wing to be adaptive in the environment of asynchronous communication, ProActive [39] was born to ease the programming in that situation, and in component-oriented development as well.

ProActive [39] is actually a set of java class libraries that add many interfaces and relative functions for parallel, distributed, concurrent programming with fine security and mobility. It uses underlying Java RMI layer [42] to realize all its ideas including asynchronous communication, and thus it is adequate to be used for the network-specific application.

Because java is a kind of pure object-oriented programming language, ProActive [39] program is based on the objects and messages. ProActive adds a new kind of object to the java software, which is called active object. Active object is the first-class object in ProActive, encapsulating the common object (called passive object in ProActive). Only active object is shared by all the other objects in the system. In other words, the interface of one active object is open to other active objects or passive objects. On the other hand, one passive object in ProActive could expose its interface to some relative passive objects and only one active object.

The general object model in ProActive [39] could be graphically represented as below, and the arrow link means the message sending between the objects:

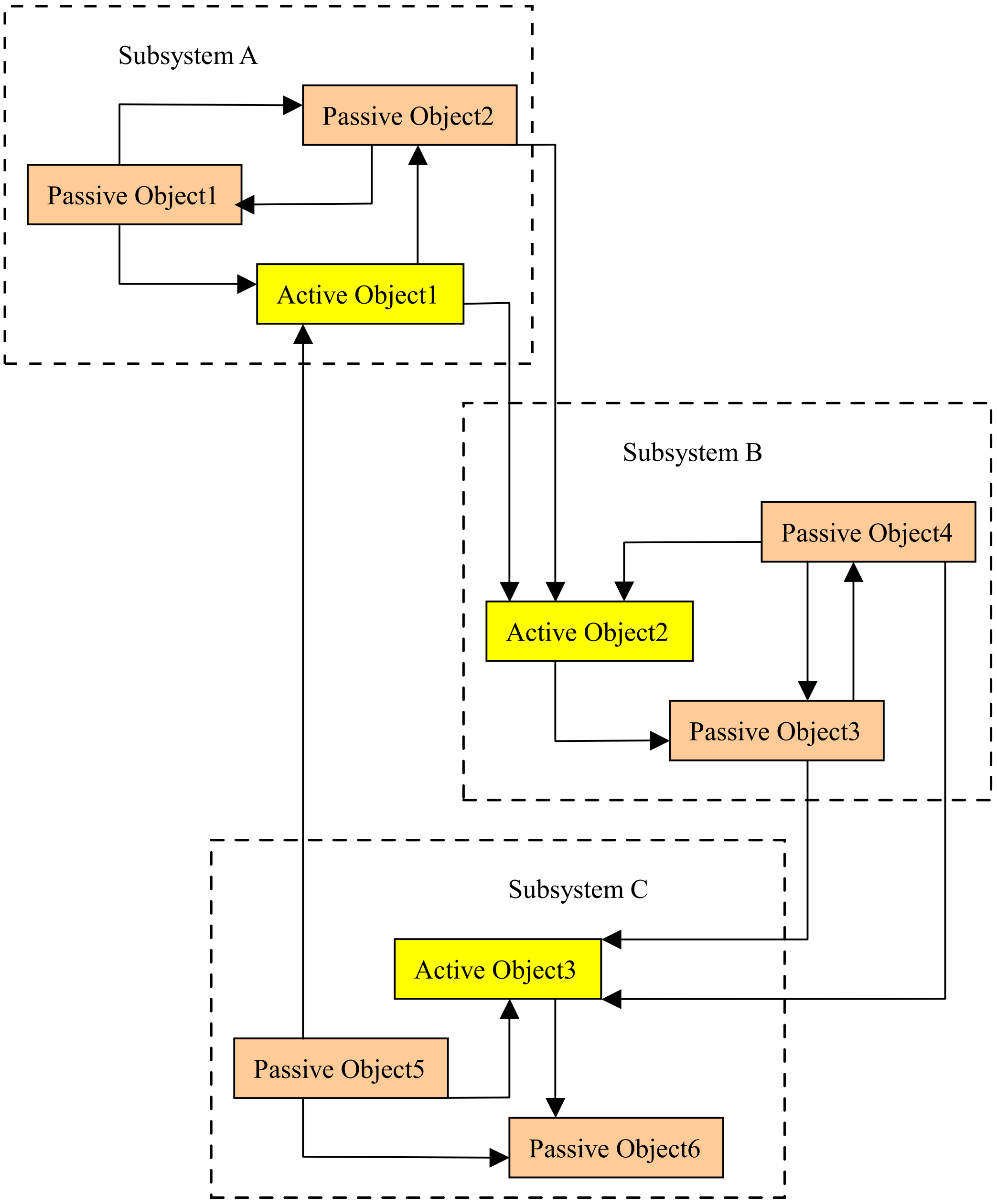


Figure 16: ProActive Object Model

The actual implementation of active object in ProActive [39] is to wrap the common object created from standard java with RMI mechanism. The wrapping makes the active object become the remote object which has some additional parts like stub and skeleton to form a kind of client/server structure.

An active object A of some class, from the view of RPC, will have 4 parts in fact:
1. stub_A (client side)
2. bodyproxy_A (client side)
3. body_A (server side)
4. instance_A (server side)

The parts of client side means that stub_A and bodyproxy_A should be used in the same JVM [43] as the client which calls the function of active object. The parts of server side means that body_A and instance_A could be put in an independent JVM from client object. The body_A and instance_A provides the real service for the request from client side, and their location depends on the creation and deployment of active object. The Figure 17 shows these 4 parts and their roles in the call to active object, and arrow link represents the message sending.

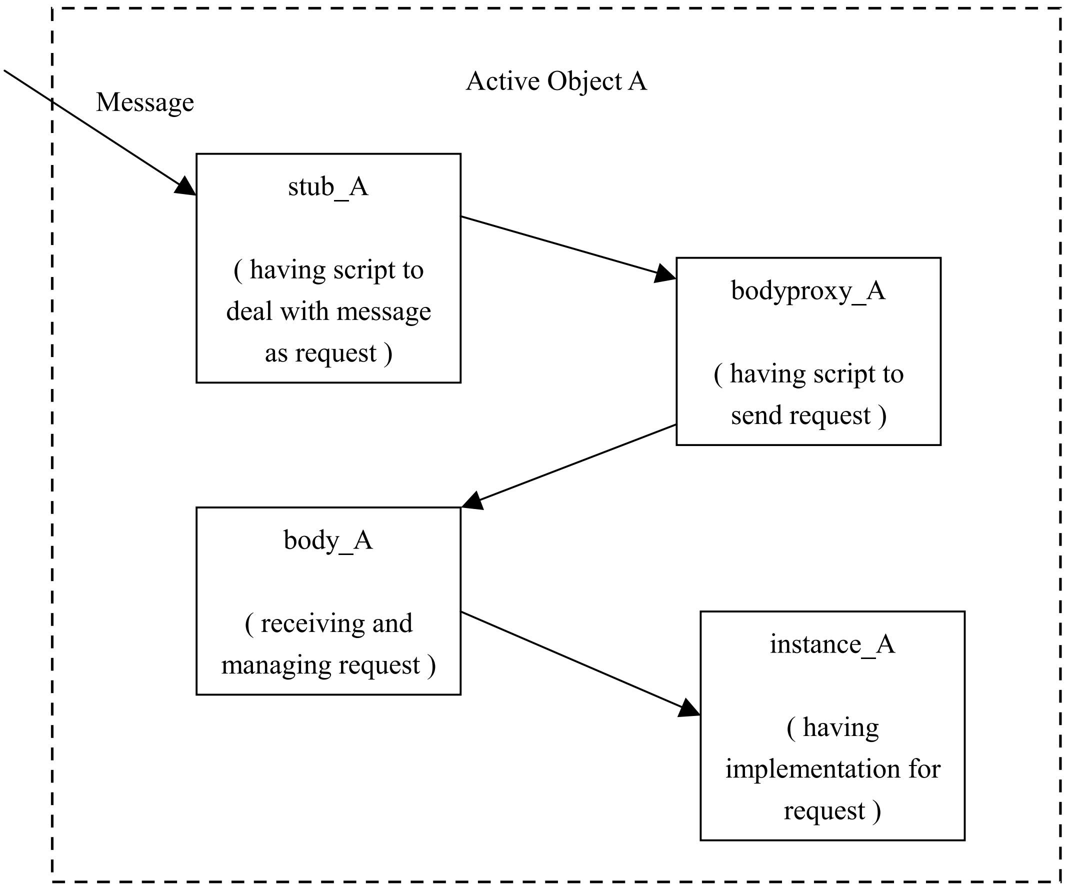

Active Object A
Message
stub_A
(having script to deal with message as request)
bodyproxy_A
(having script to send request)
body_A
(receiving and managing request)
instance_A
(having implementation for request)

Figure 17: Active Object Composition

There are three ProActive [39] interface for user to specify the management of received request to active object: InitActive, RunActive, EndActive. The standard behavior is to serve all incoming requests one by one in a FIFO order. User can adopt other order like LIFO by implementing RunActive interface, or passing a LIFO active object as the parameter to the creation of new active object.

In ProActive [39] every active object must be associated with a node, which could be thought as general and basic remote object with valid entry (RMI URL [42]) in RMI registry [42]. The default node is created automatically by ProActive on the local JVM when a first active object is created without a given node. Moreover, if no RMI Registry exists on the local machine, ProActive will automatically create one. If RMI registry exists, ProActive will automatically use it.

ProActive [39] uses future object and active object to get the asynchronous communication. The future object is the type of a refiable object that can not throw checked exception, and it includes three parts: stub_future, proxy_future, object_future. These parts are similar to the stub, bodyproxy and instance in active object in terms of function.

Suppose that object or component A calls the method of Active Object B with return the object of type V, the basic procedure is executed in 4 steps. The procedure could be graphically represented as below (solid arrow line means the message sending, and dashed arrow link indicates the object creator and its creation):

Step 1: Since stub_B is the subclass of class B, it inherits all the methods from class B, as well as the properties. However, active object's properties must be accessed through a public method, i.e. define public get and set method to do with property. Furthermore, when we create active object it returns the reference on its stub. Thus the stub_B receives the method call from A, and builds the methodcall object and passes it to bodyproxy_B by redefining all public methods to reify them through a generic proxy.

Note that stub_B cannot redefine final or non-public methods inherited from class B. Therefore, the calls to these methods are not reified but are executed on the stub, which may lead to unexplainable behavior. Meanwhile, getting or setting instance variables directly (not through a getter or a setter) must be avoided in the case of active objects because it results in getting or setting the value on the stub object and not on the instance of the class B.

Step 2: bodyproxy_B checks the type of return result, and then creates the future object of corresponding type. bodyproxy_B adds the reference of future object V (actually the reference of futureproxy_V in future object V) to the methodcall object and send methodcall object to body_B

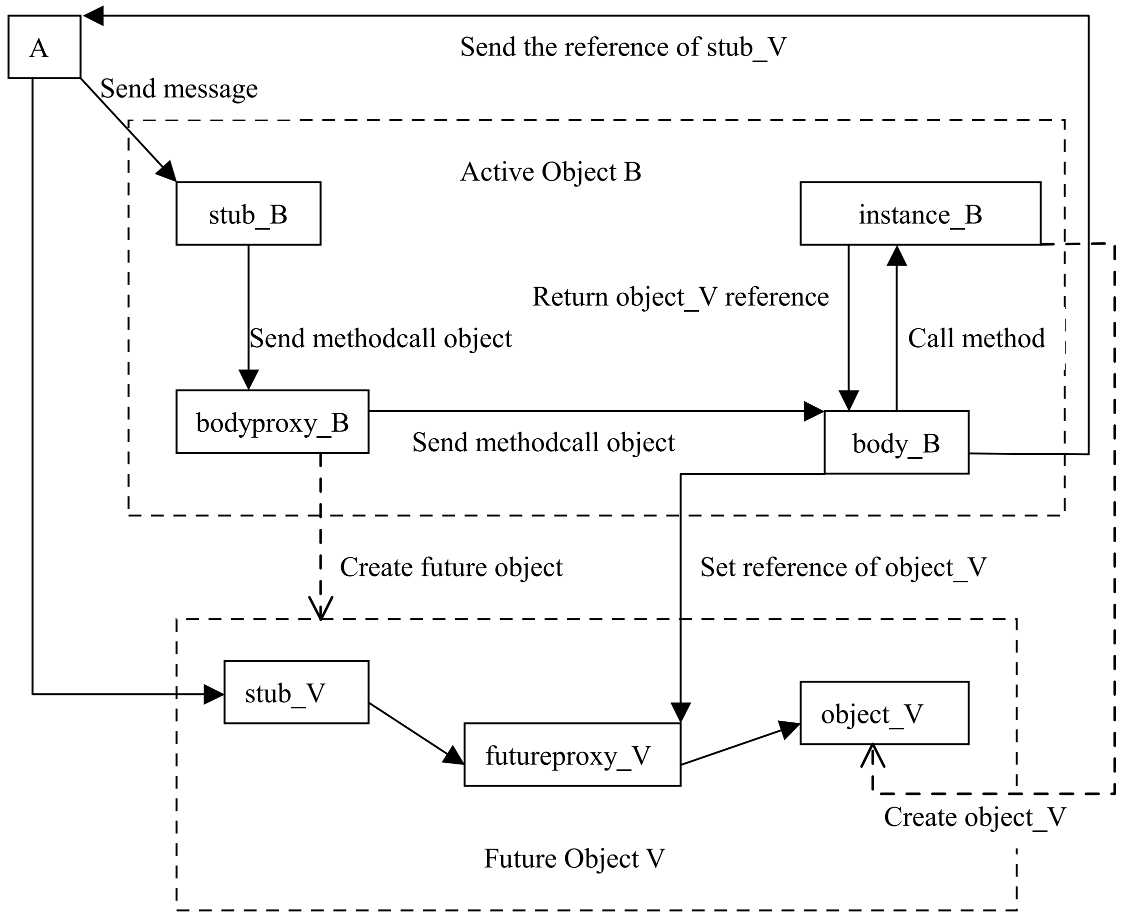


Figure 18: Asynchronous Call in ProActive

Step 3: body_B put the methodcall object to the queue, and return the reference of stub_V immediately to the object or component A. body_B would call the proper method in instance_B at last according to the default or user-specified service policy.

Step 4: instance_B executes method and create the object_V as a return result, and give the reference of object_V to body_B. Then the body_B provides the true reference of object_V to the futureproxy_V of future object V.

As illustrated in Figure 19, when object or component A wants to use the function of V, it sends the message to stub_V. Then stub_V wrap the message and send it to futureproxy_V, which invokes the real implementation in object_V. If the object_V is not available, A must wait for the result to be finished by Active Object B.

3.1.1 Component Characteristics of ProActive

As said before, ProActive [39] resulted from the application of the model that is structured in the multiple independent subsystems. Obviously, we can consider that every subsystem is a basic component, which is in charge of one basic function and communicate with each other.

As a matter of fact, due to the separation of active object and passive object, ProActive [39] is an implicit component-oriented programming language although it does not focus on the component programming. Every passive object is known only by an active object and this kind of passive object could be treated as the sub-object of that active object. Therefore one ProActive component could be constructed by one active object with any number of relative passive objects known by it.

Because ProActive [39] is used to write the program for the software, ProActive component is at the level of implementation and it has something different from general component. The most apparent point is that the interface of ProActive component is not as very structural and concentrative as general component. ProActive component is actually a set of objects, without uniform interface part. The interface of provided service by component is decided in the interface of active object, while the interface of required service is composed of the interface of active object and passive object.

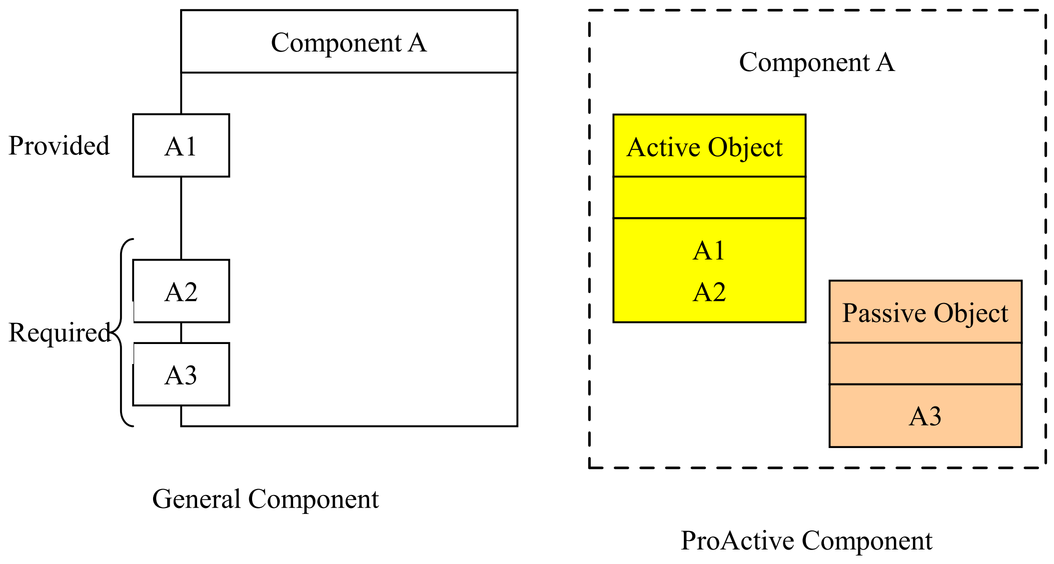


Figure 19: General Component and ProActive Component

In Figure 18, it is clear that ProActive component is a gray box, compared to the black box as general component. If one wants to know the interface of ProActive component, he has to firstly find the inside active object and passive objects of that ProActive component. Thus ProActive component is a kind of more primitive software unit according to the strictly encapsulated component with identified and uniform interface. In addition, since Java is the basic programming language to build ProActive component, this makes some restriction to ProActive component to be implemented in various environments.

The following section is to discuss the general but important characteristics of component mapped to ProActive specification.

Generality

1. Purpose

Following the application model introduced by concurrent Eiffel, it provides an active-object based framework to implement distributed, parallel, concurrent component programming.

2. Development theory

Decide the only one active object for each component in the application, and the active object could be called from inside or outside of component. The other object in each component is regarded as passive object, which can't provide the service to the objects outside the component.

3. Tools

A graphical-interface monitor called IC2D, which can show the status of active object and migrate it. For example user can drag and drop an active object in IC2D from a machine in Brazil to another machine in France.

4. Entities

Kinds

Each ProActive component is made of two kinds of object: active object (only one) and passive object (none or more). Active object could communicate with the passive object in its own component and other active objects outside the component. Passive object could only communicate with objects in the same component, or call the active objects outside the component.

Mixture

Implementation in pure java, and component is on top of java RMI layer.

Context

In fact all the components know each other though java RMI registry

First-Class

Active object and passive object are first-class citizens, and user-defined MetaObject protocol is supported for the reification of method invocation and constructor call.

Interface

1. Syntax

ProActive [39] has not keywords or abstract syntaxes to explicitly mark the interface of ProActive component in the program file. Although it uses java RMI layer to realize the concrete communication by active object in the components, all the RMI manipulation is encapsulated in its library. User only needs to write the class file in the common way and create or call active object using ProActive methods. In other words, ProActive simply provides the specific service with its own library interface for the programmer, not specifying the formal component interface.

2. Element

All the elements for the component interface are done in the class file like the standard Java programming. As a matter of fact user can't see the clear list of possible interface elements like procedure, channel, and so on. In addition, ProActive [39] regards all the public methods in the class definition as the usable elements to redefine for the active object.

3. Communication

ProActive [39] concentrates on the asynchronous communication model, and it makes the active object as a kind of message server. When a component calls a method in another component by sending a message to some active object, it will continue its execution until it needs the result of that method. This kind of interaction is called wait-by-necessity.

Each ProActive component could only have the reference of stub of the active object in other components, and the stub of active object is responsible to send the method call through a generic proxy in active object. Therefore the ProActive component interface is represented by the stub of active object, if from the view outside ProActive component.

Attribute

All the attributes of ProActive [39] component are done in the class file, and user has to implement public get and set method (as the element for interface) to operate the attribute of the active object. In order to initialize the attribute in the constructor of active object, the attribute argument is asked to implement the interface Java.io.Serializable.

Life Cycle

The life cycle of ProActive [39] component depends on the RMI registry. When ProActive creates an active object of a component, it will create a RMI registry or use an existing one, and register the component with a default or specialized node. Then other component could send the message to this ProActive component by the RMI URL. If it is unregistered from java RMI registry, or RMI registry is unloaded for some reason, the ProActive component is out of the interaction with other ProActive components in the application.

Dynamic Behavior

There are a lot of different explanations on the dynamic behavior of component, but here it means the context-specific change of component behavior in the run-time. In this way, ProActive [39] may support the dynamic behavior of the component.

ProActive gives the programmer some interfaces and classes to deal with all the method call to active object before active object accepts to execute the corresponding implementation. Thus the special communication protocol and behavior control could be easily done on the ProActive component without changing the source code of concrete function. In fact, the dynamic behavior of ProActive is to separate the control decision from behavior implementation.

Properties

The property of ProActive [39] component is mainly expressed by active object. Active object could give the dynamic and service-specific property to its interface through its RMI encapsulation and message-queue management. The interface part of active object could be local or remote. Active object could automatically migrate itself among the different JVM, and set the relative deployment by XML descriptor. User could specify and change the service policy on the message queue of active object according to the application context in the run-time.

More information about ProActive [39] could be read in the appendix A.

3.2 Translation from Asynchronous Model to ProActive

Since ProActive [39] is a programming language implicit to support component-based software development, it is possible to use the component model gotten from the architecture design phase to implement it in ProActive. The component model here used includes the basic or structural component with abstract interfaces and functions, and the main asynchronous communications between the components. In addition, the component model should also have the relative deployment for the system running.

Although the implementation for the component model by ProActive [39] is visually a development process from system specification to program code, even the most formal specification of a single function could has various writing of code in terms of individual programmer and relative tools. A more effective method to explain how to realize the component model by ProActive in a general way is to use Model-Driven

Architecture (MDA) [44] approach. MDA is to translate the component model to ProActive implementation at a higher abstraction level. This kind of translation only concentrates on the translation of most common and important aspects in the component model not the details in fact.

3.2.1 MDA Introduction

Because every development of software system has to depend on the concrete technology to make the real value for the client, there is a huge market of software tools and development theory like C++ and Java [29]. People often spend much time to choose the proper techniques to implement their respective work. The relationship between the system functions and corresponding realization is many-to-many. That's to say, one single function could have different solution through different techniques, and vice versa.

