다수모델을 이용한 객체지향적 분산처리 시스템의 디자인 방법

이 상 범[†]

요 약

하드웨어와 통신 기술의 발달로 가능하게 된 분산처리 시스템은 수행 방식이 비결정적이고 복잡하기 때문에 소프트웨어의 디자인이 비분산처리 시스템에 비해 복잡하고 또한 까다롭다. 따라서 하드웨어에 비해 상대적으로 뒤져있는 분산처리 시스템 소프트웨어 개발을 위한 디자인 방법에 관한 연구에 대한 필요성이 높아지고 있다. 한편 객체지향 시스템과 분산처리 시스템은 상호 유사한 점이 있어, 분산처리 시스템 개발에 객체지향 기술의 적용은 자연스럽게 이루어질 수가 있다. 본고에서는 분산처리 시스템 설계를 위한 객체지향 기술의 적용에 대하여 소개하고 있다. 제안된 설계방법은 다양한 그래픽 모델, 즉, 자료구조도, 상태전이도 그리고 페트리 네트의 정보를 조합하여 객체와 행위 그리고 이들 간의 관계에 대한 정보를 추출하여 분산처리 시스템에 적합한 명세서를 산출하고자 한다. 지식베이스를 정보저장소로 이용하여 정보의 저장, 검색 뿐만 아니라, 정보의 오류 여부를 검증할 수가 있다. 본 방법의 최종 결과인 객체모델은 디자인 명세서로서 분산처리 프로그램 개발에 사용된다.

Integrated Modeling of Distributed Object-Oriented Systems

Sangbum Lee[†]

ABSTRACT

The design of distributed systems is difficult to achieve as the execution patterns of distributed systems are typically more complex than those of non-distributed systems. Thus, research toward the development of design methods for distributed systems is quitely needed. As object-oriented systems and distributed systems share similar properties, the combination of these two is somehow natural. In this work, a design method of applying object-oriented techniques to the design of distributed systems is introduced. The goal of the method in this paper is to provide assistance to the process of specifying a formal object-oriented specification from graphical representation specification inputs such as data flow diagrams, state transition diagrams and Petri nets. It addresses the extraction of objects, operations and relationships from the problem domain with emphasis on the specification of the characteristics of distributed systems. This object identification method is supported by a

† 정 회 원 : 단국대학교, 전자계산학과

논문접수 : 1995년 5월 3일, 심사완료 : 1996년 3월 8일

knowledge base that provides for the automated analysis and reasoning about objects and their relationships. The final object model is represented in a format which provides a formal mechanism for representing the object information.

1. Introduction

Distributed computing systems are systems in which multiple processors with their own memories run independently by communicating with each other. The design of distributed systems is difficult to achieve as the execution patterns of distributed systems are typically more complex than those of non-distributed computing systems. Thus, research toward the development of design methods for distributed systems is quitely needed [1]. In this work, a method of applying object-oriented techniques to the design of distributed systems is presented. Object-oriented approaches have received attention since the early 1980s. With these approaches, a system is developed based on objects, operations, and their relationships. Object-oriented techniques have shown promise features due to their powerful characteristics including modularity, information hiding, inheritance and reusability [2, 4].

Meanwhile, as complex requirements cause the system size to become large, the need for effective techniques to design large systems increases. Formal methods which span the analysis and the design phases are needed for large scale systems. There is currently more research toward the development of notations and techniques for specification models that to the development of support tools for large-scale specification [13]. This method integrates information from multiple models to specify objects, object behavior and relationships among objects from a distributed object-oriented viewpoint. Multiple modeling techniques are typically used to specify a system as different models specify the system from different viewpoints. When a system is specified by a set of different models, correct integration of such information in order to derive a system specification is a critical task.

A goal of the method presented in this paper is to provide assistance to the process of specifying a formal object-oriented specification from graphical representation specification inputs, including data flow diagrams (DFDs), state transition diagrams (STDs) and Petri nets. In Section 2, distributed object-oriented systems are reviewed. The method is discussed in Section 3 in detail. Finally, Section 4 contains the summary.

2. Specification of Distributed Object-Oriented Systems

As object-oriented systems and distributed systems share similar properties, the combination of these systems is somewhat natural. To identify parallelism in software, the main objective is to decompose the system into modules that can execute in parallel. An object in an object-oriented system inherently has a suitable form for a distributed system [16]. There are two ways to exploit parallelism in distributed object-oriented systems. One way is to define an object and a process separately, and the other way is to regard an object and a process as the same parallel execution component. The second approach is more widely used for distributed object-oriented systems because the parallel execution components, objects, can be easily identified [15]. In this work, we assume that only active objects can be activated initially and execute concurrently, i.e., passive objects can be activated after they receive messages from active objects.

