

A Study on Iconic Animation based on Object Modeling Technique

Suck-Tae Joung, *Member, KIMICS*

Abstract— We propose the iconic animation of the software requirement specifications by using the object and dynamic models of the object modeling technique (OMT) methodology. In order to produce the iconic animation, we use “graphical classes” and “icon transformations.” In general, the graphical classes are defined for each class of the object diagram. The icon transformations which show the activities of the application are constructed by considering the meaning of the activities and are defined by either basic or compound icons. The icon transformations are added to the state diagrams to generate extended state diagrams. The animation system generates the header files and the code instantiating GUI from the object diagram having graphical classes. The system also generates “event methods” from the extended state diagrams. When the event methods are executed, the behavior of the events is animated by the icon transformations.

Index Terms— Iconic Animation, Object Modeling Technique (OMT), Graphical Classes, Icon Transformation, Event Method.

I. INTRODUCTION

OBJECT Modeling Technique (OMT) is based on the development of three-part models of the system [1]. The object model, represented with object diagram, captures the objects in the system and their relationships. The dynamic model, represented with state diagrams, describes the reaction of objects in the system to events, and the interactions between objects. The functional model specifies the transformations of object values and constraints on these transformations. These models are created in analysis phase and are gradually refined in design phase. In implementation phase, the models are converted into executable code. CASE tools, such as Rational Rose and Rhapsody, can generate executable code from object or state diagrams [2] [3]. However, when requirement specifications are not correct, the generated executable code becomes useless. In software development, the most important thing is to

find the correct requirement specifications at the analysis stage. Animation is a powerful means of validating the requirement specifications. It helps both the user and analyst to understand the requirement specifications because the images represent real objects as well as their spatial and temporal relationships[4].

We propose the iconic animation of the software requirement specifications by using the object and dynamic models of the OMT methodology. The iconic animation means producing animation by moving icons and changing their appearance. A set of icons is associated with each class of an application. To represent icon movement and changes in their appearance we introduce the concept of icon transformation [5].

II. Lift Problem

We show the lift problem [6] to explain our method of iconic animation. This problem concerns the logic to move lifts between floors according to the five constraints [6]. We consider the lift system controlling 3 lifts and 6 floors as a case study.

A. The object and state diagrams

Figure 1 shows the object diagram of the lift problem. There is normally one class, the Controller [7] [8], that contains the control of the system. After initializing the system, the Controller transfers its control to the user of the system. So it is the interface between an external actor and the application. The value “6” (“3”) of multiplicity specifies the number of the instances of class Floor (Lift) which is related to a single instance of the associated class Controller.

Figure 2 shows the state diagrams of the lift problem. Each state diagram shows the behavior of a particular class. In the state diagram of the Lift, the state of the Lift changes from MoveState to StopAtState on the occurrence of the event StopAt.

B. Icon definition

Icons are visual symbols of the objects of software requirement specifications. It is important to define simple and understandable icons. If they are misinterpreted, the requirement specifications cannot be understood [9] [10]. We define some basic icons (Figure 3).

- Defining basic icons for each class of the object diagram : The icons LIFT-A, FLOOR-A and PERSON-A are defined due to the presence of the corresponding classes.

Manuscript received February 20, 2008; revised August 1, 2008. Suck-Tae Joung is with the Division of Electrical Electronic and Information Engineering, Wonkwang University, Iksan, 570-749, Korea (Tel: +82-63-850-6888, Email: stjoung@wonkwang.ac.kr)

- Defining basic icons for activities of the state diagrams : The icon LIFT-B is constructed from the activity OpenDoor in the state diagram of the Lift. The icons FLOORBUTTON-A and FLOORBUTTON-B are defined from the activities FloorButtonOff and FloorButtonOn in the state diagram of the Lift, respectively. The icons UPBUTTON-A and DOWNBUTTON-A are constructed from the activity UpDownRequestButtonOff in the state diagram of the Floor. The icons UPBUTTON-B and DOWNBUTTON-B are constructed from the activity UpDownRequestButtonOn in the state diagram of the Floor.