In order to solve the contradiction between the abstract design and concrete technology, MDA [44] is used by OMG (Object Management Group) to connect the different implementation with the similar software architectures, and these architectures are based on a highly abstract business model [8, 9, 10, 11]. Thus MDA is model-centric development methodology to automatically translate the base model to various specific models (architecture) on the concrete platform or technology, with some transformation rules. Then the specific model is used to generate the formal program code by some kind of parser, and finally the formal program is completed and validated by the relative developer as usual. The base model is called Platform-Independent Model (PIM), and the specific model is called Platform-Specific Model (PSM) [44, 8, 9, 10, 11].

The separation between PIM and PSM done by MDA [44] is to reduce the gap between the architecture designs affected by different technology for the same behavior or function. Therefore MDA leads to more design reuse in the development. Meanwhile, MDA tries to get the automatic translation from abstract business model to all kinds of specific development technology and even code generation in terms of language [8, 9, 10, 11]. The automation from model to implementation also decreases the cost and time in the detailed design and implementation phase of the software life cycle. MDA help people to pay more attention to generic and abstract system modeling without technique support, and also to technology-dependent rules for automatic mapping.

Since MDA [44] works beyond the level of component and other middleware platform, it could easily create the abstract model mapping to component-based architecture. In fact, a MDA application should include one base generic model as PIM, and several specific models as PSM like .NET [37], EJB [36], and their concrete realization. In this case, the interoperability between the several specific technologies involved in a MDA development is obviously high and clear. The MDA tools can use base model specification and relative transformation rules to construct the corresponding relationship between the implementation of one PSM to another [10, 11]. In other words, the link of service or component in PIM is mapped to the interoperation among the systems of PSM due to different implementation.

The core standards used to represent PIM is Unified Modeling Language (UML) [34], Meta Object Facility (MOF) [45] and the Common Warehouse Metamodel (CWM) [45]. In this paper, I adopt the UML-like diagram to illustrate the some translation rules from general component model (PIM) to ProActive [39] Model (PSM), since UML language is very common in the system-modeling domain.

More detail about MDA could be read in the appendix B, and it is being developed and formalized to be another standard approach especially for component-based software development.

3.2.2 Translation Rules for ProActive

ProActive [39] is a Java-based programming language with featured asynchronous communication and implicitly supports component-based system development. The translation from asynchronous component model to ProActive implementation is really direct. In fact there are some ongoing projects that are using ProActive to produce component software

Using MDA [44] approach, this paper proposes some basic rules to give the insight how the asynchronous component model could be realized in ProActive [39]. The rules are represented as PIM-to-PSM in UML-like style, because UML [34] is the most popular modeling language for the system development. UML is also the modeling standard adopted by MDA of OMG.

1. Transformation on class

Besides the standard object in ProActive [39] as other OO languages, there is active object introduced in ProActive for the local or remote communication with other object. One basic independent component, which has message exchange with other components, could be mapped to a class of active object in ProActive. We can use some kind of class diagram [6] to show it as in Figure 20.

Although there seems no change in the transformation between the component and the class of active object, some special requirements are asked by ProActive [39] to ease the encapsulation and RMI [42] operation behind the active object. User has to put a non-argument constructor (ComponentA in Figure 20) in the class of active object. The non-argument constructor would be called by the stub created in ProActive, otherwise the stub calls the constructor with argument and use the argument for the stub not for the real object. The argument of constructor normally should be serializable, which means that the argument belongs to a class implementing Java.io.Serializable interface. The class of active object must be declared as a public and non-final class in ProActive, so the active object can be rebuilt by ProActive.

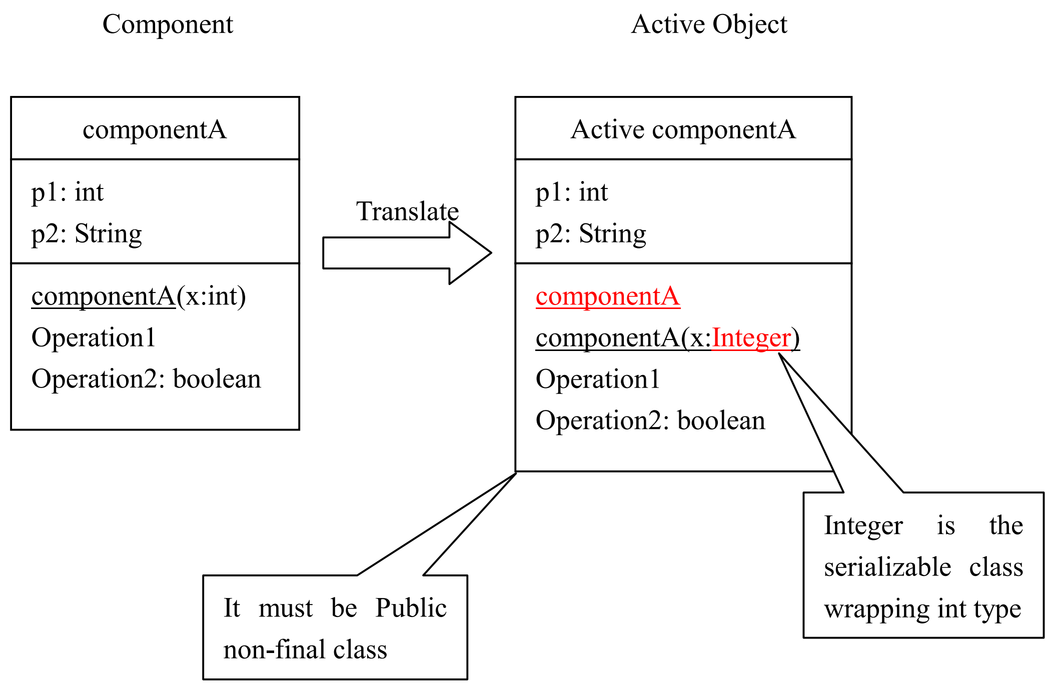


Figure 20: Translation from Component to Active Object

If the component has the special service policy other than FIFO to work with the received message, user have to implement ProActive [39] interfaces or create active object with argument which is another active object with same policy. Using three ProActive interfaces (InitActive, RunActive, EndActive) is more common to get user-defined management, and it is easier to be done in our translation, shown in Figure 21.

In Figure 21, as declared in class of active object, the InitActivity is the function defined in interface ProActive.InitActive. InitActivity is used to initialize the management of received message, like the constructor of java thread class. The runActivity is the function defined in interface ProActive.RunActive. runActivity is used to manage received message by active object, like the run() method of java thread class. Then The endActivity is the function defined in interface ProActive.EndActive. endActivity is used to do something when finishing the management of received message, like the stop() method of Java thread class.

In addition, there may be some compound component that is composed of several sub-components, with some reference relationship between each other. In this case, only the sub-component that accepts the message from outside need to be transformed to the class of active object, and others are treated as the class of passive object (Java object) in ProActive [39]. Thus the transformation from compound component to ProActive component only depends on the interface of sub-component in the compound component, no matter the relationship between sub-components.

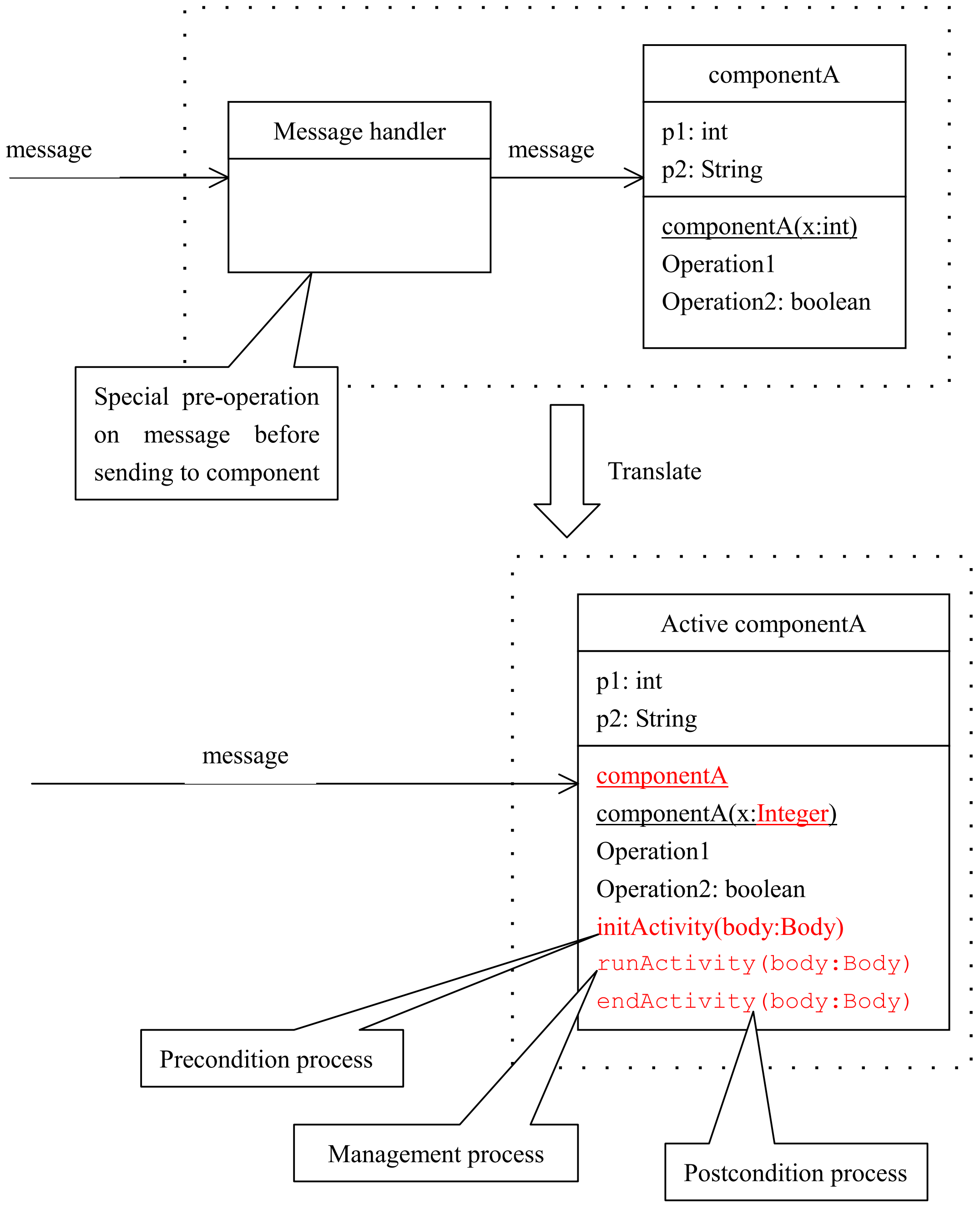


Figure 21: Message Management Translation to ProActive

The Figure 22 graphically shows the translation on compound component. The general component componentA has three sub-components: SubComponentA1, SubComponentA2, SubComponentA3. SubComponentA1 has the aggregation of SubComponentA2, SubComponentA3. Since only SubComponentA1 can receive the message and give service to other component, it is translated to active object (Active SubComponentA1) in ProActive component. SubComponentA2 and SubComponentA3 becomes the passive object (Passive SubComponentA2, Passive SubComponentA3) in ProActive component.

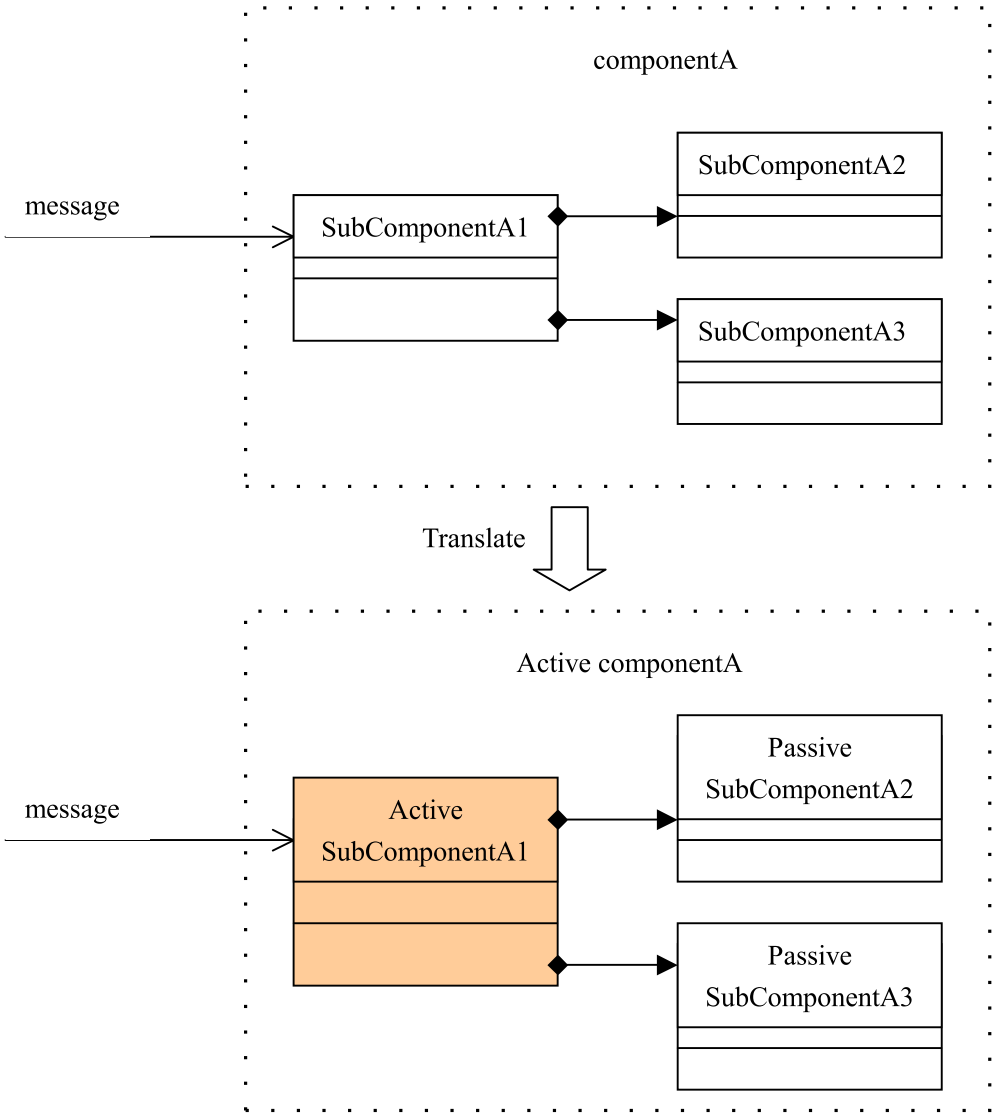


Figure 22: Compound Component Translation to ProActive

As for any structural component, it could be decomposed to a set of relative sub-components. As long as we extract the sub-components exposing the interface of provided service to other components, each sub-components of this type should be translated to active object. Then according to the relation in structural component, the sub-components not exposing their interface of provided services, are translated into passive objects to be linked with active objects. Thus one structural component be translated to one or more ProActive components, and each ProActive component has only one active object and any number of passive object.

Different from general object-oriented programming language, ProActive [39] uses its own primitive method to create the active object; in other words, we must use newActive or turnActive method in the org.objectweb.proactive.ProActive class. The newActive method creates an active object by the special instantiation from a class. turnActive method makes an active object through reconstructing the existing object that is created by the classic OO keyword new coping with the class, like following:

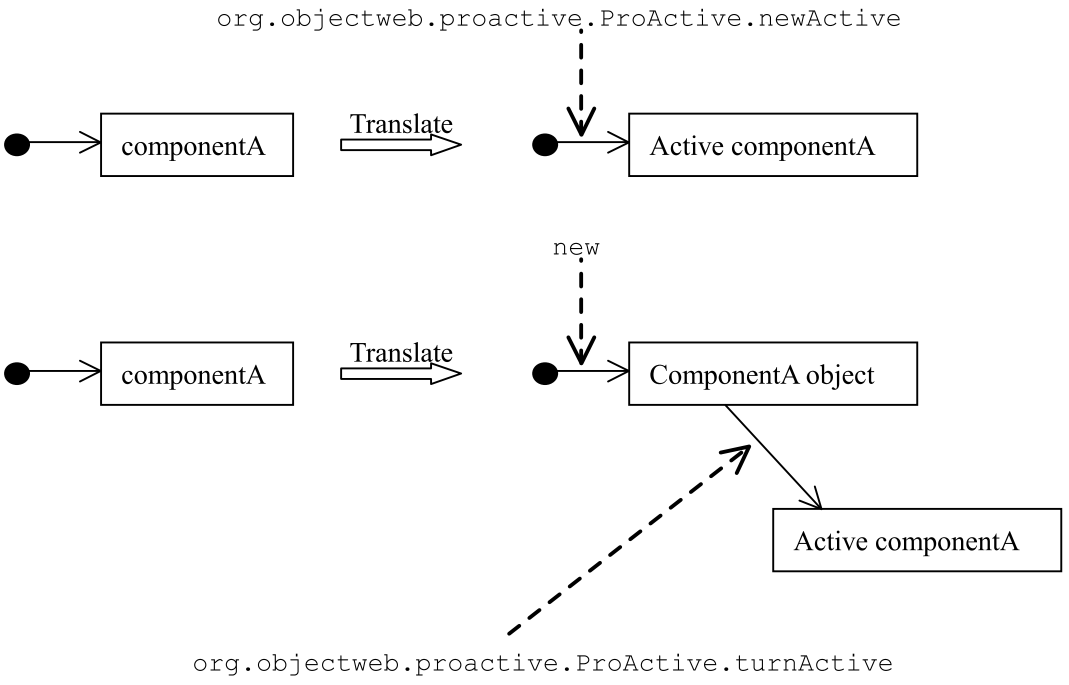


Figure 23: Creating Object Translation to ProActive

The choice between both newActive and turnActive has no difference for the function of active object, and it only affects the detail of programming. For example, newActive has the limitation on the arguments of class constructor that the argument has to be serializable, as shown in Figure 20. On the other hand, turnActive is flexible to encapsulate any Java object to be an active object, and thus it could be more useful in special context of programming. As for the rule of PIM-to-PSM, I think that newActive should be put as the default method for the creation of active object, since it is more close to the standard object creation in terms of semantics, and more simple for automatic translation. Nevertheless, turnActive is the alternative way to revise the PSM model for some reason.

2. Transformation on object call

Since ProActive [39] distinguishes the active object from the standard java object (i.e. passive object), the method call to the active object is based on the RMI mechanism. This means that the message sent to active object is always operated as a remote procedure call in Java [29] programming. Therefore, the method call to active object and passive object is different in ProActive. It is transformed like following, supposing that SubComponent1 is translated to an active object, while SubComponent2 is translated to a passive object in Active Component2.

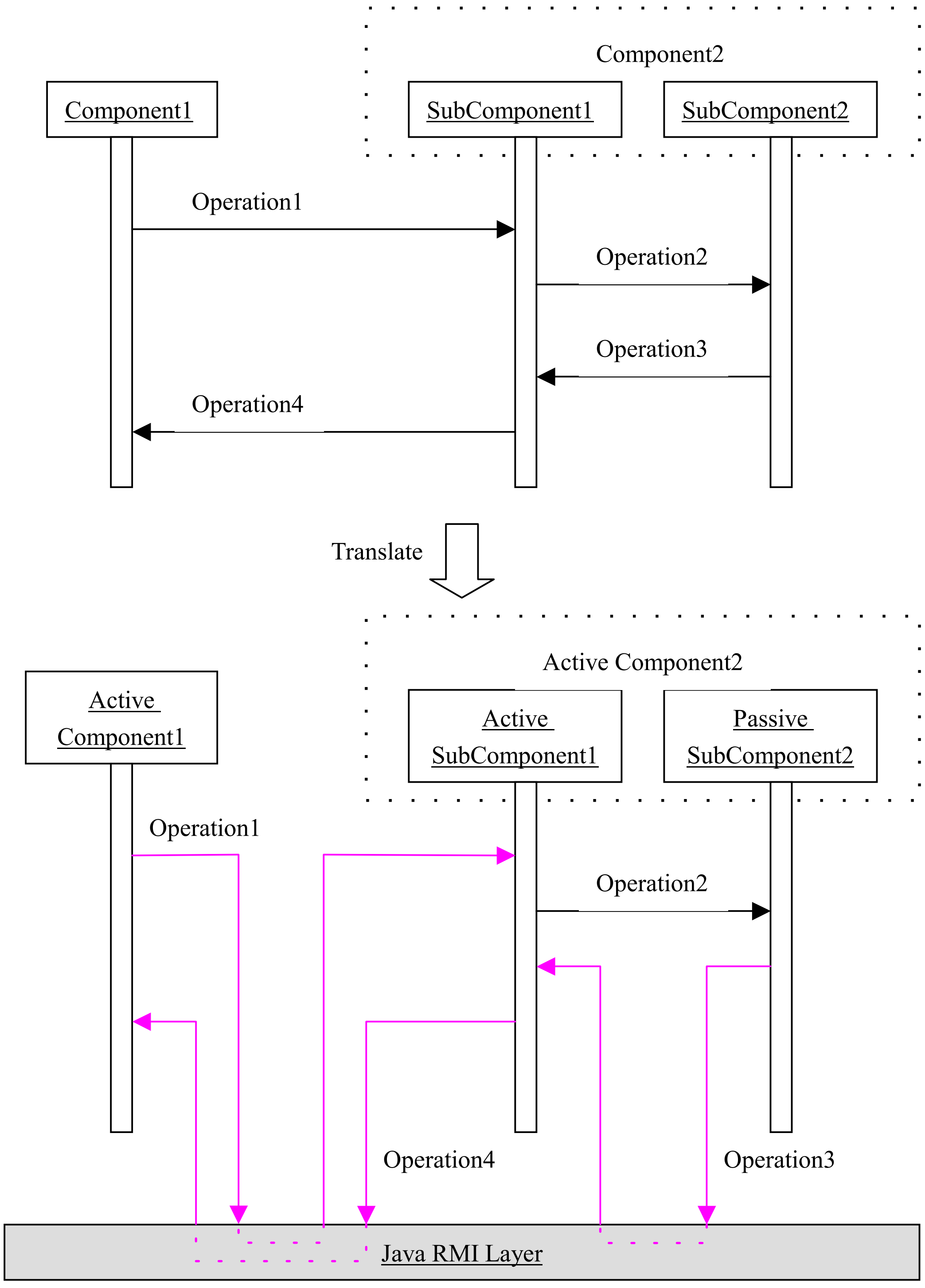
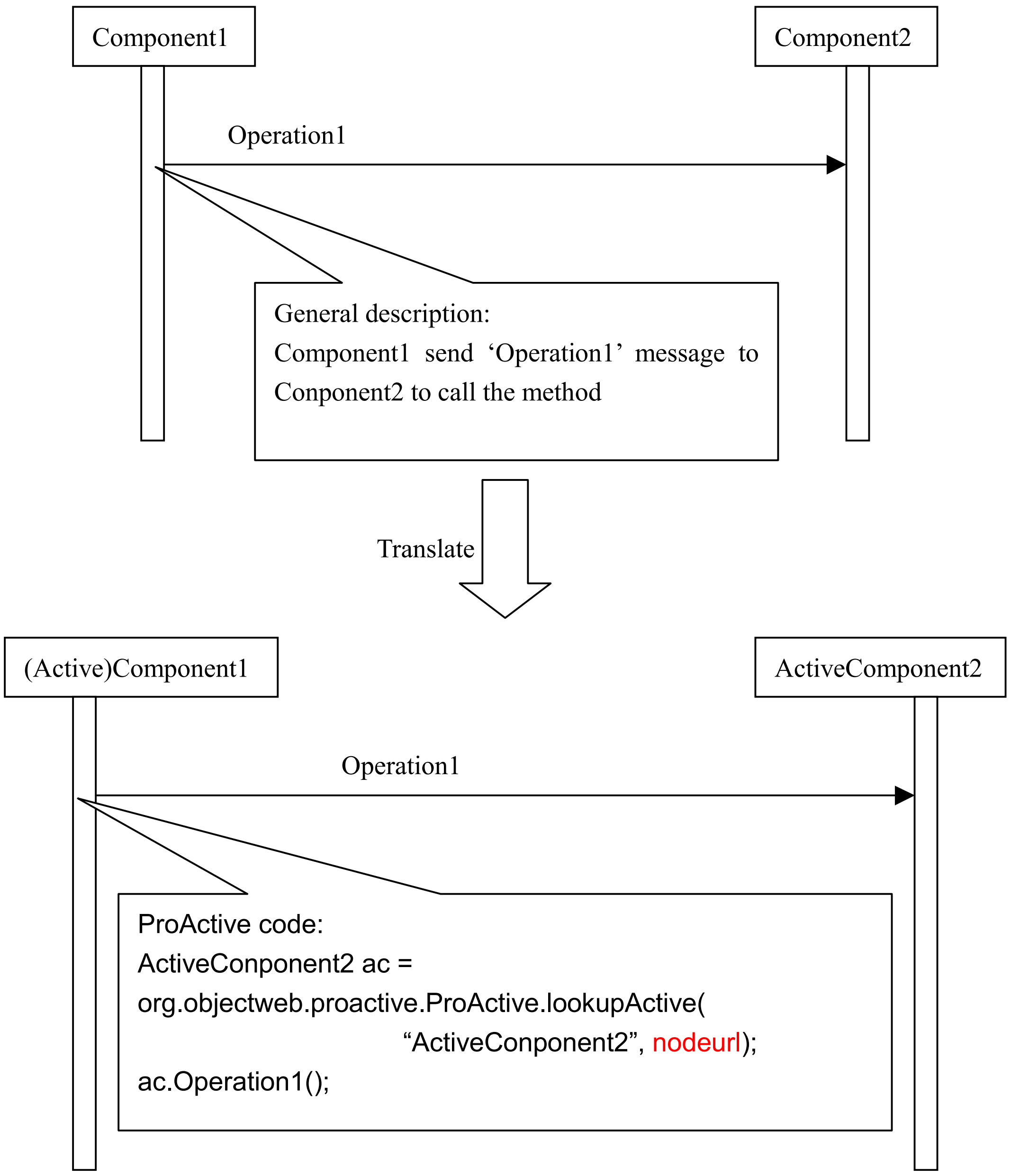