There are four different approaches to specifying distributed systems: transition-oriented, language-oriented, hybrid, and algebraic. In the transition-oriented approach, a system is represented by states and events. Finite state machine, Petri nets, and IC* [3] are representative models for this group. These models have graphical formats, thus providing relative ease of understanding of executional aspects. In a language-oriented approach, the computational aspects of a system are represented by formal specification languages or programming languages. The advantages of this approach include easy implementation, strong modeling power and ease of modeling data transfer. It also offers some disadvantages such as difficulty of modeling logical properties, difficulty of achieving automatic implementation and increased complexity according to the size of a system. A hybrid approach is the combination of the transition-oriented approach and language-oriented approach. In an algebraic approach, axioms are used to specify the requirements of systems [17] and abstract data types and processes are specified in the form of algebras. CCS [10] is a representative model of the algebraic model in which the process is specified in an algebraic pattern [14]. While a transition-oriented approach provides for easy of system modeling and understanding, the language-oriented approach enhances the implementation process. In this work, we derive a language-oriented specification model by integrating multiple transition-oriented models.

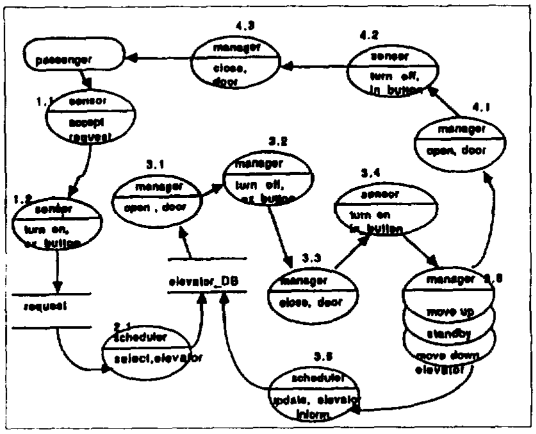
3. Object-Driven Specification Method

This object-driven method is an integrated, formalized method for identification of objects, object properties and object behaviors from multi-model formats. It addresses the extraction of objects, operations and relationships from the problem domain with emphasis on the specification of the characteristics of distributed systems. This object identification method is supported by a knowledge base that provides for the automated analysis and reasoning about objects and their relationships. The final object model is represented in a format which provides a formal mechanism for representing the object information. It also provides constructs that allow for refinement of the specification. The five sequential steps for this method are discussed in the following subsections.

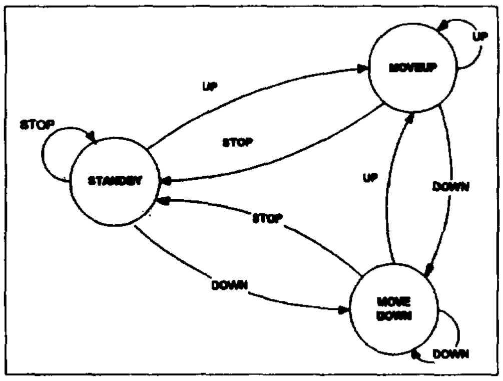
3.1 Develop graphical representations

Since a system is typically difficult to be completely represented by a single model, multiple representations are used in order to specify a system from different viewpoints. For instance, the initial problem is frequently represented with informal representation techniques, such as the DFDs and entity relationship diagrams. As the definition of the requirements proceeds, more formal methods, such as the STDs and the Petri nets, are used to show control and dynamic behavior. Among many different modeling techniques, we have selected three widely used models, DFD, STD and Petri nets, to specify the initial problem domain information. However, as these various techniques represent different viewpoints of the application, a technique which combines the requirements from the different models to produce an integrated specification is required.