We construct some compound icons (Figure 4).

- Defining compound icons for activities, considering the states in which the corresponding activities are contained : The compound icon CLOSE is defined from the activity CloseDoor of StopAtState in the state diagram of the Lift. The state StopAtState represents that a lift arrives at the floor. This icon is formed by combining the basic icons LIFT-A and FLOOR-A. The compound icon OPEN is constructed from the activity OpenDoor of StopAtState in the state diagram of the Lift. This icon is formed by combining the icons LIFT-B and FLOOR-A. The compound icons MOVE-A and MOVE-B are constructed from the activity MoveNext of MoveState in the state diagram of the Lift. These icons are constructed by combining the basic icons LIFT-A and FLOOR-A. The compound icons GETON-A and GETON-B are defined from the activity PersonGetOn of GetOnState in the state diagram of the Person. The state GetOnState shows that a person gets on the lift arrived at the floor. These icons are constructed by combining the basic icons LIFT-B, FLOOR-A and PERSON-A.

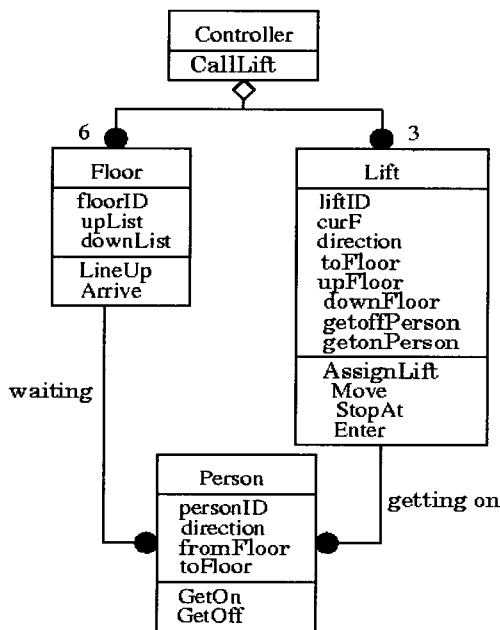


Fig. 1 Object diagram of the lift problem

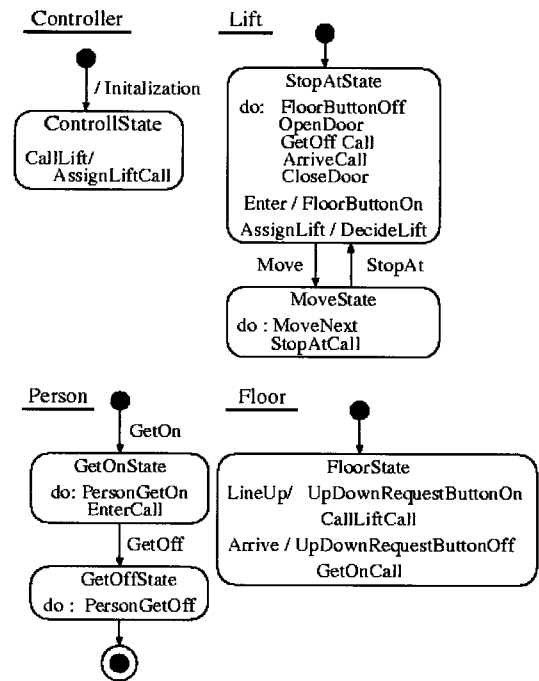


Fig. 2 State diagrams of the lift problem

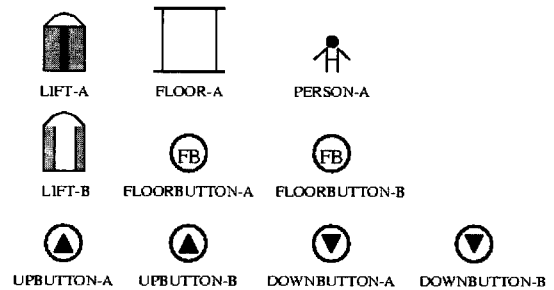


Fig. 3 Basic icons for the lift problem

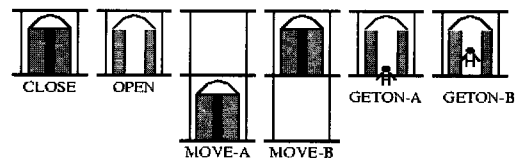


Fig. 4 Compound icons for the lift problem

C. The object diagram having graphical classes

To generate the GUI for the application, we define the object diagram having graphical classes. The icons and icon transformations which represent the objects and activities of the application become the types of the graphical classes. Figure 5 shows the object diagram having graphical classes of the lift problem. In general, the added graphical classes correspond to the classes of the object diagram. For example, the Controller, Floor, Lift and Person correspond to GController, GFloor, GLift and GPerson, respectively. For the icons of the floor buttons and up/down request button, we need the types of other graphical classes. We added GFloorButton, GUpButton and GDownButton classes to the object

diagram.