Figure 24: Generic Method Call Translation to ProActive

In order to implement special method invocation to active object through programming, ProActive [39] provides some primitive operations for the method call to hide the detail of underlying RMI [42] manipulation. As a matter of fact, if a component or an object wants to call an active object, it must know the RMI address of that active object. Every active object is registered into RMI registry when it is created using newActive and turnActive method. Therefore, the actual transformation of method call to active object on the source code level (or text representation) is like following, supposing that Component2 is translated to an active object.



```
ProActive code:
ActiveConponent2 ac =
org.objectweb.proactive.ProActive.lookupActive(
                        "ActiveConponent2", nodeurl);
ac.Operation1();
```

Figure 25: Method Call Translation to ProActive Code

In Figure 25, there are two arguments in lookupActive method. The first one is the class name of active object, and the second one is the address of node related to active object. The node has an entry in RMI registry.

Asynchronous call is the basic feature of ProActive [39], which means that when a method is called on an active object, the fake result (future object) is returned immediately to the caller. Only if the result needs to be used, the ProActive program would wait until the real result is available. This kind of asynchronous strategy is so-called wait-by-necessity. Thus suppose that the work flow in an asynchronous component model is generally like following:

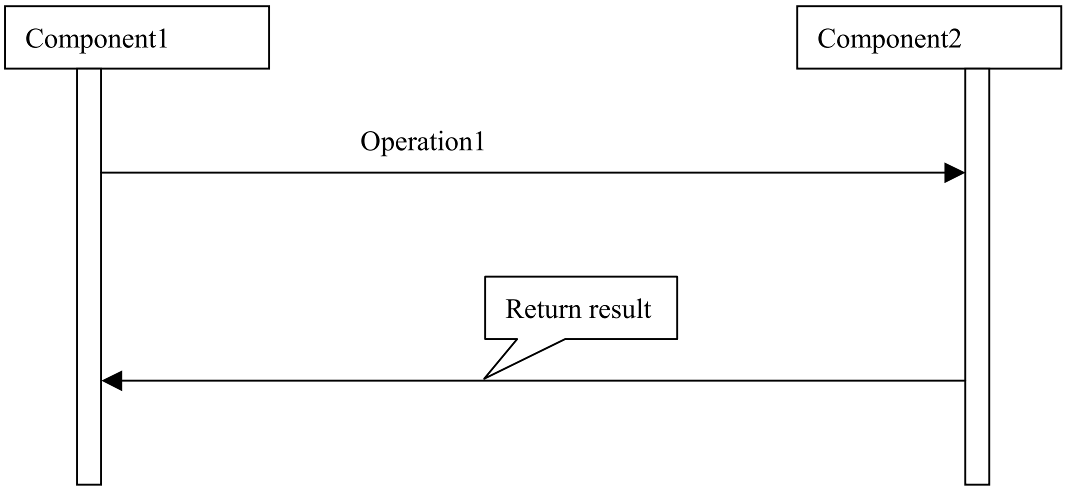


Figure 26: General Asynchronous Call of Component

The Component2 is translated into active object to handle asynchronous communication. If the result of Operation1 is returned after Component1 reaches the statement where the result is needed, the ProActive workflow looks as below:

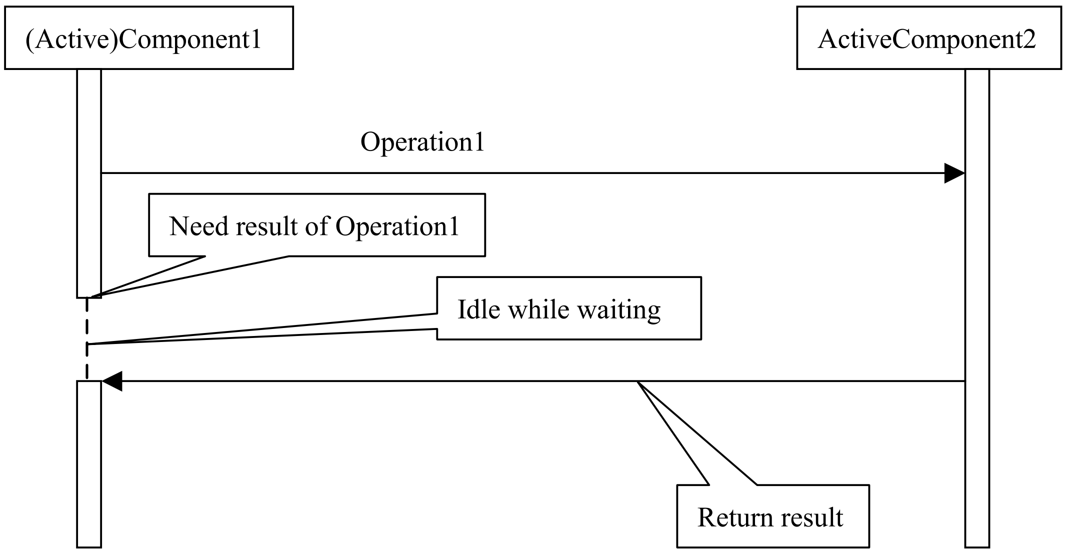


Figure 27: Asynchronous Call Translation to ProActive

If the result of method call is returned before the caller reaches the statement where the result is needed, or there is no result of method call, the asynchronous call in general component model is same as in ProActive [39]. However, Due to 100% compliance to the standard java language, there are some restrictions on the asynchronous call in ProActive:

1) Final classes cannot be the return type of asynchronous call
2) Same thing for primitive type like boolean, char
3) Same thing for classes without non-argument constructor
4) Same thing for classes throwing checked exception.

Meanwhile, if an object in proactive is the argument of a method, the class of this object must be serializable unless this object is active object. Thus the transformation on class of active object with asynchronous call is like following:

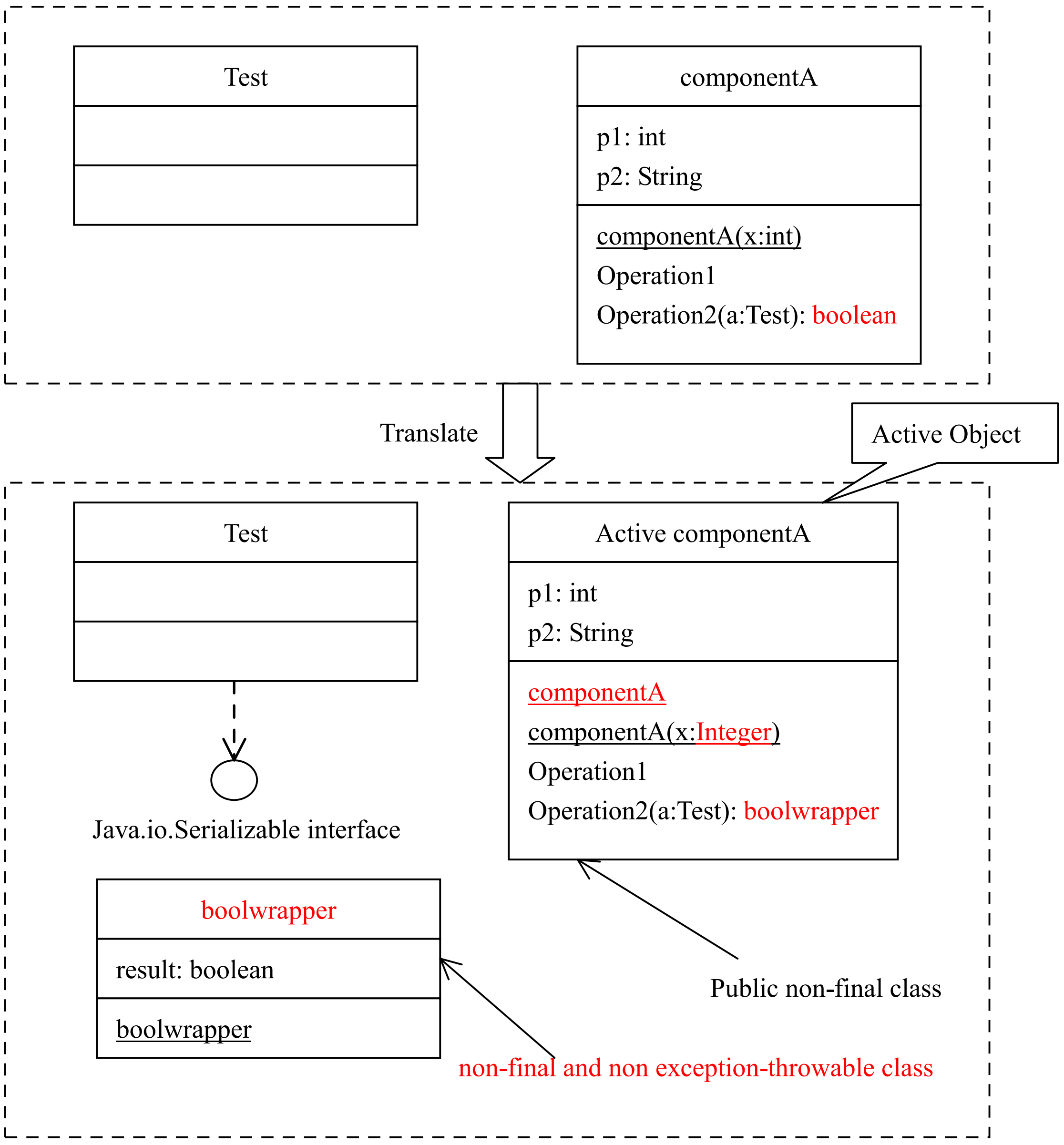


Figure 28: Asynchronous Component Translation to ProActive

On the other hand, if user wants to keep the synchronization way of method call to an active object in ProActive [39], user can modify the return type of the method according to the previously listed restriction for asynchronous method call. We can also use ObjectForSynchronousCall class, given by ProActive, as the return type for the method with no return result in the synchronous call. The transformation is like following, supposing that Operation1 and Operation2 in component componentA need to be called synchronously:

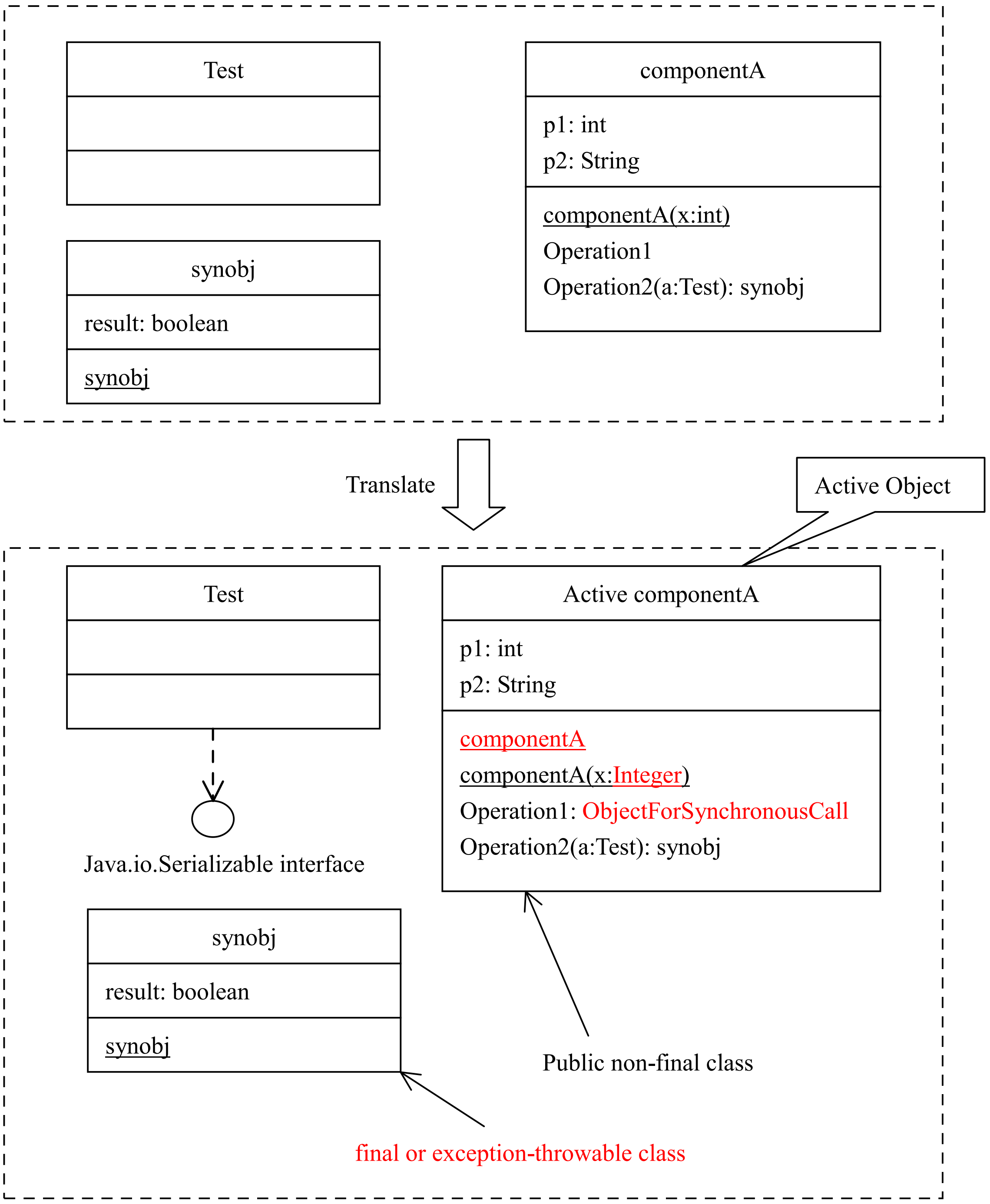


Figure 29: ProActive Translation of Component with Synchronous Call

ProActive [39] uses node to deploy the active object in the runtime environment, and here node is an object which offers the minimum but essential services to interact with a given JVM [43]. Every active object is attached to a node which indicates the JVM where active object lives. Since the node URL identifying node object is stored in

Java RMI registry, node URL follows the format of RMI protocol. Therefore, the transformation on deployment of component is like below, and the dashed arrow link means message sending in the system:

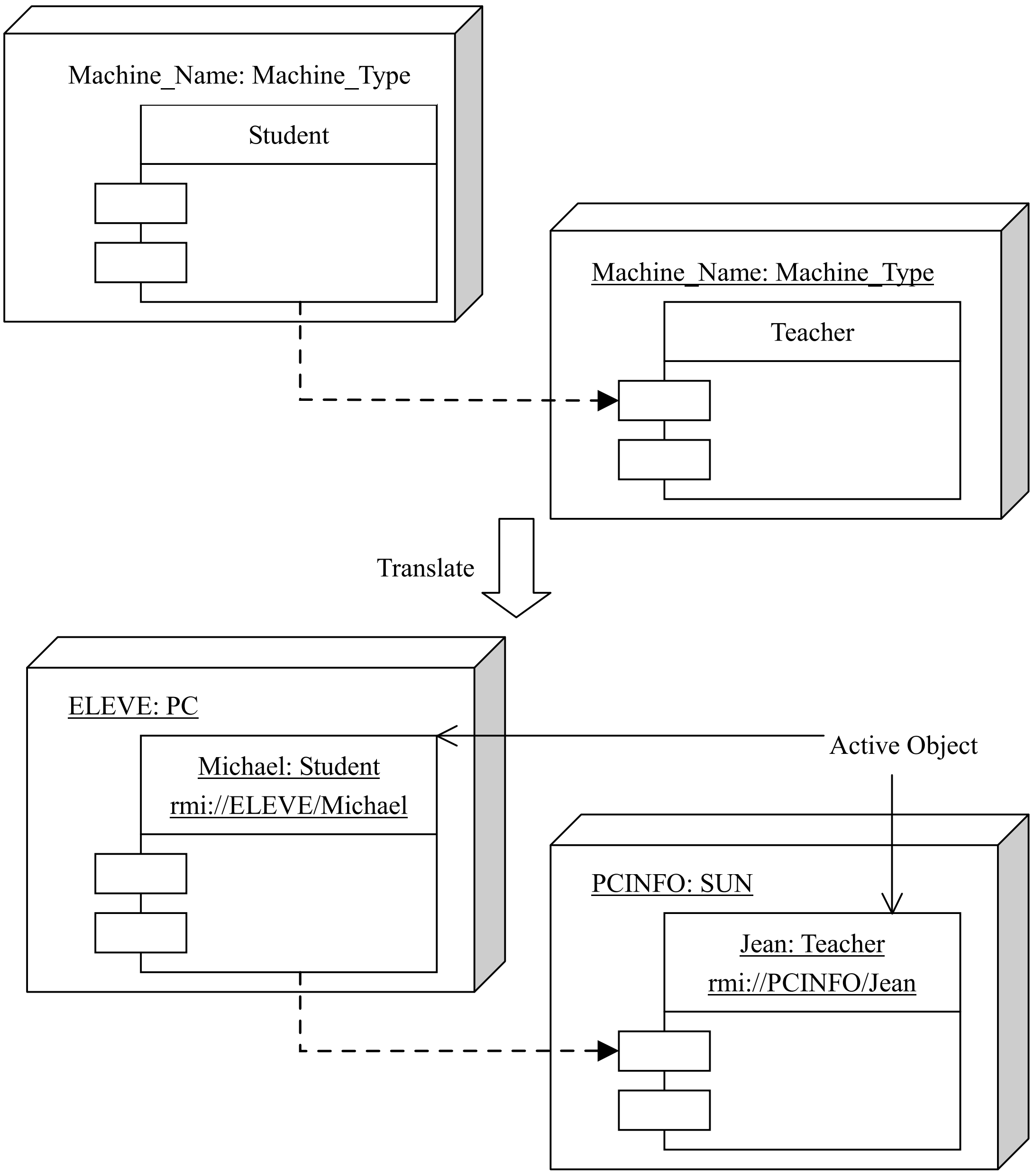


Figure 30: Deployment Translation to ProActive

In Figure 30, the general deployment model of component is translated into concrete deployment model in ProActive [39]. The ProActive active object named Michael with type Student has the corresponding node identified by node URL rmi://ELEVE/Michael. Another active object Jean has its node URL rmi://PCINFO/Jean, and thus the developer could set those node URL in ProActive deployment file for the active object related to component.