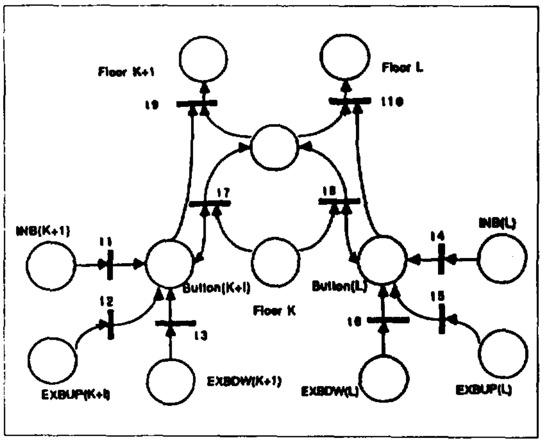
We discuss the process of the method by applying it to an example, an elevator system which works on **m** floors has **n** elevators. This problem is selected as it is widely used as a specification example [5, 8]. A



(Fig. 1) The data flow diagram of the elevator system portion of the representation of the elevator problem with the DFD, STD and Petri net is illustrated in Figures 1, 2 and 3, respectively. While the DFD emphasizes the overall view of the system, the Petri net and the STD describe a single component of the system i.e., the dynamic behavior of process 3.6 (elevator movement) is specified by a STD and a Petri net only for the demonstration. The specification of the other parts of the system with the Petri nets and the STDs is not included in this paper. The method is defined in Sections 3.2 through 3.5.



(Fig. 2) The state transition diagram for the elevator movement



(Fig. 3) The Petri net for the elevator upward movement

3.2. Convert each representation into internal format

Initially, the graphical notation of each input model is converted into an internal representative form and stored in the knowledge base. Figure 4 shows the internal format for the DFD information. Each component has its own identification number and description. If a component does not have a description in its original model, it remains as a null in the internal format. For example, *process2(id, [process_name], [function_name])* denotes a process in the level 02 DFD. Other models have similar internal formats. The internal formats for the Petri net and the STD are similar to that of the DFD. The internal representation of the Petri net in Figure 3 is given in Figure 5. If any component in the graphical representation of each input model is not fully described, it needs to be specified clearly in the internal representation. For example, transitions and places in the Petri net in Figure 3 are denoted as abbreviated symbols but the internal representation in Figure 5 has a full description for each component. This form of internal representation provides the ability to deal with incomplete information [9].

Early validation of requirements is important because early detection of errors reduces software development costs. Each model's internal form is validated according to pre-defined rules to help to detect inconsistent information. If inconsistent facts are found, the information is modified and reentered. The DFD evaluation process is assisted by a regenerated informal English document [9]. The Petri net can be

```
source2(id,[source_name]).
sink2(id,[sink_name]).
process2(id,[process_name],[function_name]*).
dst2(id,[datastore_name]).
dfw2(id,[dataflow_name]).

Note : [function_name] := [action,object]
```

(Fig. 4) Data flow diagram's internal format

```
tran(1,t_01,['set request floor(K+1)']).  tran(1,t_02,['set request floor(K+1)']
tran(1,t_03,['set request floor(K+1)']).  tran(1,t_04,['set request floor(L)']).
tran(1,t_05,['set request floor(L)']).    tran(1,t_06,['set request floor(L)']).
tran(1,t_07,['move up for floor(K+1)']).  tran(1,t_08,['move up for floor(L)']).
tran(1,t_09,['stop at  floor(K+1)']).     tran(1,t_10,['stop at floor(L)']).
place(1,p_01,['In_button(K+1)= ON']).     place(1,p_02,['Up_button(K+1)= ON']).
place(1,p_03,['Down_button(K+1)=ON']).    place(1,p_04,['In_button(L)=ON']).
place(1,p_05,['Up_button(L)=ON']).        place(1,p_06,['Down_button(L)=ON']).
place(1,p_07,['Staying  floor(K)']).      place(1,p_08,['Floor(K+1) requested'])
place(1,p_09,['Floor(L) requested']).     place(1,p_10,['Arriving floor(K+1)']).
place(1,p_11,['Staying floor(K+1)']).     place(1,p_12,['Staying floor(L)']).
%connection between transitions and places
arc(1,[p_01],[t_01],[p_08]).              arc(1,[p_02],[t_02],[p_08]).
arc(1,[p_03],[t_03],[p_08]).              arc(1,[p_04],[t_04],[p_09]).
arc(1,[p_05],[t_05],[p_09]).              arc(1,[p_06],[t_06],[p_09]).
arc(1,[p_07,p_08],[t_07],[p_08,p_10]).    arc(1,[p_07,p_19],[t_08],[p_09,p_10]).
arc(1,[p_08,p_10],[t_09],[p_11]).           arc(1,[p_09,p_10],[t_10],[p_12]).

pconnect(1,p_36,[elevator]). % this Petri net is related to process 3.6
```

(Fig. 5) The internal representation of a Petri net

validated by using the formal properties of the Petri nets, such as a reachability tree [12]. By counting the number of token at the particular places, undesirable execution of the Petri nets can also be detected.