Because each floor has several display areas, the GController has reference of the GFloor type for referring each floor. The GFloor then maintains references of GLift, GPerson, GUpButton, GDownButton and GFloorButton types. The GController initializes the GFloor objects by receiving GUIInitialization message from the Controller.

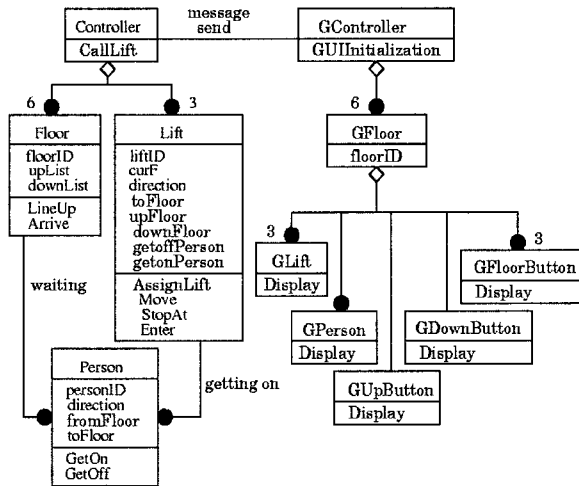


Fig. 5 Object diagram having graphical classes

D. Extended state diagrams having icon transformations

We explain guard conditions which are used as guards on transitions and add icon transformations which show animation for the corresponding activities to the state diagrams (Figure 6).

The guard condition on a transition is shown as a Boolean expression in brackets following the event name. A guarded transition fires when its event occurs, but only if the guard condition is true. For example, when StopAt event (state diagram of the Lift) occurs, if the condition StopAtNeeded is true, its guarded transition fires. The toFloor is a Boolean array whose elements that correspond to the destination floors of the Person objects become true. The upFloor (downFloor) is a Boolean array whose elements become true if the corresponding up (down) request button on the floor is pressed. For example, when GetOn event (state diagram of the Person) occurs, if Boolean array upFloor or downFloor is true in the current floor, its guarded transition fires.

The icon transformations are constructed by considering the meaning of the activities in the states. Usually they have names and are enclosed between “{“ and “}” symbols. They are composed of starting and ending icons and are enclosed between “<” and “>” symbols. These icons are defined by either basic or compound icons. For example, because the meaning of the activity FloorButtonOff (StopAtState of the Lift) animates “off floor button into the lift,” the starting and ending icons consist of the basic icons which are FLOORBUTTON-B and FLOORBUTTON-A. The icon transformation name of the activity FloorButtonOff is FloorButtonOffT. The icon transformation of activity

OpenDoor (StopAtState of the Lift) consists of the compound icons CLOSE and OPEN, because the meaning of this activity animates “open the door of the lift.” The icon transformation name of this activity is OpenDoorT.