3.3 Example: Flight reservation system

As an experiment of ProActive [39] implementation, I used ProActive to realize a simple flight reservation system [7]. The component model for this flight reservation is very easy to be shown as below:

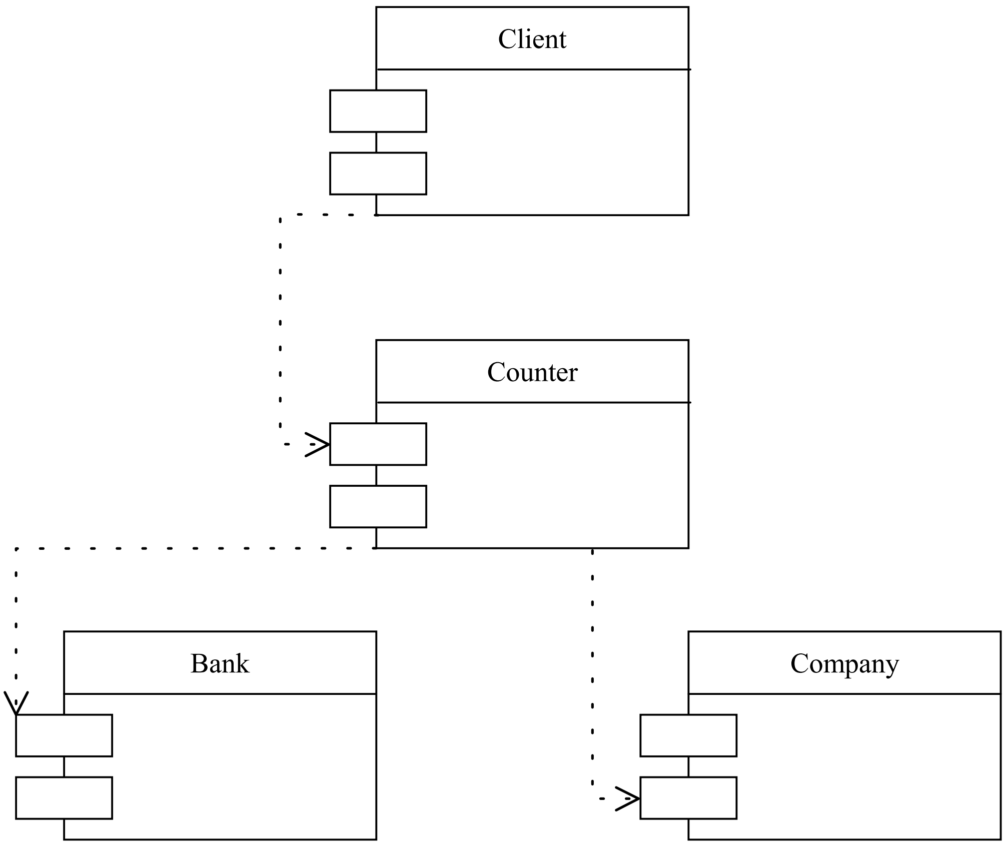


Figure 31: Component of Flight Reservation System

As seen in Figure 31, there are four components in flight reservation system. Client component asks the Counter component for the flight ticket. Before Counter component returns the ticket to Client, it has to ask the Company component if there is available seat in plane. According to the result from Company, if there is still seat, Counter component would ask the Company to book the seat and send request to the Bank account to pay the flight ticket from client's account. Then the Counter component gives the ticket to the Client. If there were not seat, the Counter component would return failure to the Client.

This simple flight reservation system obviously contains some asynchronous communications between the independent components. For example, when the Client component sends the message for the ticket, it could continue its work without waiting for the result of its call. Booking seat by the Company component and payment by the Bank component could be done concurrently if there is seat in flight. In other words, the Counter component should sends booking message and payment message one by one in an asynchronous way. It is not necessary to waiting for the end of booking seat and then do payment of flight price to the Bank component.

Therefore flight reservation system is naturally a distributed, parallel, concurrent, asynchronous software system, ideal for the ProActive implementation. The transformation rules are already applied to the component in flight reservation system, for example the general class representation of Bank Component is like below:

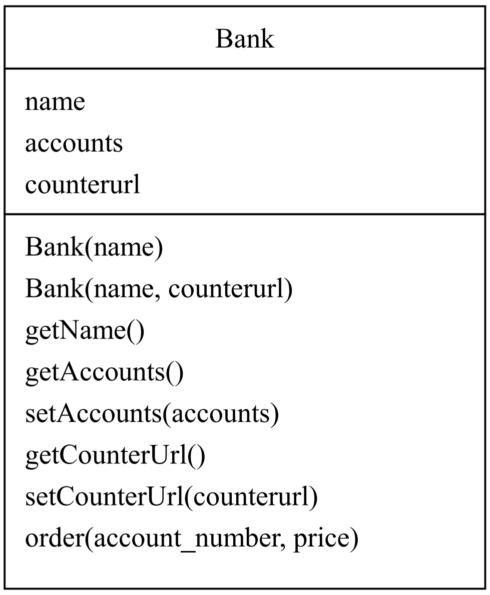


Figure 32: Bank Component of Flight Reservation System

The actual implementation of Bank component is to program it as the class of active object in ProActive [39], coding like Table 1.

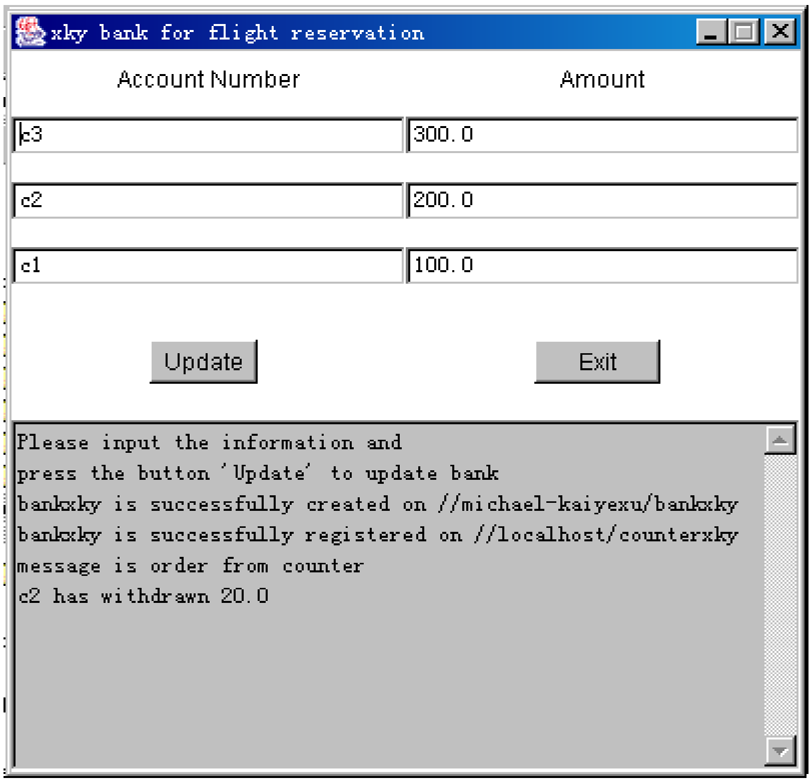
```
/*
  * author Kaiye Xu,
  * Bank.java
  */

package org.objectweb.proactive.xky.flightbooking;
import java.util.Hashtable;

/**
  * xky bank to manage the bank account in flight booking system
  */

public class Bank implements org.objectweb.proactive.RunActive {
```

```
private String name;
private Hashtable accounts;

private String counterurl = "//localhost/counterxky"; //Counter component address

public Bank(){} //necessary to ProActive active object

public Bank(String name) {
   this.name = name;
   initAccounts();
}

public Bank(String name, String counterurl) {
   this.name = name;
   this.counterurl = counterurl;
   initAccounts();
}

//--Create some new account with account number and amount
private void initAccounts() {
   accounts = new Hashtable();
   accounts.put("c1", new Float(100));
   accounts.put("c2", new Float(200));
   accounts.put("c3", new Float(300));
}

public String getName() {
   return this.name;
}

public Hashtable getAccounts() {
   return this.accounts;
}

public void setAccounts(Hashtable accounts) {
   accounts.clear();
   this.accounts = accounts;
}

// Return the RMI address of Counter component as an active object
public String getCounterUrl() {
   return this.counterurl;
}
```

```
  public void setCounterUrl(String counterurl) {
     this.counterurl = counterurl;
  }

  /**-- substrate amount of price from account identified by account_number
   *
   *   The return type boolobj is a serializable class which has the instance
   * variable with boolean type
  */
  public boolobj order(String account_number, Float price) {
     System.out.println("message is order from counter");

     Float amount = (Float)accounts.get(account_number);
     if(amount != null) {
        if(amount.floatValue() > price.floatValue()) {
           accounts.put(account_number,
                 new Float(amount.floatValue() - price.floatValue()));

           System.out.println(account_number + " has withdrawn " + price);
           return new boolobj(true);
        }
     }
     return new boolobj(flase);
  }

  //--this function is the implementation of rg.objectweb.proactive.RunActive
  //-- it makes Bank active object serving in FIFO order
  public void runActivity(org.objectweb.proactive.Body body) {
     org.objectweb.proactive.Service service =
                    new org.objectweb.proactive.Service(body);

     service.fifoServing(); //FIFO order to serve the request
  }
}
```

Table 1: ProActive Program for Bank Component

The other program files including Client.java, Counter.java and Company.java also follow the transformation rules described above to implement the functions of Client component, Counter component and Company component. Meanwhile, in order to make the user interface more comfortable, every component is attached to a java window to show the state of component. For example, the bank window is like below:

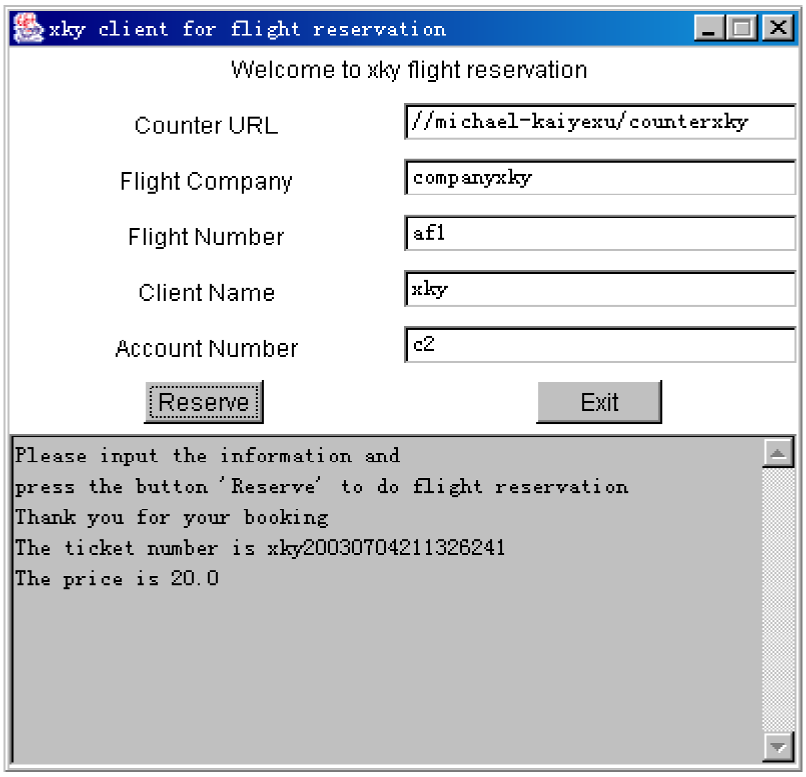


Account Number	Amount
c3	300.0
c2	200.0
c1	100.0

```
Please input the information and
press the button 'Update' to update bank
bankxky is successfully created on //michael-kaiyexu/bankxky
bankxky is successfully registered on //localhost/counterxky
message is order from counter
c2 has withdrawn 20.0
```

Figure 33: ProActive Window of Bank Component

The Normal result of client request for flight reservation is like below:


xky client for flight reservation
Welcome to xky flight reservation
Reserve
Exit

Counter URL	//michael-kaiyexu/counterxky
Flight Company	companyxky
Flight Number	af1
Client Name	xky
Account Number	c2

```
Please input the information and
press the button 'Reserve' to do flight reservation
Thank you for your booking
The ticket number is xky20030704211326241
The price is 20.0
```

Figure 34: Client Request in Flight Reservation

All the flight reservation system has been implemented and tested with ProActive ver1.0.1 [39] and with JDK 1.4.1 [29].

Though the work of this master thesis only implemented the flight reservation system with single client, it is easy to extend the system to handle N clients at the same time. When N clients concurrently send the message to ProActive component Counter (active object actually), the Body of Counter would automatically stores all the message objects to the message queue. Then the Java thread of Body would always monitor the condition of message queue, and pick out one message at some time according to the service policy of Body. The method call to ProActive component Company and Bank is included in one process of Counter to reserve the flight seat. During the time the Counter deals with each message from client, the result by Counter is never returned until the action by Company and Bank is done; In other words, Counter has to wait for the feedback from Company and Bank before it begins to operate the next message from a client. Thus the requests from N clients are served ultimately in a sequence order. Meanwhile, N clients could continue their work in an asynchronous way after they send the message to book the seat.

From the above explanation of transformation rules and example from general PIM to ProActive-specific component model (PSM), we can see that the change is direct and easy. However, those are just the basic work to connect MDA [44] fashion with the ProActive [39] implementation. As ProActive is being developed to be a standard java infrastructure, it would be a more powerful tool for the development of network-based component software with asynchronous communication.

CHAPTER FOUR

ANALYSIS OF ASYNCHRONOUS COMPONENT MODEL

The CBSE [13, 17, 18] brings some big changes to the traditional software engineering during the recent years. One of the most important things is that CBSE separates the abstract architecture design from concrete implementation at a higher degree. The system developers could obtain more time to design an applicable architecture based on reusable component, while they just often need to find the existing component proper to be integrated into the system.

However, although people can reduce the time and effort to produce the new program or component and test it, there appears another boring problem to people. It is about whether the existing component is good enough to the system, in terms of functions and environments, to be integrated into the new system.

As a matter of fact, the system requirements and the reusable component market both mainly affect the decision in the architecture design stage. Due to the variation of both requirements and market along with the time, the related design decision and the architecture of component would be modified again and again in the software life cycle. Therefore this leads to leave many instable even conflicting elements in the component model, and then the elements would proliferate through the whole software development. From CBSE perspective, even if there are all the well-adoptable existing components by hand, only the composition not the inside components decides the software quality of the component-based system. A component is just reusable function unit under the demand of software architecture.

Undoubtedly, there should be some theory and tool to analyze and protect the component architecture from the dangerous change for some reasons. As the common methodology by people for other problems in the science or society, the first step to analyze the component model is to represent it in some way like text or slides. According to the description of the model there would be some specific solution to give or validate the properties of component model. The solution of analysis could also be used at the design reuse level.

There have been already some Architecture Description Languages (ADL) used to formally represent the architecture of software, like C2, Rapide, Wright [1]. ADL generally gives the explicit specification of component, connector (a kind of component which is only in charge of communication, as message medium), and architecture configuration [1]. The interface of component and its semantics is also the important part expressed clearly by ADL. Meanwhile, some component-specific ADL like SOFA [49], gives uniform control and management on how to deploy and update the component dynamically. SOFA also eases the task to create a reasonable hierarchy for component model. However, the current ADLs seem to focus on respective domain for the software architecture of domain-specific application. There is no general support by ADL to analyze the asynchronous communication in the component model. Thus the asynchronous model is still an open issue for researcher to get more insights about it.

In this paper, I use ATAG (Asynchronous Graphic Abstract data Type) [7] to represent the component model. Then according to ATAG model, I propose an algorithm to analyze the message amount in the buffer of component. The buffer is called mailbox in my algorithm, and it is actually the classic way that asynchronous component model is simulated through synchronous model, as said in section 2. The implementation and test is presented at last.

4.1 ATAG

Different from other kinds of ADL, ATAG concentrates on the model with messaging between the components and accordingly change of the component state. It results from the previous study on Korrigan and GAT (Graphic Abstract data Type) [2, 3, 5, 19, 22], both of which is used to give a complete specification of a component-based mixture system. GAT approach is linked to the synchronous component model. Because ATAG only pays attention to the dynamic behavior of component model with asynchronous communication, it just inherits the Symbolic Transition System (STS) [2, 3, 5, 19, 22] in Korrigan and GAT. STS uses some abstract symbol and figure to feature the event, message, activity and relative component states; in other words, STS is simply a finite state chart with additional description.

The basic unit in ATAG [7] is the component, and it is graphically expressed as the view from outside, like below:

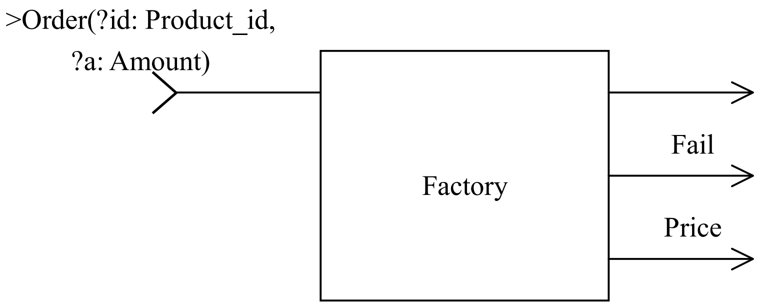


Figure 35: Component Interface of Factory

In Figure 35, the component is a box with name label, and there is one input pin on the left of box, and three output pins on the right.

The input pin could be considered as the port for receiving messages to have component provide corresponding service to it. For example, here the input pin of component Factory represents that Factory can receive the message Order with two arguments: the product ID (?id: Product_id) and amount of product (?a: Amount).

On the other hand, the output pin of component depends on its position and label in the graph. If the output pin doesn't have the label but has a relevant input pin, it means the operation on the received message, such as the uppermost output pin of the component Factory. If the output pin has a label, it means the port for sending labeled messages which is either the result of some actions or to require some services of other components, such as bottom two output pins of the component Factory.

The external view of component only gives its interface for integration with other components. In order to correctly analyze the component behavior, the internal view need to be known also. In fact ATAG [7] use something close to UML state-diagram [6] to describe the activity inside component. The component Factory in Figure 35 could have the following graph of its concrete work:

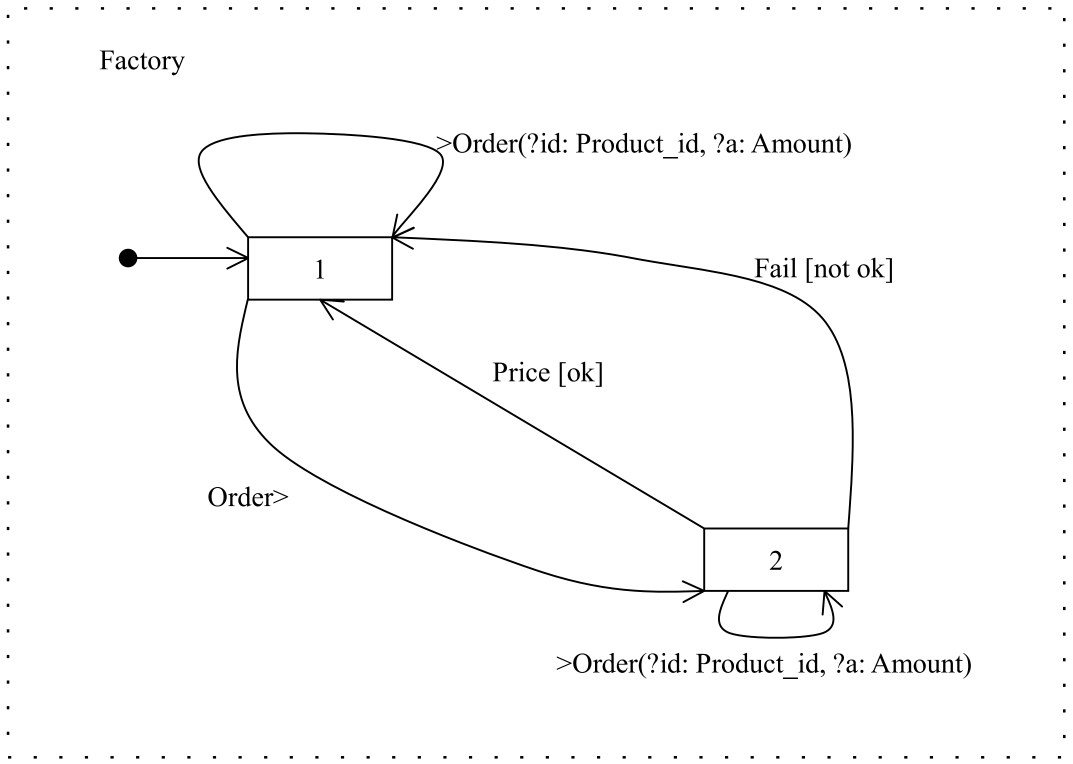

Factory
>Order(?id: Product_id, ?a: Amount)
1
Fail [not ok]
Price [ok]
Order>
2
>Order(?id: Product_id, ?a: Amount)

Figure 36: Dynamic Behavior of Factory

As seen in Figure 36, there are two states of component Factory. Each state is a box with the label of state number. The arrow with black solid round means that state 1 is the initial state of Factory. The arrow with label Order> indicates that the operation on message Order makes the component Factory change from state 1 to state 2. The two arrow links from state 2 to state 1 means that sending message Price or Fail by component Factory would also result in state change. The final interesting thing in Figure 36 is the arrow link from state 1 or 2 to itself, which explains that receiving message Order is nothing with the component behavior. Thus component Factory separates the message receipt >Order from its execution Order>.

Since the dynamic behavior and the interface of one component are just two aspects of a software unit, both could be represented together in a composition graph, shown like following:

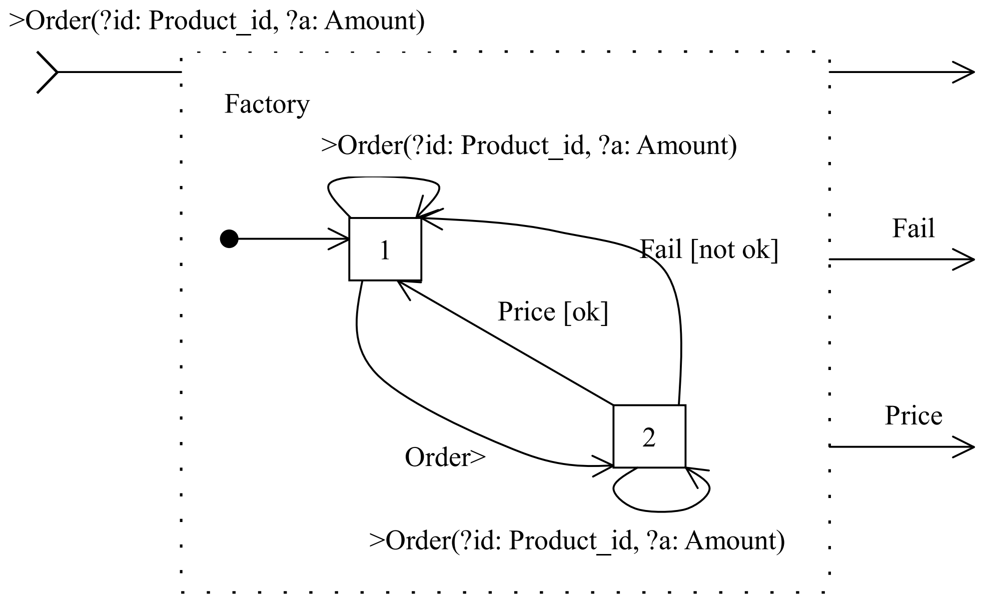

>Order(?id: Product_id, ?a: Amount)
Factory
>Order(?id: Product_id, ?a: Amount)
1
Fail [not ok]
Fail
Price [ok]
2
Price
Order>
>Order(?id: Product_id, ?a: Amount)

Figure 37: Factory Component in ATAG

According to Figure 35, Figure 36 and Figure 37, there are totally three kinds of link label in ATAG [7] graph to explain the arrow link.

1) **Autonomous operation**: this is generally written as op, which is the message sending automatically by the component. It is triggered by some methods inside the component to return the result or to ask for some additional service of other components in the system. The example is Fail and Price on the right of the component Factory.

2) **Message receipt**: this is generally written as >op, which shows the action of receiving the message by the component. The real operation may include putting the message in the buffer, as the solution of asynchronous communication. There could be some guard (condition checker, like Boolean variable or method at programming level) in this process. For example [not fullMailbox], tells whether the message buffer of component is full or not. The example is Order on the left

of the component Factory.