3.3 Build the knowledge base

The next step is to build the knowledge base. It consists of the internal information of the input models, defined rules, and the information derived by the rules. A set of rules derives additional information from initial information. The rules for extracting information from each input model are discussed in Section 3.3.1 thru Section 3.3.3. Rules for integration of information and generation of a specification model are introduced in Section 3.4 and Section 3.5, respectively. As each input model represents a different viewpoint, we obtain different types of information from different models. While the DFDs are introduced to extract the objects, operations and their relationship, the STDs and the Petri nets are used to extract detailed internal behavior of objects and/or operations.

3.3.1 Information from the data flow diagrams

The DFD is the main source of objects and operations. The frame of the specification method is based on the information in the DFD. We have developed a method which uses a set of DFDs to build an architectural view of object-oriented systems [9]. This method builds on that strategy. With this method, a set of objects are extracted from any component of the DFD, including the function name. Most objects extracted from the DFD become solution space objects since the non-solution space objects are eliminated during the design of the DFD. We mark all objects selected from processes, data stores and function names as solution space objects. Objects from sources and sinks are considered as problem space objects. Solution space objects are divided into active objects and passive objects. Objects that are extracted from the process names are classified as active objects and all other objects are classified as passive objects.

Operations which are related to a particular object are identified. The matching an object and its related operations is not straightforward. We extract the operation explicitly from the function name in the process which consists of an action-object fair. In addition, implied operations are identified from the relationship in the DFD components. During the design of the DFD, a main function is typically converted into a process, however, some operations can be denoted by the relationship of the components. The inheritance feature of object-oriented systems enhances reusability and extendibility of software systems [11]. Thus, classification of objects is a very important part of any object-oriented approach. The DFD neither specifies the objects nor explicitly denotes the hierarchical relationships between objects. If two different levels of physical DFDs are provided, we classify the objects heuristically. When a process in the high level DFD is expanded to several processes, the process name in the high level DFD become a class object and the instance objects are identified from the corresponding processes names in the low level DFD.

Visibility is defined by identifying the objects which are related to a specific object. We define the visibility of an object from the connected relationship in the DFD. While an active object can see a set of active objects and a set of passive objects, a passive object only can be seen from the related active objects. By identifying the visibility of objects, we establish the communication routines between objects.

3.3.2 Information from the state transition diagram

From the STDs, the states and events of all or part of the system are extracted. The extraction of information from the STD is straightforward as states and input symbols correspond to conditions and events, respectively. The STD to describe the behavior of *process 3.6* in DFD is given in Figure 2. A STD which represents the behavior of the elevator movement is given. The information extracted from this representation is shown in the final object module in Figure 7.

3.3.3 Information from Petri Nets

As we derive the frame of an object model from a set of the DFDs, a set of Petri nets that specifies each component (object or operation) is also used to identify behavior. Dynamic behavior of the objects can be extracted from the Petri nets.

The interpretation of the Petri nets is similar to that of the STD: the places and transitions correspond to conditions and events/actions, respectively. Since we use a set of Petri nets in which each Petri net represents a component (an object or an operation), specifying the communication aspects between objects from the Petri nets is difficult. Thus, we extract the communication routines from the DFD information.

3.4 Synthesize the input

Integration of the information of the three input models is the critical step in this method. A frame model, mainly extracted from the DFD, is constructed with a set of objects and the primary operations. This frame is the definition part of the module. The identification of active type objects is very important, as we regard that only active object modules can execute concurrently by message passing [6]. After the objects and operations are extracted from the DFDs, the detailed behavior is specified by the STDs or the Petri nets. For flexibility, we do not require both the STDs and the Petri nets. The internal behavior and properties in the body part of each module are specified with a generic format.

Integration rules are as follows. The integration process is performed with identification numbers because each model's component is denoted by its own identification number. Since the DFD represents the overall view of a system, information from the DFD becomes a framework for other models. Thus the internal representations for the Petri nets and the state STD must contain extra facts which indicate the relationship between them and the components of DFDs. Moreover, internal representation of each

Petri net and each STD should have two identification slots; one for itself and one for the relationship with the DFD. For example, the fact, pconnect(*1*, *p_36*, [*elevator*]), in Figure 5 is introduced to show that the Petri net which has identification '1' is related to the *process 3.6* in the DFD. The connectivity between the DFD and the STD is specified in the same way.