Sometimes an activity has two icon transformations. If an activity has two icon transformations, we add a condition following the icon transformation name because of selecting an appropriate icon transformation. This condition is enclosed between “[“ and “]” symbols. For example, the activity UpDownRequestButtonOn (LineUp event of the Floor) has UpRequestButtonOnT and DownRequestButtonOnT. The condition which selects an appropriate icon transformation is that the request button pressed by Person object is whether up or down request button. UpDownButton is method which detects a kind of up/down request button. If Person object presses up request button (UpDownButton() == 1), UpRequestButtonOnT is displayed. If Person object presses down request button (UpDownButton() == -1), DownRequestButtonOnT is displayed.

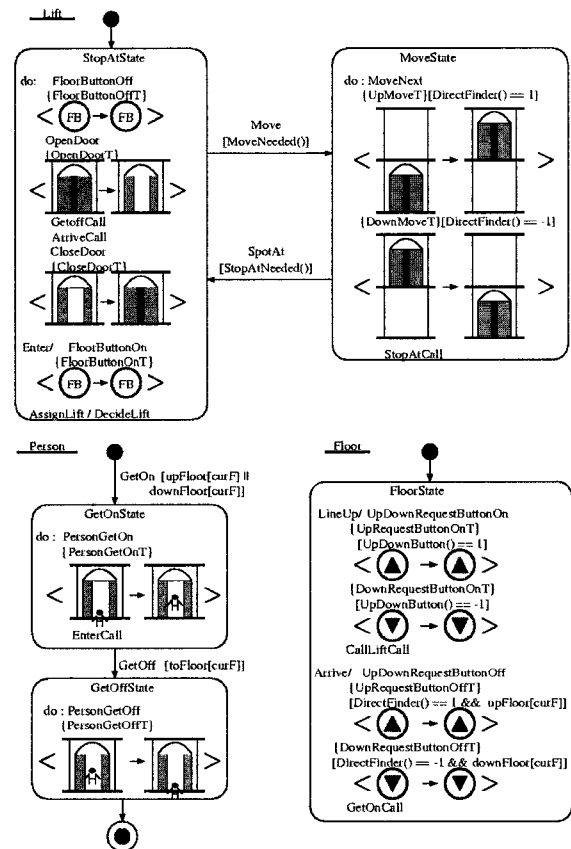


Fig. 6 Extension of the state diagrams

III. Generating animation code

A. Generating header files and the code instantiating GUI

After finding information about the classes in the object diagram having graphical classes, the header files for them and the code instantiating GUI are generated. In

general, attributes and operations in the object diagram having graphical classes become variables and headers of the event methods in the header files, respectively.

For example, the header files for the classes Controller and Lift in Figure 5 are generated as in Figure 7. The Controller class has aggregation relationship with Floor and Lift classes. The composite class, the Controller, contains floorList and liftList objects of types Floor and Lift classes, respectively. The main method which is used to initialize these objects is defined to the Controller class. The number of these objects is generated as many as the values “6” and “3” of multiplicity of Floor and Lift, respectively. In order to construct the GUI components, the GUIInitialization message is sent to GController object. The Lift object keeps its own control and does not wait for incoming messages from other objects. It executes a continuous loop and in each iteration it checks whether there is any outstanding request that should be serviced. That is why, the Lift class is defined as a Java thread. The continuous loop which calls Move method of the Lift is placed inside the run method.

```
class Controller {
    public static Floor[] floorList;
    public static Lift[] liftList;

    public static void main(String args[] ) {
        floorList = new Floor[6];
        liftList = new Lift[3];
        for (int i = 0; i < 6; i++)
            floorList[i] = new Floor(i);
        for (int i = 0; i < 3; i++)
            liftList[i] = new Lift(i);
        GController.GUIInitialization( );
    }

    public static void CallLift(int curF) { }
}

class Lift extends Thread {
    int liftID, curF, direction;
    boolean[] toFloor, upFloor, downFloor;
    Person[] getoffPerson, getonPerson;

    public void AssignLift( int dir, int curF) { }
    public void Move() { }
    public void StopAt(int curF) { }
    public void Enter(Person per) { }
}
```