3) **Action**: this is generally written as op>, which simply means that the component executes the corresponding method or service for the received message. The action always changes the state of component in ATAG. Therefore, in contrast to synchronous communication, ATAG use >op and op> to denote the two independent steps in asynchronous communication. The example is Order> of the arrow link from state 1 to state 2 in component Factory.

The communication between the components is not only one-to-one, but also broadcasting, multiple inputs and conditional communication. ATAG [7] uses the link line from the output pin to input pin, to show the message path and participants in the all kinds of communication scheme. The graph example is like following:

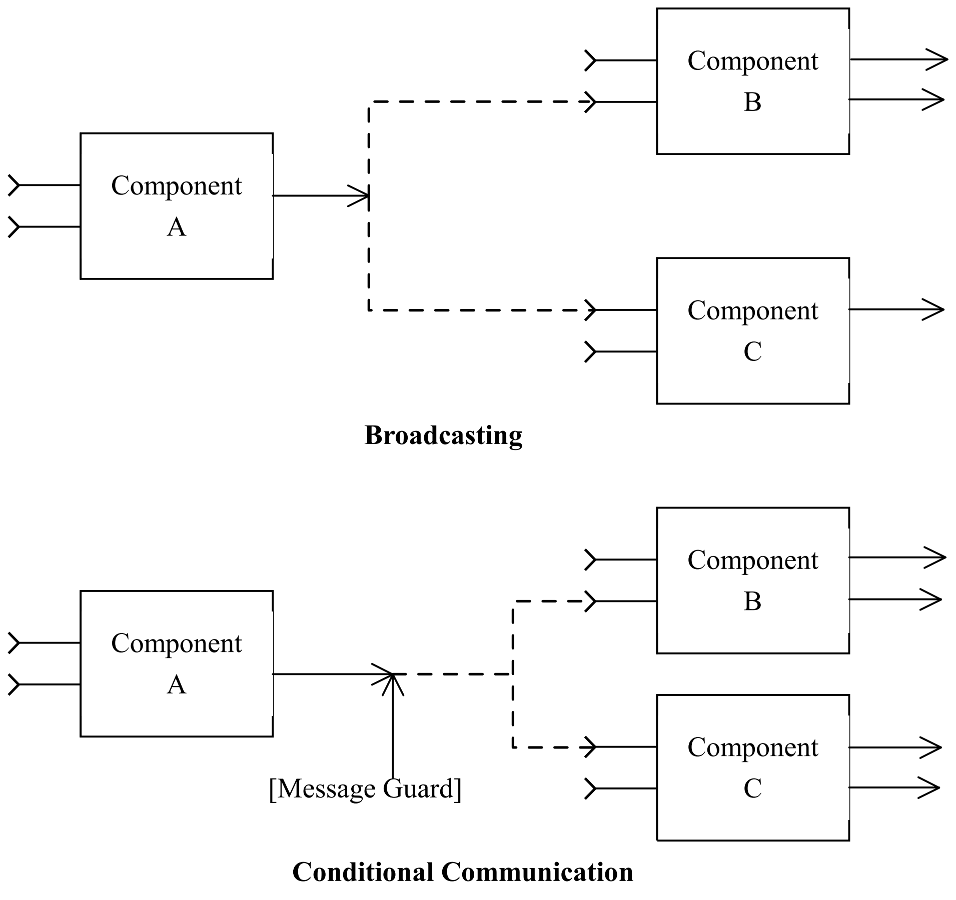


Figure 38: Communication Scheme in ATAG

The [Message Guard] in Figure 38 is to control the access of the component to the received message. It is just like a message filter with some fixed or dynamic service policy to plug on or embedded in the component.

With all above abstract graph, it is easy for ATAG [7] to build a component-based architecture in the process of system modeling. For example, there is a very simple system of ordering product, composed of two components. One is component Factory illustrated in Figure 37, and another is component Client. The Client could give the message Order to the Factory with the product ID and amount. The component Factory should decide if the product is available to the Order message, depending on its stock. If the message Order could be satisfied, Factory returns the message Price to Client for the Order, and if Factory can't provide as what message Order asks, it returns the message Fail to Client.

The Client component is represented in ATAG like following:

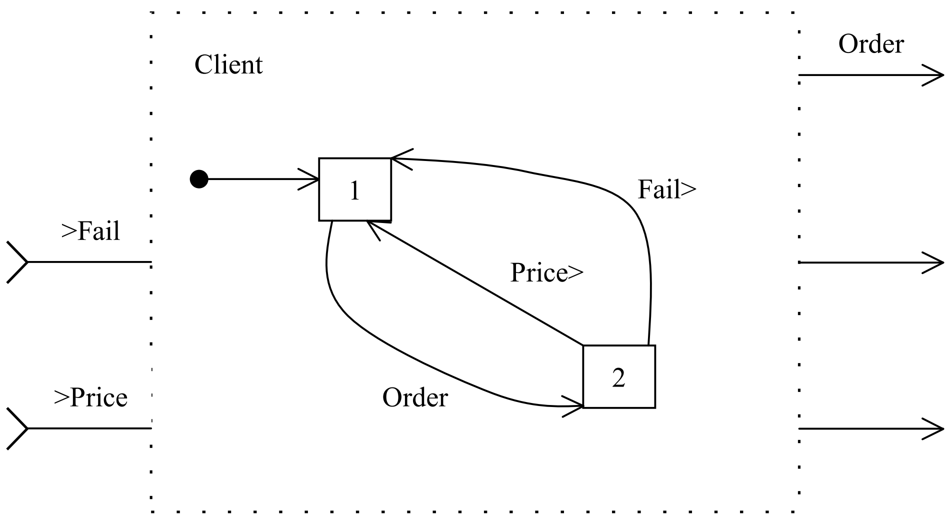


Figure 39: Dynamic Behavior and Interface of Client

Using the link line to connect the input pin and output pin of component Factory and Client, the system architecture looks as Figure 40.

The graph of system architecture gives the message-based communication following the interface of the components in the system, but it doesn't tell the relationship between the message flow and the system dynamic behavior. Especially it has not clear view on the synchronous or asynchronous style in the behavior description. Therefore, ATAG [7] uses some kind of STS to graphically express the message transferring and relative state change on multiple concurrent components.

As a matter of fact, ATAG [7] inherits the synchronization product of STS from Korrigan and GAT [2, 3, 5, 19, 22] to represent the global dynamic behavior with message transition between the components. Since in asynchronous component model, the message buffer is hidden in the component and buffer operation is also invisible from outside, it is feasible to use synchronous diagram to express the synchronous or asynchronous communication. ATAG adds an extra label '-' to show the special nil transition for the state of component; It means that nothing happens to the component. Generally, the concurrent communication by message between the components is represented as Figure 41.

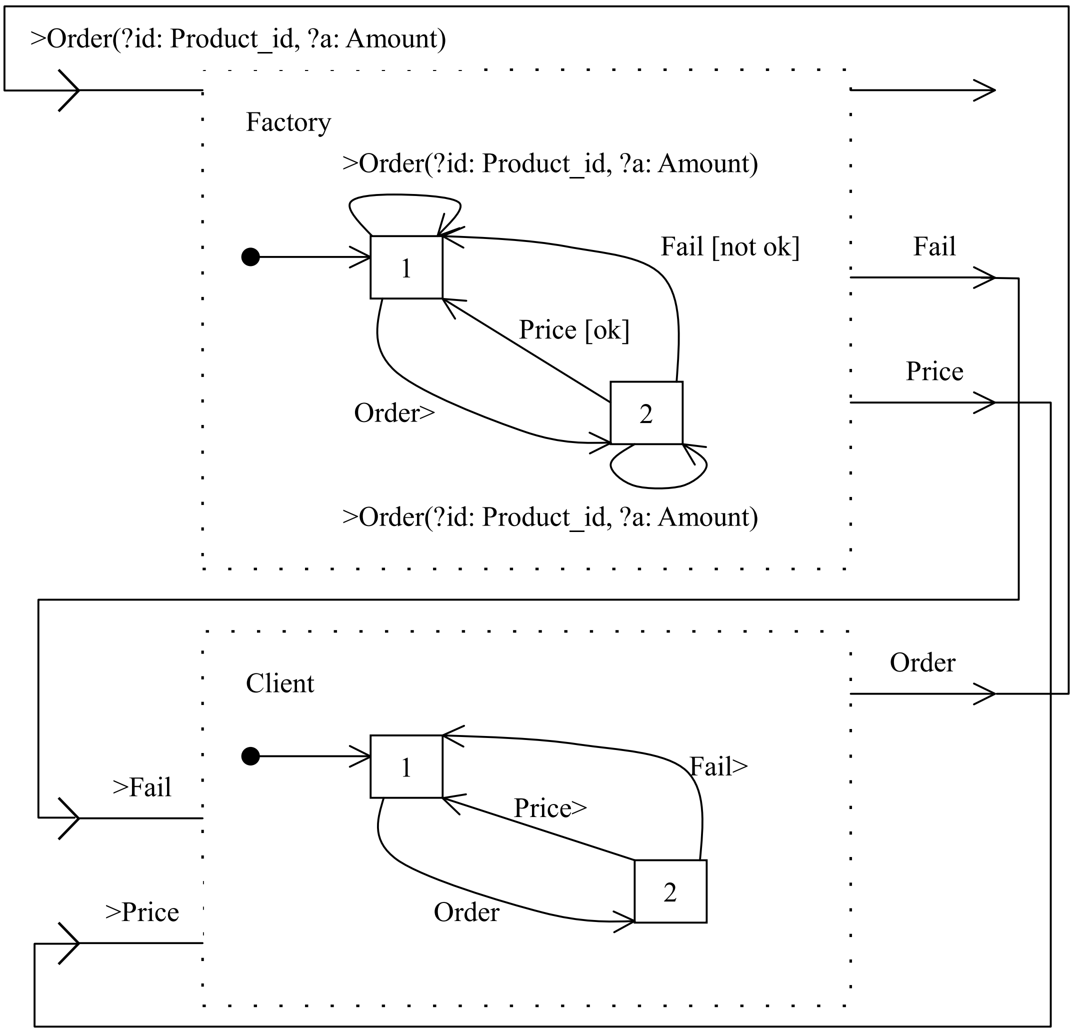

>Order(?id: Product_id, ?a: Amount)
Factory
>Order(?id: Product_id, ?a: Amount)
1
Fail [not ok]
Price [ok]
Order>
2
>Order(?id: Product_id, ?a: Amount)
Fail
Price
Client
Order
1
Fail>
Price>
>Fail
2
>Price
Order

Figure 40: Architecture of Product-Ordering System

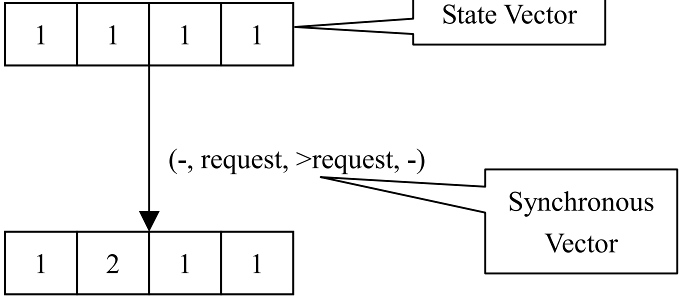


Figure 41: Dynamic Behavior with Communication in ATAG

Here synchronous vector is a tuple involving all the message-transmitting labels corresponding to the components in one-to-one way. Thus in Figure 41 there are four components in the system, and only the second component has the state change from state 1 to state 2 because it sends the message request as an autonomous operation. The sending and receiving of message request is treated in synchronous way, represented in synchronous vector shown in Figure 41. This means that there is a message link from the output pin of the second component to the input pin of the third component. This kind of abstract graph ignores the time factor of the message transferring between two components in the concrete implementation. Meanwhile, the order and size of synchronous vector and state vector is the same, so each component has its own corresponding item in the synchronous vector to express relative message transition. The state change of component is only triggered by autonomous operation and action for received message.

As for the previous ordering system composed of two components: Factory and Client, the global dynamic behavior with messaging could be represented as following:

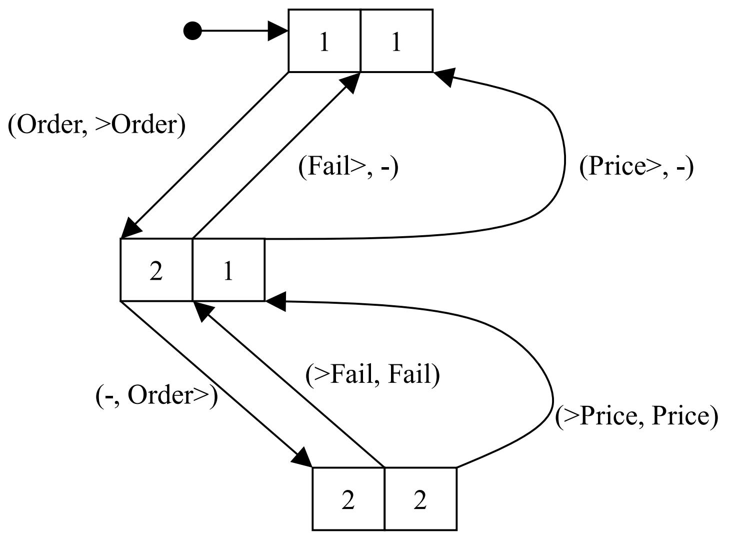


Figure 42: Dynamic Behavior and Messaging of Ordering System

4.2 Bound algorithm

It is clear that the asynchronous component model divides the method call into message sending by emitter and message processing by receiver in the different temporal point. As previously said, the classic way to simulate this asynchronous communication is to add a message buffer for a component, which stores all received messages to be operated by component in the future.

Thus this kind of solution obviously raises the question whether the buffer size is exactly enough to contain all possible messages given to the component in the asynchronous communication. If the buffer size were super than needed, this would cause idle resource and waste on maintenance. To the worse, if the buffer size were smaller than needed, it would lead to failure on message sending. In the latter case, the concrete implementation could be much expensive to handle the asynchronous method call to distinguish a slow response with a message losing, especially in complex environment. People need to obtain the specific and automatic tool to solve the question of buffer size. In addition, the tool should help the model designer to consider the potential danger of message buffer and adjust the whole component model with more security and reliability in mind.

Here I introduced my depth-first search algorithm based on ATAG [7] model to find out if each message buffer of its corresponding component in the asynchronous communication is bound or not. The algorithm also indicates the existing message cycle between the components. The message buffer is called mailbox in my algorithm, since mail-posting is a common asynchronous communication in the world.

In order to precisely describe the change of mailbox in the global dynamic behavior of asynchronous component model, the mailbox is assigned to each component state in a synchronous vector which is shown in Figure 41. Therefore, the message transition at each time would explain how the mailbox of every component is dynamically changed due to message receiving or operation.

In addition, the autonomous operation in ATAG [7] model is ignored in my algorithm, since it does nothing with message buffer. Moreover, it is supposed that the component sends the message immediately in the execution. My algorithm also assumes that the connection between any two components is always good enough to support messaging; that's to say, the algorithm only focuses on the problem of mailbox, no matter the other problems possibly affecting the communication.

Basically the algorithm is a kind of architecture behavior checker. The input data of my algorithm should be the ATAG without mailbox data, while the output data is the analysis result of component mailbox for each state in the synchronous vector. The output may include some message cycles. The UML data structure of my input is like Figure 43.

Though the data structure in Figure 43 seems considerably simple, my algorithm is general enough to deduce to more complex one to analyze all the mailboxes about the asynchronous component model. In addition, my algorithm follows that the two StructuredStates equal to each other, only if each ComponentState in the ordered aggregation of one StructuredState is the same as the corresponding one in another StructuredState. The number of component in the ATAG [7] model is fixed during the asynchronous communication.

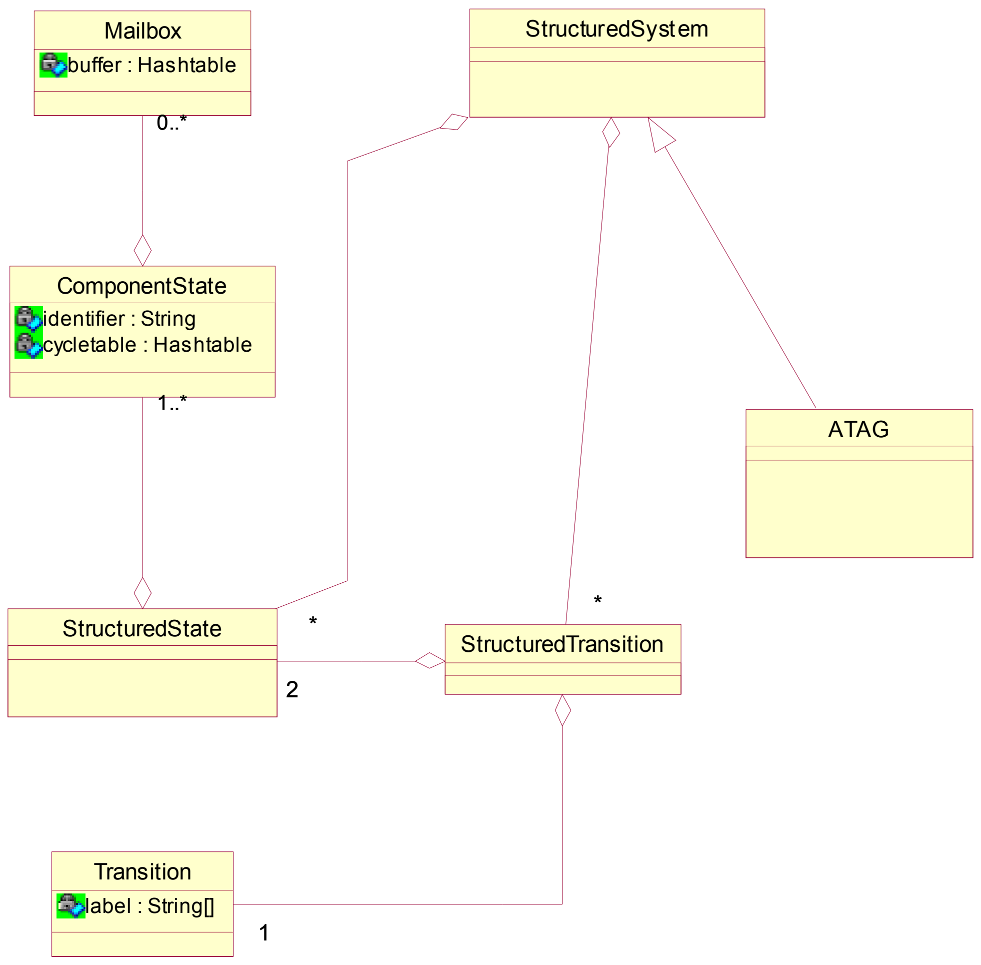


Figure 43: The UML Data Structure of Input

4.2.1 Statement of Bound Algorithm

My depth-first algorithm is some kind of graph traversal, and it puts W to the message number to indicate that the message is infinite to be received by the component. Infinite message brings to the unbound mailbox in fact.

In the ATAG [7] graph of dynamic behavior, there could be self-link to the state to make the component receive the message. Sometimes the number of a message in currently computed mailbox of component is bigger than the number of same message previously computed in the in the same mailbox. In both situations, W is set to the message. Graphically the condition to set W looks like Figure 44.

Meanwhile, whenever there is a message marked with W, this W should be propagated to other mailboxes for other states of the component in the ATAG [7] model. This is because each mailbox for one component is computed, based on the previous mailbox of that corresponding component as the beginner of the link of message transition.