3.5 Generate an object-oriented specification

The object module format is shown in Figure 6. This generic object specification model consists of two parts: definition and body. In the definition part, the environment and the frame of the object module are specified. If the information is not available from the input models, the frame is empty. The informal requirements document, which is shown in Section 3.1, can be used as comments to help the user. The input/output data items of each object module are extracted from the data dictionary. The information from the DFDs is used to specify the definition part. In the body, the message passing methods are identified with ***send*** and ***accept*** operations. The internal behavior of an object is specified with the CSP[7]-like format. The syntax and semantics of this format is not included in this paper. The body of an object module is specified with the information from the Petri nets and the STDs.

```
ObjectModule :: [object_name]
  Definition_part is
      object_type : {passive,active}
      inherit : {class objects}
      visible : {visible objects}
      data_type : {data items}
    method  action1 ( );
      . . . . . . .
 Body_part is
  accept (message) from [object]; {accept a message}
  send  (message) to [object];    {send a message}
 When ==> action1 ( );
     *[ .......... ]
 When ==> action2 ( );
      . . . . .
End
```

(Fig. 6) The general format of an object module

In Figure 7, a list of the active objects and passive objects and one of the object modules derived from the elevator problem are illustrated. The final object specification consists of a set of passive and active object modules which are derived in the same man-

```
Active Objects  : [Sensor] [Scheduler] [Manager]
Passive Objects : [door] [in_button] [ex_button]
                  [elevator] [request]  [elevator_DB]

ObjectModule :: [elevator]
 Definition_part is
     object_type : passive
     inherit :
     visible : [Scheduler] [Manager]
     data_type : K,L := integer; {floor number, K < L}
       method select ( );
       method move_up ( );
       method move_down ( );
       method stay ( );
Body_part is
        accept (message) from [Scheduler];
        select  (  );      --:  The  Scheduler  selects  an
available elevator
        --: Petri net for this part is not specified
        accept (message) from [Manager]; --: The Manager
moves the elevator
        When ==> move up ( );
*[{precondition::             ((In_button(K+1)=ON          or
Up_button(K+1)=ON or Down_button(K+1)=ON) and state =ANY)
                 set request floor(K+1)::
 postcondition :: (Floor(K+1) requested and  state=ANY)
[]
{precondition :: ((In-button(L)=ON or Up_button(L)=ON or
Down_button(L) =ON) and  state = ANY)
                 set request floor(L)::
postcondition :: (Floor(L) requested and state = ANY)}
[]
{precondition   ::(Floor(K+1)   requested   and   Staying
floor(K) and state =ANY)
                 move up for floor(K+1)::
postcondition  ::  (Arriving  floor(K+1)  and  Floor(K+1)
requested and state = MOVEUP)}
[]
(precondition :: (Floor(L) requested and Staying floor(K)
and state = ANY)
                 move up for floor(L)::
postcondition  ::  (Arriving  floor(K+1)  and  Floor(L)
requested and state = MOVEUP))
[]
{precondition  ::  (Floor(K+1)  requested  and  Arriving
floor(K+1) and state = MOVEUP)
                 stop at  floor(K+1)::
postcondition :: (Staying floor(K+1) and state=STANDBY)}
   []
{precondition::(Floor(L)    requested    and    Arriving
floor(K+1) and state=MOVEUP)
                 stop at floor(L)::
postcondition :: (Staying floor(L) and state=STANDBY)]

When ==> move down ( );
  --: no representation with Petri net
    {precondition :: state = ANY
                  move down ::
     postcondition :: state = MOVEDOWN}

When ==> stay (  );  --: no representation with Petri net
    {precondition :: state = ANY
                 stop ::
     postcondition :: state = STANDBY}
 End
```

(Fig. 7) The object specification module of the elevator ner. An active object module acts as a monitor which controls the execution of the passive object modules and executes in parallel with other active object modules.

4. Summary

This paper addresses the formal specification of distributed systems. It presents an object-oriented specification method that consists of analyzing requirements from multiple modeling formats and integrating them into a high level specification model. There is a general lack of supporting tools and methodologies to assist the specifier with the assistance for writing formal specifications. We have developed a method to help address this problem by providing automated support that has the potential to provide assistance for large-scale software development. The method provides flexibility as it does not require all three input models but has the capability to integrate all three models. The DFD is the minimum requirement for this method.