Fig. 7 Definition of Controller and Lift

The code instantiating GUI is generated from the graphical classes. The header files for the graphical classes GController and GFloor are generated as in Figure 8. The GController class has aggregation relationship with GFloor class and contains gFloorList objects of type GFloor class. In the GUIInitialization method, the gFloorList objects are created as many as the value “6” of the multiplicity of the GFloor. The GFloor class has aggregation relationship with GLift, GPerson, GUpButton, GDownButton and GFloorButton classes. The GFloor class contains gLift, gPerson, gUpButton, gDownButton and gFloorButton objects of types GLift,

GPerson, GUpButton, GDownButton and GFloorButton classes, respectively. The number of these objects is decided to the value of the multiplicity of the corresponding classes. The GUIInitialization method is executed when receiving the corresponding message from Controller.

In the object diagram having graphical classes, there is no information about the body codes of the event methods. The body codes of the event methods can be generated after finding information in the extended state diagrams.

```
class GController {
    public static GFloor[] gFloorList;

    public static void GUIInitialization () {
        gFloorList = new GFloor[6];
        for (int i = 0; i < 6; i++)
            gFloorList[i] = new GFloor(i);
    }
}

class GFloor extends Panel {
    int floorID;
    GLift gLift[];
    GUpButton gUpButton;
    GDownButton gDownButton;
    GFloorButton gFloorButton[];
    .....

    public GFloor(int floor) {
        floorID = floor;
        .....
        for (int i = 0; i < 3; i++) {
            gLift[i] = new GLift(floor);
            add (gLift[i]); }
        for (int i = 0; i < 3; i++) {
            gFloorButton[i] = new GFloorButton(floor);
            add (gFloorButton[i]); }
        gUpButton = new GUpButton(floor);
        gDownButton = new GDownButton(floor);
        add (gUpButton); add (gDownButton);
        .....
    }
}
```

Fig. 8 Definition of GController and GFloor

B. Generating event methods

The information shown in the extended state diagrams is used to generate event methods of the classes. If the event in the extended state diagram is an internal one, the body code of the event method contains a method call which executes the associated action. If the event in the extended state diagram has a guarded transition, the body code of the event method contains the following method calls after checking the guard condition. Method calls to execute 1) the exit action of the current state, 2) the actions related to the transition, 3) the entry action of the new state and 4) the activities of the new state. For example, in the state diagram of Figure 9, the body code for event method event1 first calls action3 which is exit action in state1. Next, it calls action1 related to event1, and then calls action4 which is entry action. Finally, it calls activity2 in state2.

To be able to execute the generated body code of the event method, it is necessary to generate the body code of the methods that are called from inside event methods. We use icon transformations to define body code of these methods. For example, the event method StopAt in

Figure 6 checks the guard condition `StopAtNeeded` and calls `FloorButtonOff`, `OpenDoor`, `GetOffCall`, `ArriveCall` and `CloseDoor` methods. Figure 10 shows the body code for event method `StopAt` after using the icon transformations. The `FloorButtonOff` method displays icon transformation `FloorButtonOffT` of the `GFloorButton` type. Considering relationship links of the object diagram containing graphical classes, we use standard navigation expression, "`GController.gFloorList[curF].gFloorButton[liftID]`". Display ("`FloorButtonOffT`")." The expression to call the method in the graphical classes must start from `GController` class. The `ArriveCall` method calls `Arrive` method of the `Floor` class. We use navigation expression of "`Controller.floorList[curF].Arrive(curF)`". The expression to call the method in the original classes must start from `Controller` class. The `Display` is an animation method which displays the icon transformations. Executing this code produces animation for event method `StopAt`.