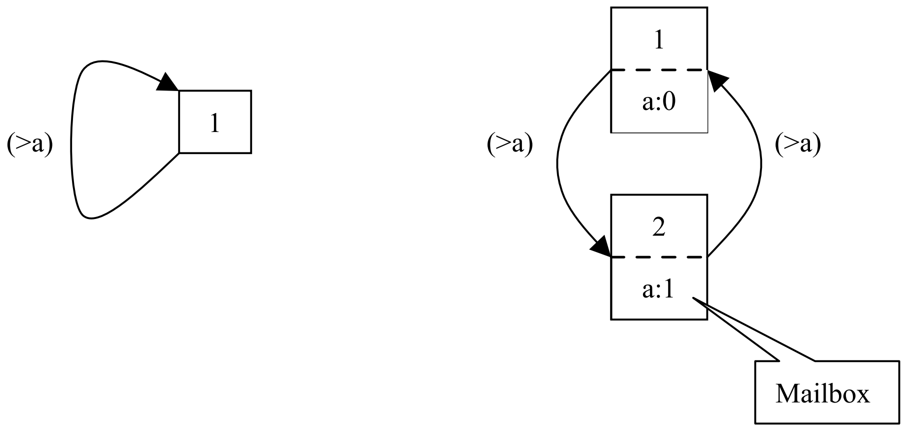


Figure 44: Condition of Setting W

The bound algorithm looks like following, and all the bold words means variables:

```
//** input ATAG data variable     **//
list_state_trans      //this list stores the structured transition of ATAG model
initialstate    //this structured state is the first state accessed from the start

//**    Temporary variable    **//
    list_state_search_path        //this list stores all the structured states on the current path of
                                  traversal, like a data stack
list_state_visited        //this list stores all the structured states visited in the traversal,
                          notes that here the structured state is with mailbox after applying
                          transition
hashtable_statetrans_neighbors   //this hashtable stores the list of structured transition
                                 of one structured state which is the source of
                                 transition

public check_ATAG_Mailbox()
{
  initialize list_state_search_path
  initialize list_state_visited
    initialize hashtable_statetrans_neighbors

    add initialstate to list_state_visited
    check_Structured_State_Mailbox( initialstate )
    clear list_state_search_path
    clear list_state_visited
    clear hashtable_statetrans_neighbors
}

private check_Structured_State_Mailbox( current_state )
{
  add current_state to list_state_search_path

  if (hashtable_statetrans_neighbors has the value for current_state ) then
          using current_state as key, copy the structured transition list in
          hashtable_statetrans_neighbors to list_neighbours
  else
  {
          in list_state_trans search all the structured transitions of current_state which
          is the source state of transition, and put them in list_neighbours

          store list_neighbours to hashtable_state_neighbors with current_state as
          key
  }

  while list_neighbors is not empty
  {
          put the first structured transition of list_neighbors to statetrans_neighbour,
          and remove that first one from the list_neighbors

    get dest_state from statetrans_neighbour
          search list_state_visited with dest_state, and put the search result in
          old_dest_state

    get src_state from statetrans_neighbour
          search list_state_visited with src_state, and put the search result in
          latest_src_state

    get transition from statetrans_neighbour
    apply transition to latest_src_state to get new_dest_state

    if( old_dest_state is null )
    {
      add new_dest_state to list_state_visited

      check_Structured_State_Mailbox( dest_state )
    }
```

```
        else
        {
                merge old_dest_state and new_dest_state by comparing their relative
                mailboxes to final_dest_state. In the process of comparing if one message
                has more in new_dest_state than in old_dest_state, then propagate that
                message with W sign if old_dest_state is in list_state_search_path, or
                propagate the number change if old_dest_state is not

          replace old_dest_state in list_state_visited with final_dest_state
        }
    }

    remove current_state from list_state_search_path
}
```

Table 2: Bound Algorithm

The relative propagation algorithm looks like below, and here –1 means W:

```
//** relative variable used in depth-first algorithm   **//
list_state_trans      //this list stores the structured transition of ATAG model
initialstate    //this Structured state is the first state accessed from the start
list_state_search_path    //this list stores all the structured states on the current path of traversal,
                          like a data stack
list_state_visited      //this list stores all the structured states visited in the traversal,
                        notes that here the structured state is with mailbox after applying
                        transition
hashtable_statetrans_neighbors   //this hashtable stores the list of structured transition
                                 of one structured state which is the source of
                                 transition

//**   Temporary variable   **//
list_state_propagated        //this list stores all the structured states propagated

//**   Parameters **//
//startstate is the structured state to start propagation of message
//componentindex is the index of component state in structured state
//message is the string of propagated message name
//number is the value to set or increase to message number, -1 means W

public propagate_Message( startstate, componentindex, message, number )
{
        initialize list_state_propagated
```

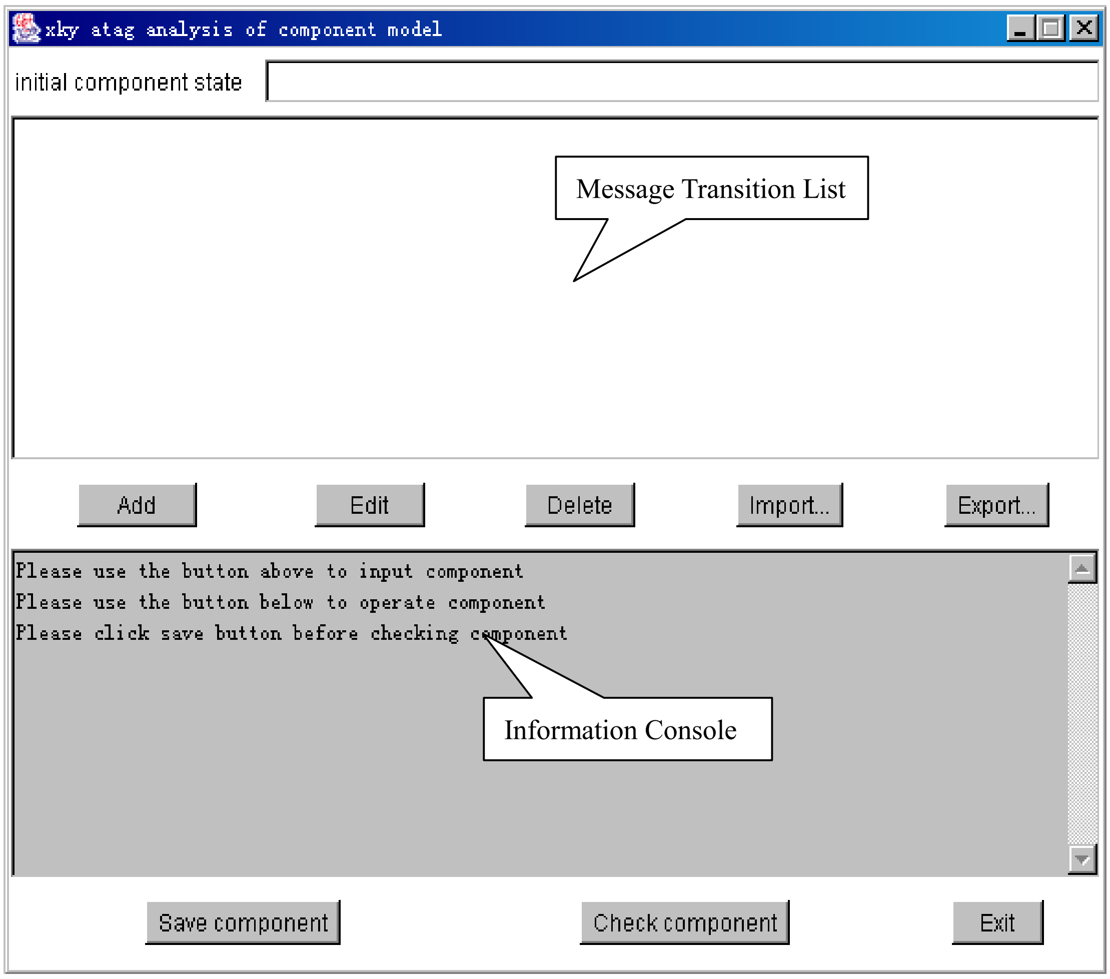
```
    propagate_State_Message( startstate, componentindex, message, number )

    clear list_state_propagated
}

private propagate_State_Message( current_state, componentindex, message,
number )
{
  if (hashtable_statetrans_neighbors has the value for current_state ) then
          copy the structured transition list in hashtable_statetrans_neighbors to
          list_neighbours
  else
    return

  while list_neighbors is not empty
  {
          put the first structured transition of list_neighbors to statetrans_neighbour,
          and remove that first one from the list_neighbors

    get dest_state from statetrans_neighbour
    if( dest_state has corresponding structured state in list_state_visited )
    {
      if( dest_state is not in list_state_propagated )
      {
                search list_state_visited with dest_state, and put the search result in
                latest_dest_state

                apply the change to latest_dest_state with componentindex,
                message, number, and put the result to final_dest_state

        replace latest_dest_state in list_state_visited with final_dest_state
        add dest_state to list_state_propagated
                propagate_State_Message( dest_state, componentindex, message,
                number )
      }
    }
  }
}
```

Table 3: Propagation Algorithm

The execution time and required source of my bound algorithm depends on the amount of message links and message cycles in the ATAG model. The message links decide the complexity of depth-first search in bound algorithm, and the message cycles claim the possible traversal in propagation algorithm.

4.2.2 Implementation of Bound Algorithm

The above algorithm has been implemented and tested in JDK 1.4.1 [29]. However, this algorithm is language independent. It is easy to be realized by a lot of popular programming languages to analyze the mailbox of asynchronous model. The main Java window of bound algorithm is shown in Figure 45.

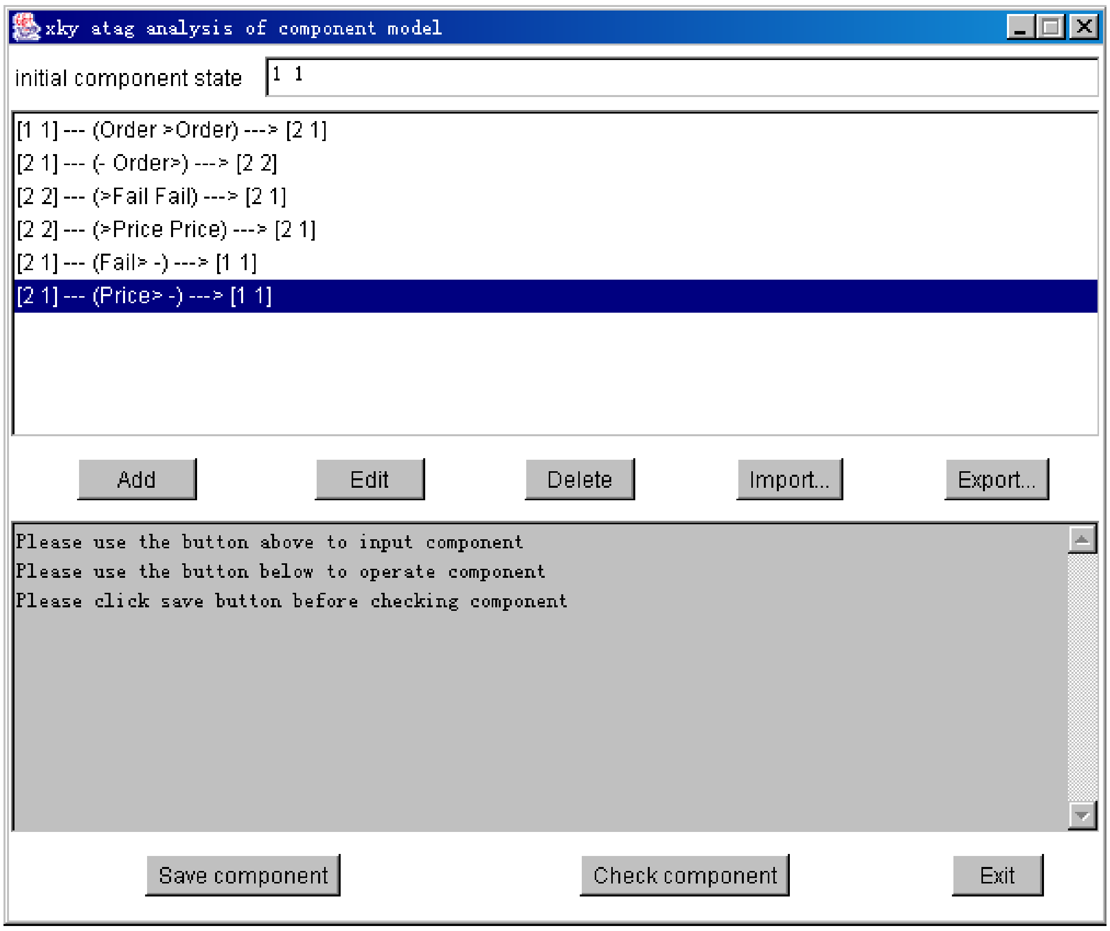


```
Please use the button above to input component
Please use the button below to operate component
Please click save button before checking component
```

Figure 45: Main Window of Bound Algorithm

If we take the ordering system introduce in section 4.2.1, the input data is based on that ATAG [7] model of its dynamic behavior and message transition shown in Figure 42. The actual description of that abstract graph should be transformed in a list where each item is a message transition with the synchronous vector as sender and receiver, with message label as well. It looks like in Figure 46.

As seen in Figure 46, the synchronous vector of all the states for the components is represented like [state1 state2 …]. The message vector of the concurrent messaging for all components is represented like (message1 message2 …). The --> in the line expresses the direction of state change during the message transition. Synchronous vector [1 1] explains the initial component states for the dynamic behavior in the ordering system.



```
1 1
```

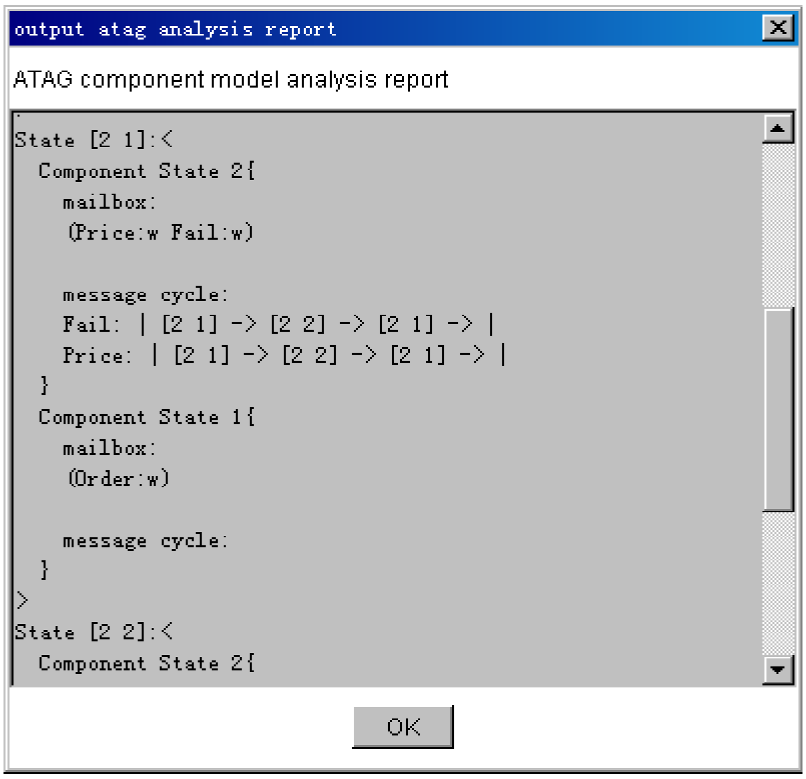
```
[1 1] --- (Order >Order) ---> [2 1]
[2 1] --- (- Order>) ---> [2 2]
[2 2] --- (>Fail Fail) ---> [2 1]
[2 2] --- (>Price Price) ---> [2 1]
[2 1] --- (Fail> -) ---> [1 1]
[2 1] --- (Price> -) ---> [1 1]
```

```
Please use the button above to input component
Please use the button below to operate component
Please click save button before checking component
```

Figure 46: Input of Ordering System

After inputting all the data of ordering system, the user should click the “save” button to construct the data structure for the bound analysis of that asynchronous component model. In addition, the user can use “import” and “export” button to save the input data to some text file, and the relative parser also was done in this java implementation to give the comfortable support. Using “analyze” button, the program gives the analysis result in Figure 47.

The bound algorithm could be applicable in more domain-specific development with component model. Supposing the classic operation of bank system, here we assume that there are two clients and one bank counter in the system. The client can deposit or withdraw the money from bank counter by the account number. Each time the bank counter can handle only one client; in other words, two clients can’t send the message to bank counter at the same time. This case would be extended to N clients with M bank counters, while the bound algorithm need no change in fact.


output atag analysis report
ATAG component model analysis report
OK

```
State [2 1]:<
  Component State 2{
    mailbox:
     (Price:w Fail:w)

    message cycle:
    Fail:  | [2 1] -> [2 2] -> [2 1] -> |
    Price:  | [2 1] -> [2 2] -> [2 1] -> |
  }
  Component State 1{
    mailbox:
     (Order:w)

    message cycle:
  }
>
State [2 2]:<
  Component State 2{
```

Figure 47: Analysis Window of Ordering System

The ATAG architecture of simple bank system is like following:

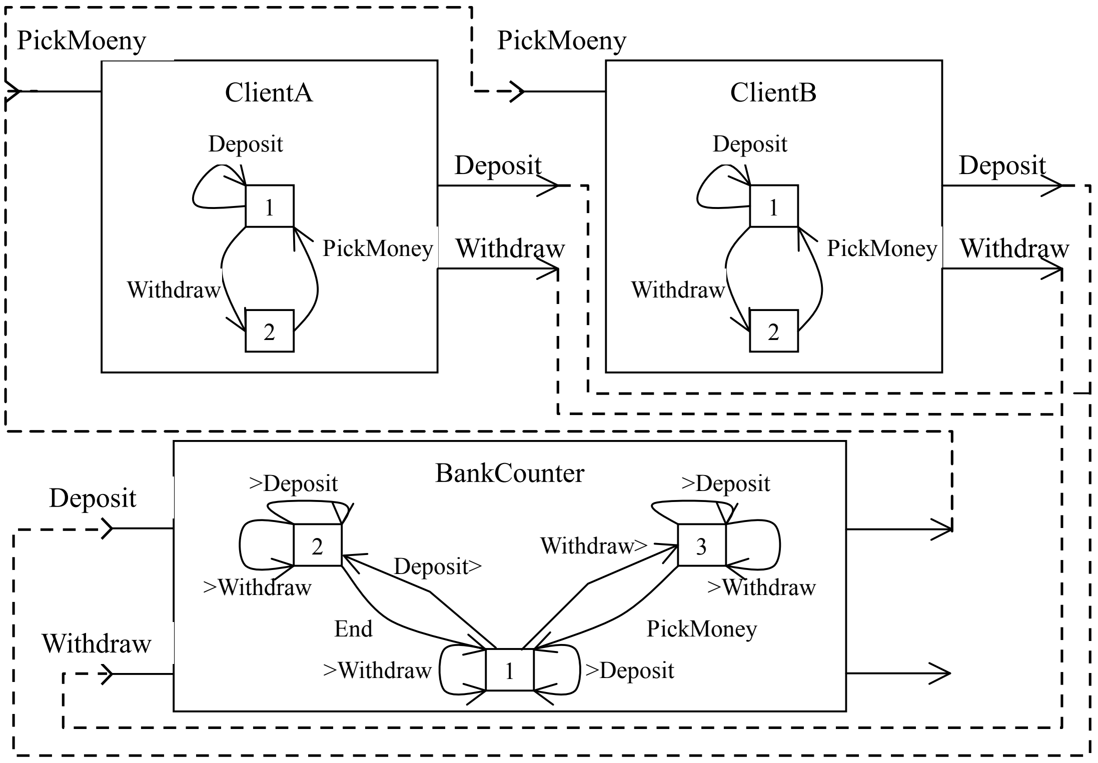


Figure 48: Architecture of Simple Bank System

According to the communication in Figure 47, the dynamic behavior of this simple bank system could be represented as below:

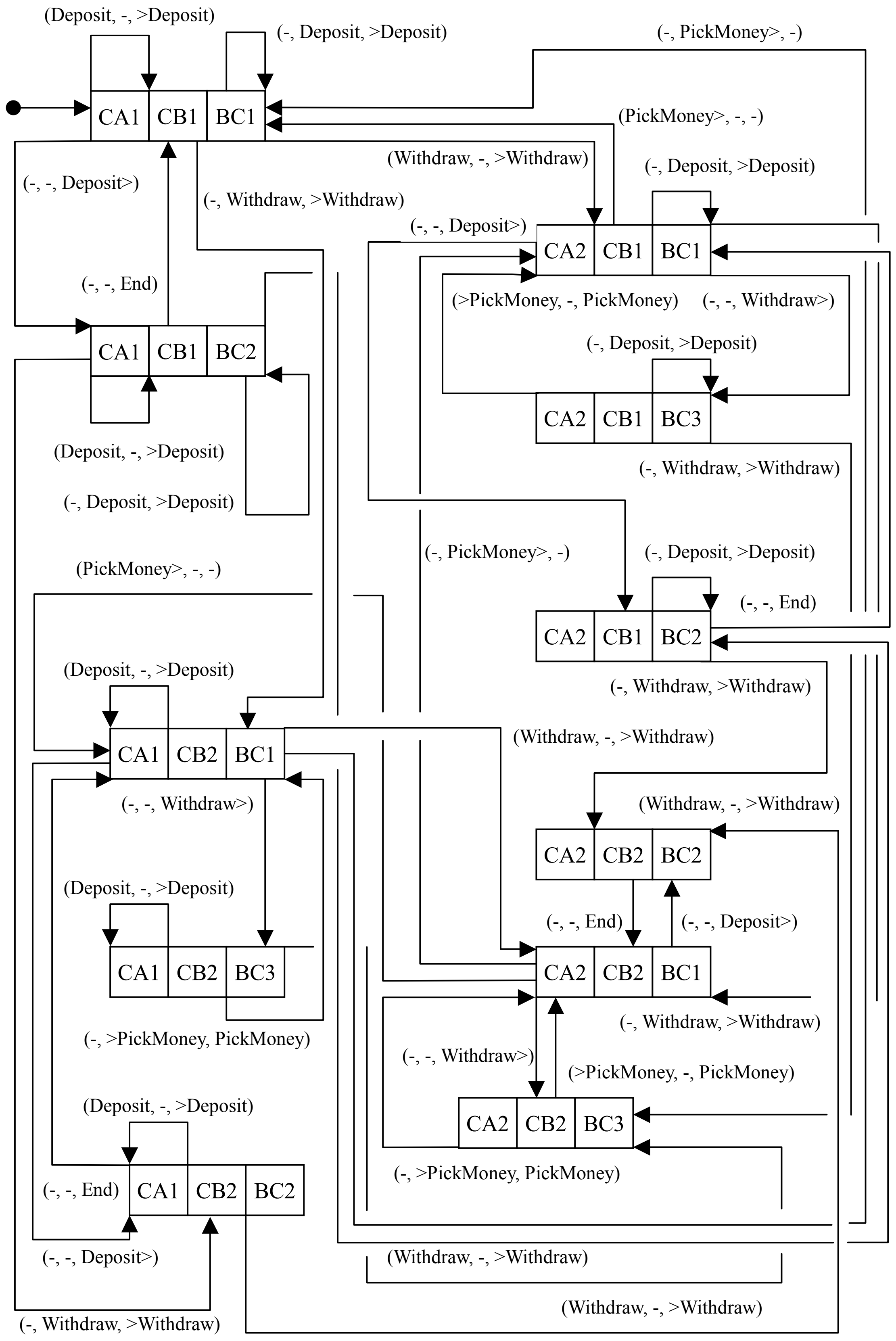
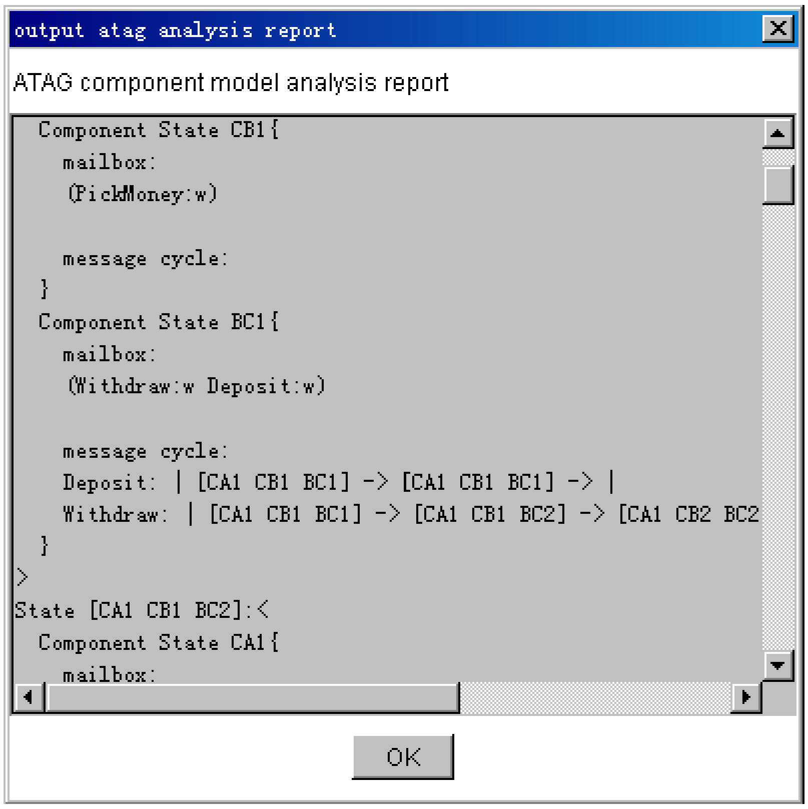


Figure 49: Dynamic Behavior and Messaging of Bank System

In Figure 49, CA means ClientA, and CA1 is the state name of ClientA. Thus component ClientA has two states shown in Figure 49. Likewise, CB means ClientB, and BC means BankCounter. The state name of ClientB and BankCounter follows the same rules as of ClientA. Due to the complex nature of asynchronous component model, the simple bank system has 11 state vectors with 39 transitions between the states, and 9 kinds of message vectors.