The major advantage of this method is that it provides an environment that includes a partially automated technique which helps to save time to eliminate inherent mistakes that happen during the manual process. In addition, by establishing a knowledge base, the method becomes a semi-automatic method and potentially provides the support for large-scale software development. Moreover, it contains a technique to integrate the different models of the system. In the general case, multiple modeling techniques are used in order to represent a system from different viewpoints. However, for the design of the system, there is a need for a method that can combine the requirements, in a well-defined manner, from the different models to produce an integrated specification. Another contribution of this method is that it is useful as an assistant for the novice system specifier who is not familiar with the object-oriented development techniques by converting functional-oriented representations into an object-oriented representation.

In [18], most of currently developed OOD methods and techniques are analyzed and the result of the analysis is summarized in the table. This method is the extension of the Lee&Carver's method [9]. Compared to other OOD techniques such as OMT and BOOCH's method, this method emphasizes the process of the objects and operations from the well-known functional modeling approaches rather than representation. Even the OMT is widely known and used in now, it does not support the technique how to identify the appropriate objects and other information. The main difficulty in OOP is identifying the objects and their related information.

A formal specification language which is appropriate for this method is under development. This language should provide for the formal specification of the characteristics of distributed object-oriented systems. An unavoidable constraint of the method is that the final specification model is dependent on the input models, that is, errors or incompleteness in the input models result in an incomplete or erroneous object model.

References

[1] H. E. Bal, "Programming Lang. for Distributed Computing Systems" ACM Computing Surveys, Vol. 21, No. 3, Sept., pp. 261-322, 1989.

[2] G. Booch, Object-Oriented Design with Applications, The Benjamin/Cummings, Redwood city, CA, 1991.

[3] E. J. Cameron et. al., "The IC* Model of Parallel Computation and Programming Environment," IEEE Trans. Comm., Vol. COM-28, Apr., 1980.

[4] P. Coad and E. Yourdon, Object-Oriented Analysis, Yourdon Press, Prentice Hall, Englewood Cliffs, 1990.

[5] C. Ghezzi, M. Jazayeri and D. Mandrioli, Fundamentals of Software Engineering, Prentice

Hall, Englewood Cliffs, NJ, 1991.

[6] H. Gomma, "Structuring Criteria for Real Time System Design", Proc. 9th International Conf. on Soft. Eng., 1990.

[7] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall Intl., 1985.

[8] G. R. Kampen, "An Eclectic Approach to Specification", International Workshop on Software Specification and Design. Washington, D. C., IEEE Computer Society Press., pp. 178-182, 1987.

[9] S. Lee and D. Carver "Object-Oriented Analysis & Specification: A Knowledge Base Approach", Journal of Object-Oriented Programming, pp. 35-43, Jan. 1991.

[10] R. Milner, "A calculus of communicating systems", Lecture notes in computer science, Vol. 92, Springer-Verlag, Berlin, 1980.

[11] B. Meyer, Object-Oriented Software Construction, Prentice Hall Int'l, Englewood Cliffs, NJ, 1988.

[12] J. L. Peterson, Petri Net Theory and the Modeling of Systems. Prentice-Hall, Inc., Englewood Cliffs, N. J. 1981.

[13] I. Sommerville, Software Engineering (3rd ed.), Addison-Wesley Inc., 1989.

[14] S. J. Song, "The Modeling, Analysis, and Design of Distributed Systems Based on Communicating Petri Nets", Ph. D dissertation, Department of Electrical Engineering and Computer Sciences, UC Berkeley, 1988.

[15] Y. Yokote and M. Tokoro, "Concurrent Programming in Concurrent Smalltalk", Object-Oriented Concurrent Programming, A. Yonezawa and Tokoro (eds.) MIT Press, Cambridge, MA, pp. 9-36, 1987.

[16] A. Yonezawa and M. Tokoro, (ed.) Object-Oriented Concurrent Programming, The MIT Press, Cambridge, MA, 1987.

[17] J. M. Wing, "A Specifier's Introduction to Formal Methods", IEEE Computer, Sept., pp. 8-24, 1991.

[18] 양해술, 조영식, 이용근, "객체지향 설계 방법론의 비교 분석", 정보과학회지, Vol. 11, No. 2, pp. 32-41, 1993.

이 상 범

1983년 한양대학교 기계공학과 졸업
1989년 Louisiana State University(미) 전자계산학과 (이학석사)
1992년 Louisiana State University(미) 전자계산학과 (이학박사)
1992년~1993년 한국전자통신연구소, 선임연구원
1993년~현재 단국대학교 전자계산학과 조교수
관심분야: 객체지향 설계, Formal Specification Languages, 분산처리 데이타베이스 모델링