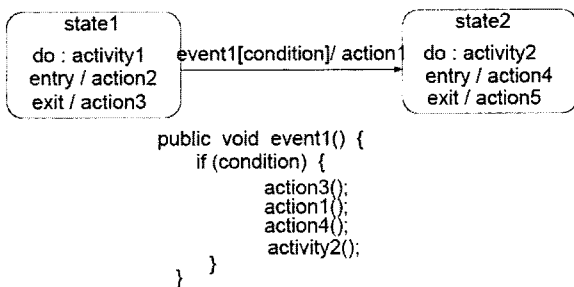


Fig. 9 Body code of the event1

```

public void StopAt(int curF) {
    if (StopAtNeeded()) {
        GController.gFloorList[curF].
            gFloorButton[liftID].Display("FloorButtonOffT");
        GController.gFloorList[curF].
            gLift[liftID].Display("OpenDoorT");
        Controller.liftList[liftID].
            getoffPerson[curF].GetOff(curF, liftID);
        Controller.floorList[curF].Arrive(curF);
        GController.gFloorList[curF].
            gLift[liftID].Display("CloseDoorT");
    }
}
    
```

Fig. 10 Body code of the StopAt

IV. Generating animation

For generating animation, the generated header files, the code instantiating GUI and event methods are used. When the main method of the `Controller` class is executed, the objects of the `Floor` and `Lift` are initialized, and `GUIInitialization` message is sent to the `GController` class for creating GUI components. The number of the initialized objects is decided according to the values of multiplicity in the object diagram having graphical classes. The `GController` object represents itself as in Figure 11. Figure 12 shows the snapshot of the animation for the lift problem.

To make the system interactive, each `GFloor` object contains up/down buttons at right in Figure 11, except ground and top `GFloor` objects. These buttons are clicked by user with mouse.

When the user clicks up/down buttons, the `Floor` and `GFloor` objects create a new `Person` and `GPerson` objects, respectively. Then the method `LineUp` of the `Floor` class is called. The `Floor` object adds new `Person` objects to the list of persons already waiting at the floor. The destination floor for a `Person` object is randomly determined when the object is newly created. If the user clicks up button, when the `LineUp` is executed, it is animated that up request button is illuminated. Then `Floor` object sends the `CallLift` message to the `Controller`. When the `CallLift` is executed, the `Controller` sends the `AssignLift` message to each of the `Lift` objects asking its convenience for servicing the request and assigns a lift to floor pressed up/down request button. If the `Lift` object is assigned, run method is executed and `Move` method is called. When `Move` is executed, it animates the lift moving between the floors. During moving between the floors, the `Lift` object calls `StopAt` method and checks guard condition `StopAtNeeded`. If `StopAtNeeded` condition is satisfied, the body code of `StopAt` method is executed. When `StopAt` is executed, it animates the floor button inside the lift being reset and the door of the lift being opened. Then `GetOff` message is sent to `Person` object and `Arrive` message is sent to `Floor` object. Finally the action closing the door of the lift is animated. When `GetOff` is executed, it is animated that persons get off in the lift. When `Arrive` is executed, it animates the up or down request button being reset, and `GetOn` message is sent to `Person` object. When `GetOn` is executed, it animates the persons getting on the lift, and `Enter` message is sent to `Lift` object. When `Enter` is executed, it animates the floor button which is corresponded to destination floor for a `Person` object being illuminated. If the `Lift` object has any request, `Move` method is called repeatedly.