After inputting the data in ATAG model of simple bank system, we can get the analysis result of bound algorithm, shown in Figure 50.


output atag analysis report
ATAG component model analysis report
OK

```
  Component State CB1{
    mailbox:
     (PickMoney:w)

    message cycle:
  }
  Component State BC1{
    mailbox:
     (Withdraw:w Deposit:w)

    message cycle:
    Deposit: | [CA1 CB1 BC1] -> [CA1 CB1 BC1] -> |
    Withdraw: | [CA1 CB1 BC1] -> [CA1 CB1 BC2] -> [CA1 CB2 BC2
  }
>
State [CA1 CB1 BC2]:<
  Component State CA1{
    mailbox:
```

Figure 50: Analysis Window of Simple Bank System

Obviously, the mailbox in ATAG model of bank is unbound because the infinitive messages like Deposit or Withdraw are stored inside. The message cycle in the dynamic behavior of bank system is also clear, and always produces the increasing number of message in the context of asynchronous communication. Thus this bank system has to be added more restrictions to avoid the unbound mailbox and other potential danger for the work.

CHAPTER FIVE

CONCLUSION AND FUTURE WORK

Component-based development has been proved as an effective software technology to reduce the time and effort in software life cycle. The heart of component-based software engineering is the reusability of implementation even design. To properly reuse the existing component depends on the architecture design only which could decides what kind of component could be integrated into the system. Moreover, a good architecture could not only help the developer to easily find the off-the-shelf component in the market or legacy system, but also to develop some domain-specific or generic component to be reused in the future.

Generally, component is logically an independent function unit which is usually too simple to be solely used for even single application. On the other hand, nowadays due to the great advance in the network and telecommunication field, the communication in the software system is developed with all kinds of local-area network or wide-area network. Therefore, to design the correct communication in the component architecture becomes more important in architecture design phase. However, there is much less study on the asynchronous communication than on synchronous communication, though both of them is equally important to adopt in fact.

The work of this master thesis is to build a translation from general asynchronous component model to some concrete implementation, and help to get some sense different from implementation of synchronous communication. Furthermore, the work proposes some kind of representation to the asynchronous component model, and related algorithm to automatically analyze the characteristic of that model.

5.1 Conclusion

This paper describes some principles and rules in the component design and component-based architecture design of the CBSE. Some of the principles are not publicly accepted, but they are useful to indicate the important aspects and push the relative research forward.

ProActive, as a new java extension which default communication is asynchronous, is introduced to make a general implementation platform for asynchronous component model. The general PIM with asynchronous communication could be translated to

ProActive PSM with a set of simple rules in UML-like style. The corresponding programming in ProActive is also direct, and the synchronous and asynchronous communication is both easy to realize in ProActive.

The asynchronous component model is represented by ATAG in this paper. ATAG actually focuses on the messaging between components, as well as dynamic behavior of component. There is a bound algorithm presented in this paper, to estimate whether the mailbox (message buffer) is bound or not, according to the ATAG model. The implementation of bound algorithm is done in java, also with a user-interface to manipulate the ATAG data.

5.2 Future work

Component and CBSE is still being researched to make the advance to the next-generation methodology of software development. The principles about how to design and develop a satisfying component, and how to design a qualified architecture based on the component, are the centric topics to be discussed by the people with different background. For example, there is some question about whether customizable component is usually useful to the end-user who should change that software component by himself.

We should continue to collect all the stuffs of component study. Using the research result and experience on component-based system, we could get a more precise statement and understanding of the essence of component. The new rules about component and CBSE would be extracted from previous knowledge and current work not only in component field but in other IT domains; Simply saying, we need to evolve the principles in component-based software technology.

MDA are the new approach to produce the software system. Although this paper proposes some rules to translate the PIM to ProActive PSM, it is not enough and formal to cover all the aspects of PIM. Meanwhile, ProActive is just an experimental programming language based on java, so it still has a lot of things to realize for its goal. Thus, the translation rules have to be refined and complemented with the revising MDA standard by OMG, and with the new version of ProActive. The MDA-aware tools supporting the implementation from PIM to ProActive is also needed to ease the usage of asynchronous component model.

The ATAG model and bound algorithm presented in this paper assumes the asynchronous component model at a very high-abstract and high-reliable level. One feasible addition is to associate the maximum size to the mailbox in the component. In this case, the dynamic behavior would lead to more different state of mailbox, and corresponding component state as well. Some component would be blocked if the size of the mailbox of a component reaches its maximum, because at that time the mailbox refuses all other messages to the component. A straightforward solution to get analysis result of mailbox with maximum size is to split the work in two steps: the first is to reuse bound algorithm, and the second is to use maximum size to cut and revise the result of first step.

Since asynchronous component model is composed of dynamic behavior and static data type. We should use the experience of research on algebraic specification to static aspect of component, and then translate the static data part in asynchronous component model to ATAG model and even the implementation.

There are also some similar bound or analysis algorithms existing in other domain, such as rewriting logic for object model [51, 52], deadlock checking for components [53, 54], Petri Net with cover algorithm [12, 25]. Thus one possible way to analyze the asynchronous component model is to transform here ATAG model to other structural model, with a clear and exact mapping rules. The relative transformation interface and tool is also needed in fact.

Appendix A: Summary of ProActive

ProActive [39] is a kind of java extension library to ease the parallel, distributed, concurrent programming with good security, mobility and properties; It is the research project developed by *INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE*, as the part of work of ObjectWeb consortium [46] for OpenSource middleware.

The basic idea behind the ProActive [39] is the encapsulation of the object to generate the uniform remote object in the application. In other words, ProActive uses java RMI [42] mechanism to implicitly wrap all necessary objects to independently run in distributed environments (operation system, machine).

RMI [42] means Remote Method Invocation in Java technology. It is used for the system environment where there is two or more homogeneous Java Virtual Machine (JVM) [43] concurrently existing and communicating with each other. RMI exploits remote object to be invoked by the other objects in different JVM. In fact, the method of remote object able to be called from other JVM is separately defined in remote interface of remote object. Meanwhile, People have to implement the remote interface in some other class to finish the functions of remote object.

Moreover, the remote object should register itself to the RMI [43] Registry as a naming server, and every computer could have only one RMI Registry. The identifier of registered remote object is usually called RMI URL, like mail address. The general format of RMI URL is "rmi://hostname: portnumber/name". Here hostname means the machine address in the network, such as test.emn.fr. The portnumber is referred to the number of message port of RMI protocol, and the default is 1099. The name in RMI URL is decided by the programmer while he registers the remote object to RMI Registry. Using RMI URL, any object could locate the relative remote object and then ask the service of it.

However, RMI [43] operation and the manipulation on remote object is not that easy as doing with common Java object. Thus ProActive [39] provides some way to update the remote objects. The new entities (active object) in ProActive could communicate like remote object but they are more like local and standard java object from the point of ProActive programming. ProActive is trying to shorten the gap between remote call and local call as in Java programming.

In addition, ProActive [39] focuses on the asynchronous call, Automatic future-based synchronizations, migrating and mobile agents, remote creation of remote objects, group communications with dynamic group management, sophisticated synchronizations and collaborative applications.

ProActive [39] doesn't make any change to standard java library, java compiler or JVM. Thus ProActive keeps 100% compliance to Sun. The following are some interesting but important additions to make ProActive support its own characteristics.

1. Active Object

Active object is the basic concept in the ProActive [39] programming, and it resulted from the application of a model introduced by the concurrent Eiffel (OO language) [50]. The model requires that:

a. The application is structured in subsystems. There is one active object (and therefore one Java thread) for each subsystem and one subsystem for each active object (or say thread). Each subsystem is thus composed of one active object and any number of passive objects (possibly zero). The thread of one subsystem only executes methods in the objects of this subsystem.

b. There are no shared passive objects between subsystems; in other words, the passive objects are only visible and usable to the other objects within the same subsystem.

This model implies that one subsystem shows its service through its single active object, and only that active object is known to all the objects outside the subsystem. Therefore if an object o1 has a reference onto a passive object o2, then o1 and o2 belongs to the same subsystem. The general object model in ProActive is shown in Figure 15.

Meanwhile, when passive object is passed as parameters of calls to active object, it is always passed by deep-copy. On the other hand, active object is always passed by reference.

ProActive [39] follows active object pattern to create active object, and active object pattern is the uniform way to encapsulate:

1. A remotely accessible object, as the servant that provides the real object's implementation of function.
2. A thread as an asynchronous activity, which dequeues the method request and send it to servant from the pending request queue according to the service policy.
3. An actor with its own script to decide the corresponding behavior for the received message, like a stub.
4. A server of incoming requests, which enqueues the method request to the pending request queue.

5. A mobile and potentially secure entity, which is responsible for the migration and security.

Therefore as a result, when you create an active object A of some class, from the view of RPC you will have 4 parts in fact, shown in Figure 17:
1. stub_A (client side, including the actor)
2. bodyproxy_A (client side, request sender)
3. body_A (server side, the request-receiving server including the thread)
4. instance_A (server side, the real object implementation)

Here the body_A of the active object is the only object able to access directly the instance_A. This means that only body_A could directly call the method of instance_A. Additionally, body_A is a non-changeable object that delegates most of its tasks to helper objects called MetaObjects, which could be customized by implementing the MetaObjectFactory interface.

The parts of client side: stub_A and bodyproxy_A should exist in the environment of client object which sends the message to active object A. On the other hand, the parts of server side: body_A and instance_A run in the place possibly different from that of stub_A and bodyproxy_A

To create an active object, you should invoke one of the static methods newActive or turnActive of the ProActive class. ProActive.newActive creates an active object based on the instantiation of a new object; ProActive.turnActive creates an active object based on an existing Java object. When using ProActive.newActive you must make sure that the arguments of the constructor are Serializable. If ProActive.turnActive method is done on a remote node, the class used to create the active object in this way has to be Serializable. The simple example code is as following:

```
//---- in newActive case
A activeA = (A) ProActive.newActive("A");
Or
//---- in turnActive case
  A a = new A();
A activeA = (A) ProActive.turnActive(a);
```

Table 4: Creation of Active Object in ProActive

Table 4 shows how to create the non-argument active object activeA by the newActive or turnActive method. In fact activeA is a direct reference onto an instance of the generated ProActive stub_A for the class A because stub_A is a subclass of class A. Therefore ProActive allowes instances of class stub_A to be assigned to variables of type A, as the basic polymorphic rule in OO paradigm. In addition, activeA is associated with a default node created by ProActive. The node is a kind of remote object with entry in RMI registry, and user can also create node and then explicitly link the active object to the node.

There are some restrictions to create active object:
1) Final classes cannot give birth to active object
2) Same thing for non-public classes
3) Classes without a no-argument constructor cannot be reified.
4) Final methods cannot be used at all. Calling a final method on an active object leads to inconsistent behavior.
5) A direct call to a method of the originating object without using its active object will break the model.

The 3) limit is from the reason that when ProActive [39] constructs the stub of active object, it will call the non-argument constructor of standard java class as its parent constructor. If the class only has the constructor with argument, the argument would be used on stub not on the instance of that class, so this leads to some unexpected behavior at last.

Since active object not only owns the method implementation from the true standard java class, but has also its own activity to manage the method request from the call of other object. One can completely specify what will be this activity through three ProActive interfaces: InitActive, RunActive, EndActive. The default activity is to serve all incoming requests one by one in a FIFO order. There are two ways to define the activity of your active object:
1) Implementing one or more of the three interfaces of active object directly in the class used to create the active object, like following code:

```
import org.objectweb.proactive.*;

public class A implements InitActive, RunActive, EndActive {
   private String a;

   public A() {

   }

   // -- implements InitActive interface
   public void initActivity(Body body) {
      ……………
   }
```

```
    // -- implements RunActive interface
    public void runActivity(Body body) {
       …………….
    }

    // -- implements RunActive interface
    public void runActivity(Body body) {
       …………….
     }
  }
```

Table 5: Implement Activity Interface in ProActive

2) Passing an existing object implementing one or more of three ProActive interfaces to the method newActive or turnActive. Using the active object in Table 5, the code is like in Table 6.

   In this case, there is one restriction that you cannot access the internal state of the reified object.

```
B activeB = (B) ProActive.newActive("B", null, null, new A(), null);
```

Table 6: Pass Old Active Object to Create New One

2. Asynchronous Call

Since one of the essential nature of network application is asynchrony, ProActive [39] try to solve this difficult problem on the language level. When a method is called on an active object in ProActive, it returns immediately (as the thread cannot execute methods in the other subsystem) a future object, which is like a placeholder for the result of the methods invocation. The future object would be populated with the true result later by the method of active object.

The future object is the type of a refiable object which can not throw checked exception, and it includes three parts: stub_future, proxy_future, object_future. The stub_future is to receive the message for future object, and it reifies the message and sends it to proxy_ future. proxy_future calles the object_future for the proper operation. Here the object-future is not evaluated until the true result is returned by called active object. It is graphically similar to active object like below, and the arrow link means message sending between the parts of future object.

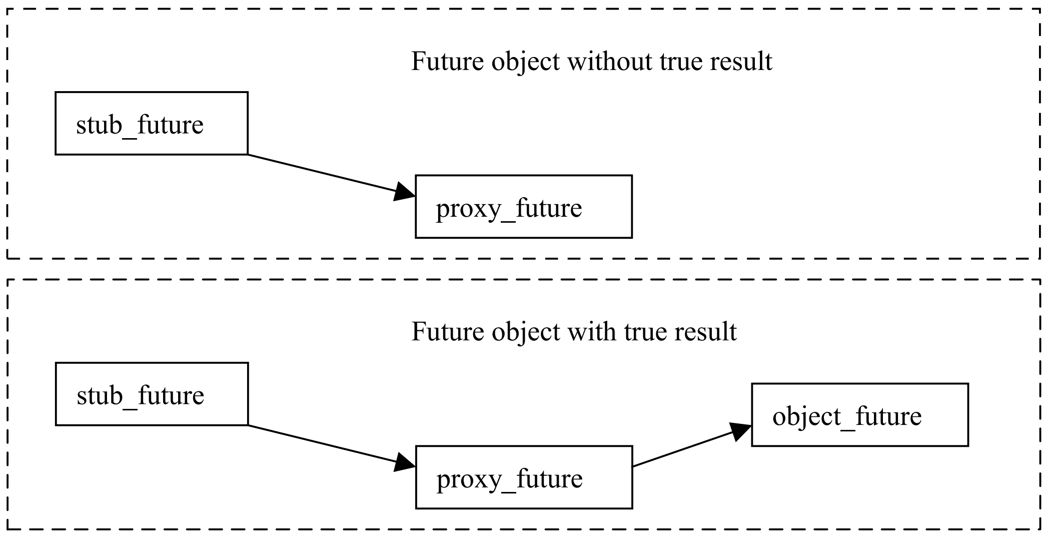

Future object without true result
stub_future
proxy_future
Future object with true result
stub_future
object_future
proxy_future

Figure 51: Future Object Composition

All the future objects created by the active object are kept in the futurepool of active object. However, two basic Java methods don't follow this asynchronous call scheme: equals and hashCode. When the other method calls equals and hashCode on future object, the function of both methods is not correct as usual. Sometimes both methods easily lead to deadlock when the future object is not available yet. Likewise, toString() method of future object should also be taken care of since it possibly blocks the program to deadlock.

The procedure to implement asynchronous communication by ProActive [39] is based on the interaction among caller, active object and future object. The stub of active object wraps the received message to methodcall object, and sends it to bodyproxy of active object. The bodyproxy creates the future object and send methodcall object to the body of active object. The body returns the stub_future to the caller, and calls the function of instance of active object at some time. The instance provides the true result as object_future, and the body of active object tells the reference of object_future to proxy_future of future object. Figure 19 illustrates the general method call with asynchronous communication in ProActive.

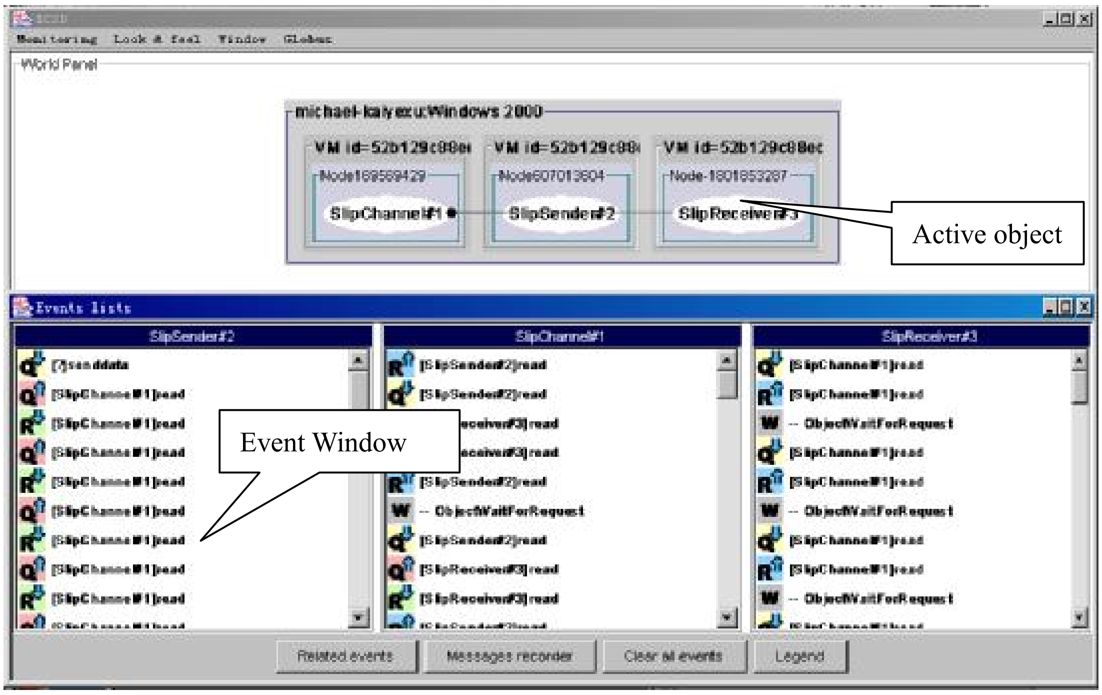
During asynchronous communication in ProActive [39], the caller would proceed to execute until it need to use the result (future object) of method call. Then the caller actually sends the message to the stub_future of future object instead of true result (object_future). The stub_future wraps the received message and gives to proxy_future of future object. According to the reference of object_future of future object, proxy_future calls the true function in object_future. If object_future were not available, it would wait until the future object is given the result by the active object. This solution of asynchronous communication is called wait-by-necessity.

3. IC2D

ProActive [39] gives a graphical user-interface to monitor and control the status of various active objects in the heterogeneous environments, and its name is IC2D. For example, user can drag and drop one active object from one machine in USA to the machine in France, and observe the message interaction between all kinds of active object.

The Figure 49 shows the window of IC2D. It is monitoring a system based on Slip Protocol. Because the system is running on a machine with Windows 2000, IC2D represents this with a big box and some label of the computer information. There are three active objects running in the system, so IC2D shows three active objects by three small boxes involved in machine box plus relative class name and number. Corresponding to the active objects, IC2D presents three event-monitoring windows for the message executed in the related active objects. IC2D uses the different icon to explain the type of message. In fact, IC2D has many options and menu items to adjust its function and graphical interface.

IC2D is being developed to interface with Jini and Globus, both of which supports global-scope distributed application, metacomputing, computing, and at the higher level than RMI.



```
[?]senddata                 [SlipSender#2]read          [SlipChannel#1]read
[SlipChannel#1]read         [SlipSender#2]read          [SlipChannel#1]read
[SlipChannel#1]read         [SlipReceiver#3]read        -- ObjectWaitForRequest
[SlipChannel#1]read         [SlipReceiver#3]read        [SlipChannel#1]read
[SlipChannel#1]read         [SlipSender#2]read          [SlipChannel#1]read
[SlipChannel#1]read         -- ObjectWaitForRequest     -- ObjectWaitForRequest
[SlipChannel#1]read         [SlipSender#2]read          [SlipChannel#1]read
[SlipChannel#1]read         [SlipReceiver#3]read        [SlipChannel#1]read
[SlipChannel#1]read         [SlipReceiver#3]read        -- ObjectWaitForRequest
```

Figure 52: ProActive IC2D Window

4. Migration

ProActive [39] provides two sets of static methods on the migration operation

1. The first set is aimed at the migration triggered from the active object that wants to migrate.
2. The second set is aimed at the migration triggered from another agent than the target active object.

Since we rely on the serialization to send the object on the network, the active object that wants to migrate must implements the Java serializable interface. The migratable active object must have a method which contains a call to the migration primitive, and this call statement must be the last one in the method.

In case of non-serializable type, in front of the declaration of that kind of variable, user should put the keyword transient. This indicates that the value of this variable should not be serialized. Therefore after the first migration, the value will be set to null since it has not been saved in the migration. Then we have to rebuild that variable with its old value by overriding the standard method in Active object: readObject(ObjectInputStream).

5. Deployment Descriptor

Parameters about the deployment of an application should be totally described in a XML deployment descriptor saved in an XML file. ProActive [39] nodes can be obtained from VirtualNodes (VN) declared and defined in a ProActiveDescriptor, being used to create active object. After activation of program, VirtualNode is mapped to Node or a set of Nodes, and the Node name is VirtualNode_Name + RandomNumber. VirtualNode must be associated with the certain JVM [43] to create the useful node for active object. The mapping between the Virtual Node and JVM could be: 1 to 1, 1 to n, n to 1, and you can use RMI [42] or Jini [47] protocol to explain that mapping.

6. Group Operation

ProActive [39] gives the group operation on a typed group of active objects which are located on a set of nodes. Programmer can use type of class or group interface to manipulate the group creation and operation. The result of method invocation on a group is also a group, and each element of the result could be a future object to do asynchronous calls. ProActive allows user to scatter the parameter of method call to the members of one group.

7. MetaObject Protocol

ProActive [39] is built on top of a metaobject protocol (MOP) that permits reification of method invocation and constructor call, so it need java reflection API which is in JDK1.1 or higher.

If the programmer wants to implement a new metabehavior using ProActive metaobject protocol (basically RMI [42] protocol), he or she has to write both a concrete (as opposed to abstract) class and an interface. The concrete class provides an implementation for the metabehavior he or she wants to achieve, while the interface contains its declarative part.

There are two methods to create the instance based on this meta-object protocol:
1) MOP.newInstance according to the class of the interface
2) MOP.newWrapper according to the existing object of the class of the interface

Through what I explained above, we can see that ProActive [39] is a new approach to realize the high efficient development on the network-specific application, thanks to the previous RPC study and internet language Java. If SUN company could deal with ProActive to make it as perfect plugin for the standard java, it must be much better with less restriction but more freedom to use ProActive.

Appendix B: Model-Driven Architecture

Nowadays human life is full of “living” software, and thousands of software companies are developing the new system to make world run in a faster and more autonomous way for all kinds of dimensions. According to the investigation of the cost during the life cycle of software development, the maintenance phase usually occupy a big part in total and it could be very difficult. People is never able to exactly estimate the possible change in a long time; That’s to say, the developers or teams must spend a lot of time and effort to update the original version of software for various emerging requirements.

The software must be designed and implemented in the binary code by the support of concrete software technology, which is usually the choice decided in the early stages of software life cycle. In this case, the problem to be solved by updating the software, in terms of technology, mainly exists in the connection (integration) with other (legacy) system and the adjustment to the new software techniques or platform.

The increasing amount of usable development technology makes software updating more common, as each technology has its own advantages and disadvantages to meet the specific requirement from the certain application. Meanwhile not all the technology has the downward compatibility in its evolution. For example, a bank system developed by Visual Basic 5, has to be modified for the condition whether it need to interact with a client-management system developed by Visual FoxPro, or it should be installed in a computer with Windows XP.

Obviously the more complex and bigger the system is, the more expensive the maintenance is. In fact, because the software development technology changes too fast, many big software systems have to be re-developed almost as a new one compared to its last version. To the worst, the extreme expenditure of maintenance would force the company to bankruptcy. Thus people need to find some solution to keep the soul of software more self-adaptive.

One new way to address above questions is suggested by OMG (Object Management Group), called Model-Driven Architecture (MDA) [44]. This approach is not a brand new idea but a formalized solution to the suffering update of software. The key of MDA is the separation between business model and technology support, which means to minimize the implementation-based effect to the system model or architecture. Therefore, MDA is model-centric software development. MDA is not referred only to the architecture design process but to the whole life cycle of implementation, integration, deployment and so on. MDA allows the developer to build the software system in his favorite way, but makes that system able to communicate or adapt to other software or platform. Moreover, MDA supports the evolving standards in the software development. The system following MDA looks like below [44]:

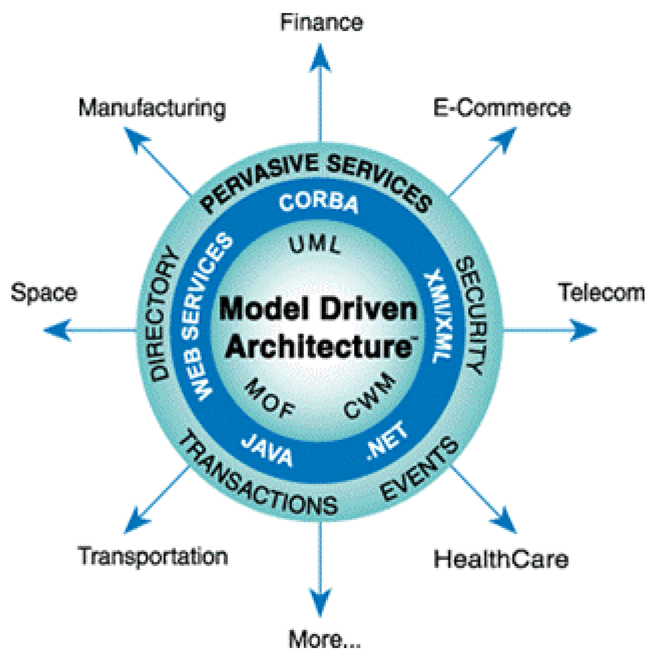


Figure 53: MDA-Based Application

In Figure 50, it shows that the system could be reused for the diverse domains like space or e-commerce, as long as the core architecture is applicable for those domains. The core modeling in MDA depends on OMG's three open standards for model design: Unified Modeling Language (UML) [34]; Meta-Object Facility (MOF) [45]; Common Warehouse Meta-model (CWM) [45].

1. UML

UML is a graphical language to describe the model or architecture of software system. UML is used mainly for the object model or component model. By writing the abstract functions into concrete and visual graph, it is more convenient for people to understand the design decision or control flow. UML also help developer to follow the model in the implementation and validation process of development.

Since now UML language becomes the industry-standard for the software modeling, there have already been some tools to translate the UML to other formal system specification. The specification could be technology-independent like XML, or technology-dependent such as EJB [8].

2. MOF

MOF [45] is a standardized framework with well-defined abstract language to express the metamodel composed of a set of metadata. MOF includes a set of necessary syntax, type, rules of metamodel used to build the object or component model for the software system. Moreover, MOF defines a hierarchy of reflective interfaces to make the metamodel interoperatable with other MOF-compliant metamodel which has different domain-specific or generic interfaces.

MOF also contains the common service for the metadata repository which is made to support the construction, traversal and update of a model as the instance of a particular metamodel [8]. Therefore, people can use MOF to visually create, publish and integrate multiple metamodels from the metadata repository according to some semantics and environments. In addition, the developer can use MOF-compliant tools to get the default implementation of instance of metamodel, and then fulfill it with other essential codes or data.

3. CWM

CWM [45] gives another metamodel, which is usually beyond MOF, to represent both business and technical metadata interchange among the data warehousing, business or knowledge process domain [8]. CWM could be expressed by UML notation, but extends some of core metamodel of UML with its unique features [8]. As a matter of fact, CWM is composed of a hierarchy of metamodels for the different purpose. For example, data resource metamodels are used to deal with the various types of data resource like SQL Server, while foundation metamodels are used to describe the general data or service for data resource metamodel or other high-level metamodel [8].

CWM-aware system could exchange the metadata of component for data warehousing or business behavior as long as the metadata follows the data specification defined in CWM. There are some tools that could directly generate the domain-specific instance of model in CWM, and then the developer should revise the instance to meet system requirement.

Although CWM provides highly generic and interoperatable metadata to build the MDA application, it is still possible for CWM-compliant system to handle with other different metadata from some system [8]. In this case, the developer has to make the extension to CWM through CWM service or metamodel, or use other relative metadata-supporting tools [8].

After using tools of UML, MOF or CWM for the MDA system modeling, people can get triple-layer model structure according to the external view of architecture design. In terms of order of simple-to-complex, the top layer is Computation-independent business model, and the middle floor is Platform-Independent Model (PIM), and the bottom is Platform-Specific Model (PSM).

Computation-independent business model is the most abstract representation of software system which only gives the brief solution model for the domain-specific problem with no concrete or clear computation or function. Thus the second layer of PIM is designed to fill up its super model with more detail of function or behavior assigned to object or component in the software system. However, PIM ignores the detail related to the implementation technology or platform, which is complemented in the PSM, such as communication constraint in EJB or .NET. In addition, the relationship between two neighboring layer of model is not unidirectional but both-way to refine and optimize the model and relative architecture control.

The MDA model structure is graphically shown like this [8, 9, 10, 11]:

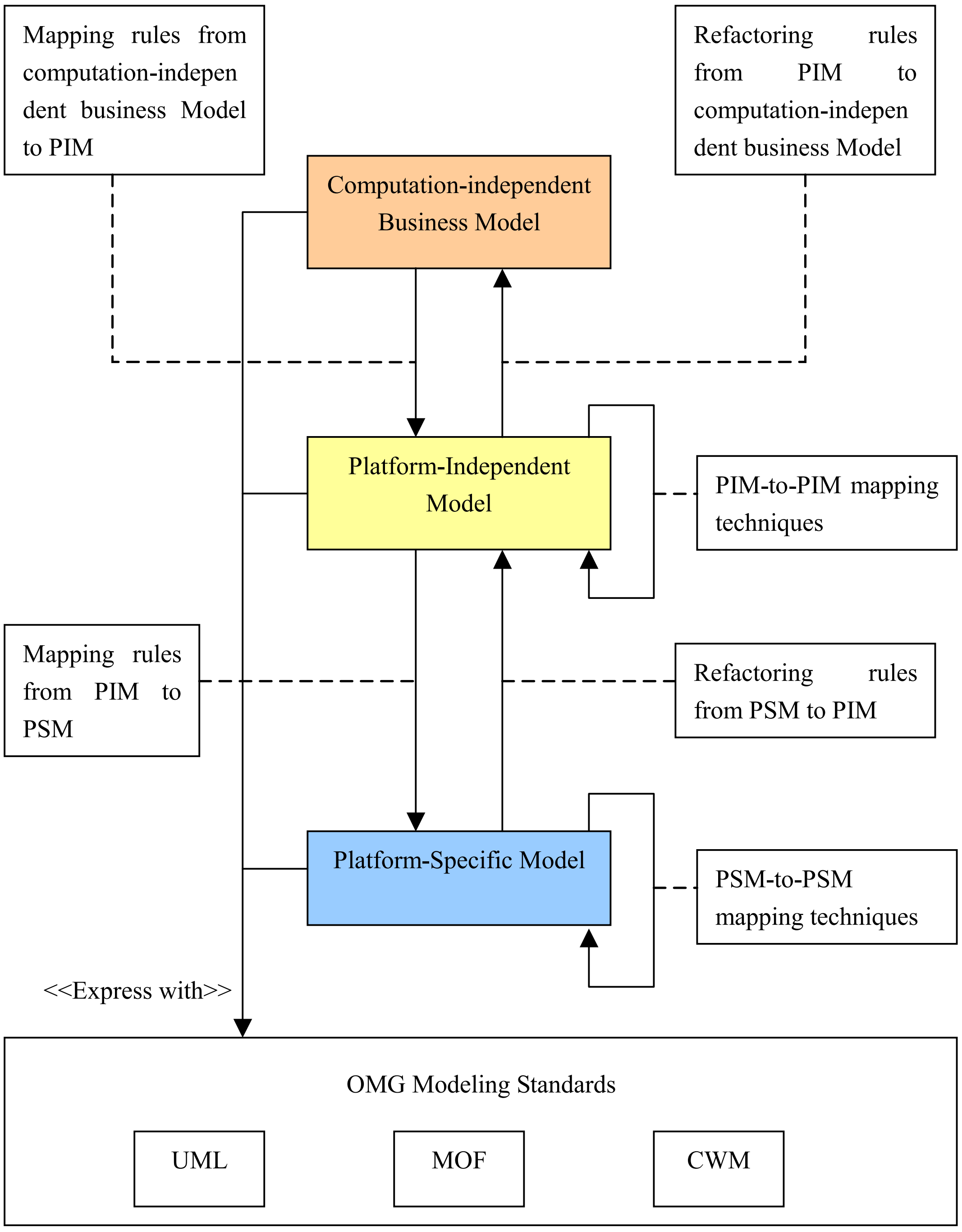

Mapping rules from computation-independent business Model to PIM
Refactoring rules from PIM to computation-independent business Model
Computation-independent Business Model
Platform-Independent Model
PIM-to-PIM mapping techniques
Mapping rules from PIM to PSM
Refactoring rules from PSM to PIM
Platform-Specific Model
PSM-to-PSM mapping techniques
<<Express with>>
OMG Modeling Standards
UML
MOF
CWM

Figure 54: MDA Model Hierarchy

In MDA [44] design, the important model is PIM which could be mapped to many PSMs if there exists the mapping rules to support this kind of transformation. For example, the PIM model for course schedule may be mapped to multiple PSMs like CORBA model, Java model, and so on.

There have already been some software tools to make the mapping from PIM to a certain PSM automatic like a translation machine. The tools even support the automatic code generation such as UML-to-C++ [9]. Furthermore, mapping techniques could also be modeled in MDA [44] way, so MDA application could be considered as a pure model-based approach. Following the generated PSM and default program code, the rest work is to build the real functional object and component in the software implementation phase, and then validate it with related PIM or PSM

Besides, as seen in Figure 51, there are mapping techniques to translate one PIM to another PIM, and PSM-to-PSM as well. Therefore, this enhances the interoperatability of heterogeneous system based on different PIM or PSM. The mapping techniques also enforce the quick development of new system, especially in the system modeling if there are some existing PIM or PSM with same business logic of new system. In fact, the developer could use MDA [44] tools to realize this kind of inter-connection (automating bridge called in MDA) between the respective MDA model and corresponding implementation, like below:

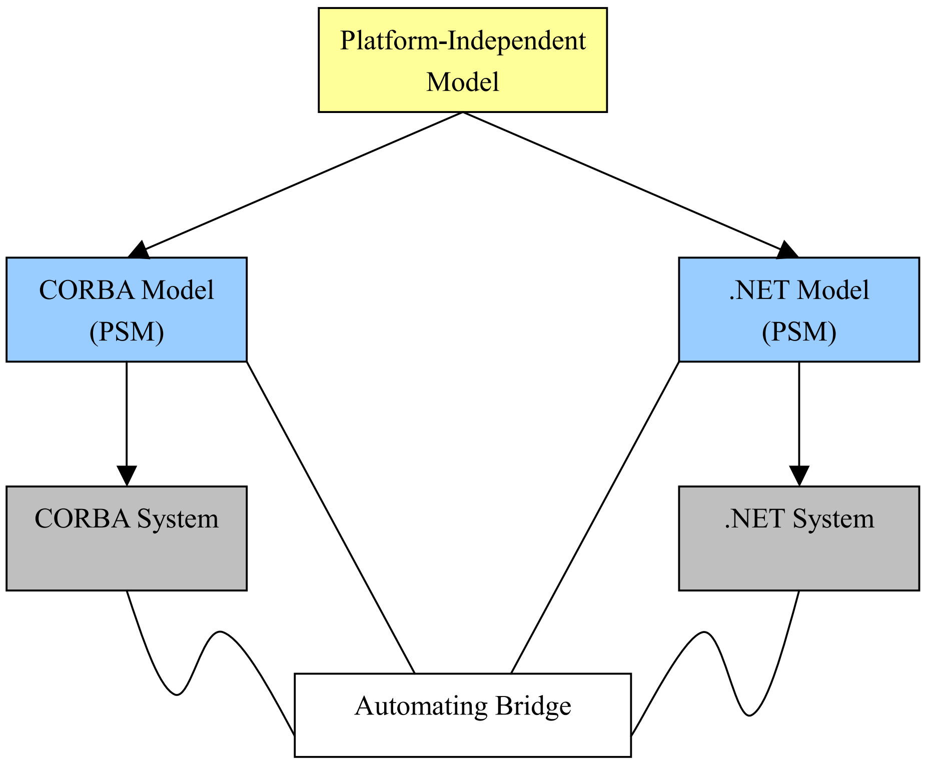


Figure 55: Automating Bridge in MDA

Since MDA [44] targets enterprise computing and network-oriented application, it is defining some general services called Pervasive Service. These services should be supported no matter the specific technology or platform. Now there are four general services in MDA environment:

1. Directory service
2. Transaction service
3. Security service
4. Distributed events and notification service

MDA [44] also has the tools for the legacy system or program, which could be reverse-engineered to discover its PSM or PIM. Then the legacy PIM or PSM could be integrated to the new MDA model. Even the legacy code would be wrapped or redeployed automatically or partial-automatically to be a part of implementation.

From the above introduction of MDA [44], it is clear that MDA is a new method for system development, to avoid the big overhead in the system design and maintenance due to the effect from all kinds of technology and platform. MDA increases the automation and reuse of design and meet the unpredicted change in the future. Thus it reduces the technique investment cost in a long term. However, MDA is still be developed by OMG with other evolving standards, and it need more support and attention to obtain its goal and benefits for IT industry.

References

1. Nenad Medvidovic and Richard N. Taylor. ***A Classification and Comparison Framework for Software Architecture Description Languages***. IEEE Transactions on Software Engineering, VOL. 26, NO. 1, January 2000.

2. Christine Choppy, Pascal Poizat and Jean-Claude Royer. ***Formal Specification of Mixed Components with Korrigan***. In proceedings of the 2001 International Conference on Asian-Pacific Software Engineering Conference (APSEC'2001), IEEE Computer Society Press, December 2001, Macau, China.

3. Liang Peng, Annya Romanczuk and Jean-Claude Royer. ***A Translation of UML Components Into Formal Specification***. In proceedings of TOOLS East Europe 2001, July 2001.

4. P.Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. ***The Many Faces of Publish/Subscribe***. Technical report, Swiss Federal Institute of Technology in Lausanne (EPFL), 2001.

5. Pascal Poizat, Christine Choppy and Jean-Claude Royer. ***From Informal Requirements to COOP: a Concurrent Automata Approach***. In proceedings of the **Formal** Methods, World Congress on **Formal** Methods in the Development of Computing Systems (FM'99), LNCS 1709:939-962, Springer-Verlag, Toulouse, France, 1999.

6. Martin Gogolla. ***UML for the Impatient***. Research Report 3/98, University Bremen, 1998.

7. Jean-Claude Royer. ***Comportement dynamique des composants asynchrones***. Research Report, Ecole des Mines de Nantes, March 11, 2003.

8. John D.Poole. ***Model-Driven Architecture: Vision, Standards, And Emerging Technologies***. Position paper submitted to ECOOP 2001, Workshop on Metamodeling and Adaptive Object Model, April 2001.

9. Tony Mallia. ***MDA Reality/Implementation***. MDA seminar series, April 2002.

10. Richard Mark Soley. ***Model Driven Architecture: An Introduction***. OMG White Paper, document 00-11-05, 2000.

11. Philippe Merle. ***Component-Based Engineering in MDA***. In summer school "MDA for Embedded System Development", Brest, Brittany in France, September 17th, 2002

12. TADAO MURATA. ***Petri Nets: Properties, Analysis, and Applications***. In Proceedings of the IEEE, VOL. 77, NO. 4, April, 1989.

13. Kyo C. Kang. ***Issues in Component-Based Software Engineering***. In the proceedings of International Workshop on Component-based Software Engineering, 1999.

14. Ye Wu. ***Introduction to Component-based Software Development (CBSD)***. http://www.isse.gmu.edu/~wuye/classes/ 699/lecture/699-Intro-2.pdf, August 28, 2002.

15. E. James Whitehead Jr., Jason E. Robbins, Nenad Medvidovic and Richard N. Taylor. ***Software Architecture: Foundation of a Software Component Marketplace***. In Proceedings of the ICSE17 Workshop on Architectures for Software Systems, Seattle WA, April 24-25, 1995.

16. Kirby McInnis. ***Component-based Design and Reuse***. http://www.cbd-hq.com/articles/1999/990715_cbdandreuse.asp, 1999

17. David S. Rosenblum and Richard N. Taylor. ***Component-Based Software***. ICS 221, fall 2002.

18. Paul Krause. ***Component-Based Software Engineering***. http://www.computing.surrey.ac.uk/courses/csm15/CSM-15_Lecture_2.ppt.

19. Pascal Poizat, Christine Choppy and Jean-Claude Royer. ***Specification of Mixed Systems in Korrigan with the Support of a UML-inspired Graphical Notation***. In H. Hussmann (Ed.), FASE'2001 - Fundamental Approaches to Software Engineering, Genova, Italy. Lecture Notes in Computer Science (LNCS) 2029:124-139, Springer-Verlag. (c) Springer-Verlag. 2001

20. Craig Thompson and Frank Manola. ***Component Software Glossary***. http://www.objs.com/survey/ComponentwareGlossary.htm, 1997

21. David Garlan and Mary Shaw. ***An Introduction to Software Architecture***. Carnegie Mellon University Technical Report, CMU-CS-94-166, January 1994.

22. Pascal Poizat, and Jean-Claude Royer. ***Korrigan: A Formal ADL with Full Data Types and Temporal Glue***. Research Report, University d'Evry Val d'Essonne,

Ecole des Mines de Nantes, June 27, 2003.

23. David E. Bakken. ***MIDDLEWARE***. in Encyclopedia of Distributed Computing, Kluwer Academic Press, 2001.

24. Allen Parrish, David Hale, Brandon Dixon and Joanne Hale. ***A Case Study Approach to Teaching Component Based Software Engineering***. Submitted to the CSEE&T 2000 Conference on Software Engineering Education and Training, September 1999.

25. Olaf Kummer. ***Introduction to Petri Nets and Reference Nets***. Sozionik aktuell, No.1, 2001. ISSN 1617-2477.

26. Peter J. Denning. ***Reuse Practices***. Invited paper for Communications of ACM, February 1997.

27. Smalltalk. http:// www.smalltalk.org.

28. Mary Campione. ***The Java Tutorial***. http://java.sun.com/docs/books/tutorial/index.html, 2000.

29. Java. http://java.sun.com/.

30. Middleware. http://www.omg.com/middleware/.

31. TCP/IP.http://www.protocols.com/pbook/tcpip1.htm.

32. EMN Component Research. http://pcguehen.info.emn.fr/ocm-wiki/Composants.

33. ComponentJ. http://www-ctp.di.fct.unl.pt/~jcs/ComponentJ/.

34. UML. http://www.omg.org/uml/.

35. C#. http://msdn.microsoft.com/vcsharp/.

36. EJB. http://java.sun.com/products/ejb/.

37. .NET. http://www.microsoft.com/net/.

38. CORBA. http://www.corba.org/.

39. ProActive. http://www-sop.inria.fr/oasis/ProActive/.

40. Franz Achermann and Oscar Nierstrasz. ***Applications = Components + Script s- A***

Tour of Piccola. In Mehmet Aksit, editor, Software Architectures and Component Technology, Kluwer, 2001.

41. Sharpie. James R. Larus and Sriram K. Rajamani and Jakob Rehof. ***Behavioral Types for Structured Asynchronous Programming*** . Technical report, Microsoft Research Technical Report, 2003. http://www.research.microsoft.com/behave/sharpie-report-abs.html.

42. RMI. http://java.sun.com/products/jdk/rmi/.

43. JVM (Java Virtual Machine). http://java.sun.com/docs/books/vmspec/.

44. MDA. http://www.omg.org/mda/.

45. CWM and MOF. http://www.omg.org/technology/cwm/.

46. ObjectWeb Consortium. http://consortium.objectweb.org/.

47. Jini. http://www.jini.org/.

48. Globus. http://www.globus.org/.

49. SOFA. http://nenya.ms.mff.cuni.cz/projects.phtml?p=sofa&q=0.

50. Denis Caromel. ***Toward a method of Object-Oriented Concurrent Programming***. Communications of the ACM, vol. 36, no. 9, September 1993.

51. G. Denker, J. Meseguer, and C. Talcott. ***Protocol Specification and Analysis in Maude***. In N. Heintze and J. Wing, editors, Proc. of Workshop on Formal Methods and Security Protocols, Indianapolis, Indiana, 25 June 1998.

52. G. Denker, J. Meseguer and C. Talcott. ***Rewriting Semantics of Meta-Objects and Composable Distributed Services***. Technical Report, Computer Science Department, Stanford University, 1999.

53. Paola Inverardi, Alexander L. Wolf and Daniel Yankelevich. ***Static Checking of System Behaviors Using Derived Component Assumptions***. ACM Transactions on Software Engineering and Methodology, Vol 9, No 3, July 2000.

54. N. Kaveh and W. Emmerich. ***Deadlock Detection in Distributed Object Systems***. In V. Gruhn (ed). Proc. of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, ACM Press. 2001