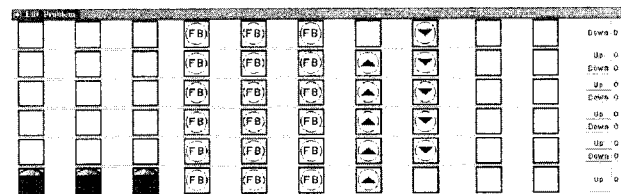


Fig. 11 GUI for the lift problem

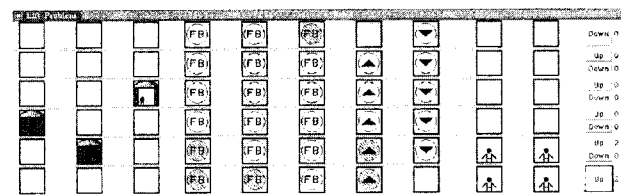


Fig. 12 Snapshot of the animation

V. Related works

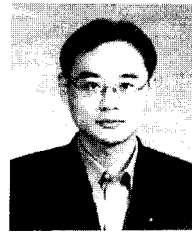
A method for the visual requirement specifications is discussed in [11]. It only works for elements of the system that are easier to visualize, such as the data flow of the system. They are specified by using VRDL. We perform animation of the flow of control represented by the state diagrams. In [12], the behavior of a system is simulated by executing the production rules combined with icon animation. We perform animation by executing the event methods whose body code are generated by our system. In [13], only textual information is displayed and describes the behavior of the system without any animation. In [14], it presented an interface toolset for constructing and using iconic interfaces for interactive workspaces. We animate the behavior of the system by using the icon transformations.

VI. Conclusion

We proposed the iconic animation from the software requirement specifications based on the OMT methodology. The iconic animation is performed by the GUI components and executing event methods which contain icon transformations. Through animation, it becomes easy to understand the software requirement specifications and to examine the correctness of the software requirement specifications at the analysis stage.

REFERENCES

- [1] M. Rumbaugh, J. Blaha and W. Premerlani, *Object-oriented Modeling and Design*. Prentice-Hall, 1991.
- [2] Rational Rose, Rational Software Corporation, <http://www.rational.com/>.
- [3] D. Harel and E. Gery, "Executable object modeling with statecharts," *Proc. 18th International Conf. on Software Engineering*, IEEE, pp. 246-257, 1996.
- [4] Thomas Nocke, Stefan Schlechtweg and Heidrun Schumann, "Icon-based Visualization using Mosaic Metaphors," *Proc. Ninth International Conference on Information Visualization*, pp. 103-109, 2005.
- [5] Joung, S. Ali and J. Tanaka, "Automatic animation from the requirement specification based on object modeling technique," *Proc. International Symposium on Future Software Technology (ISFST-97)*, pp. 133-139, 1997.
- [6] Broy, M., "Requirements and design specification of distributed systems: the lift problem," *Proc. International Conference on Distributed Computing Systems*, pp. 164-173, 1988.
- [7] J. Rumbaugh. "Controlling code," *Journal of Object-oriented Programming*, pp. 25-30, May 1993.
- [8] J. Ali and J. Tanaka, "Generating executable code from the dynamic model of omt with concurrency," *Proc. International Conference on IASTED*, 2(5), 1997.
- [9] M. Hirakawa, M. Tanaka and T. Ichikawa, "An iconic programming system, hi-isual," *IEEE Transactions on Software Engineering*, 16(10), pp. 1178-1184, 1990.
- [10] S. K. Chang, "Visual languages: A tutorial and survey," *IEEE software*, pp. 29-39, January 1987.
- [11] A. Ohnishi, "A visual software requirements specification technique," *Transaction of Information Processing Society of Japan (in Japanese)*, 36(5), pp. 1183-1191, May 1995.
- [12] R. St-Denis, "Specification by example using graphical animation and a production system," *Proc. IEEE HICSS*, 2, pp. 237-246, 1990.
- [13] P. Hsia and A. T. Yang, "Screen-based scenario generator: A tool for scenario-based prototyping," *Proc. IEEE HICSS*, 2, pp. 455-461, 1988.
- [14] Jacob T. Biehl and Brian P. Bailey, "A Toolset for Creating Iconic Interfaces for Interactive Workspaces," *Lecture Notes in Computer Science*, Vol. 3585, 2005.



Suck-Tae Joung

received the B.S. degree in Computer Science from Chonnam National University in 1989 and M.S. and Ph.D. degrees in Computer Engineering of from Tsukuba University, Japan, in 1996 and 2000, respectively. In 2001, he joined the Division of Electrical Electronic and Information Engineering, Wonkwang University, where he is presently professor. His research interest is in the area of Visual System and Information Communication.