1997 한국정보처리학회 논문지 제6권 제6호(993)

객체 지향 설계 명세서에 대한 설계 검증 방법

김 은 미†

요 약

본 논문에서는, 객체 지향 설계 명세서를 대상으로, 안전성 검증 방법을 제안하였다. 먼저 검증의 대상으로 하는 폴트를 명확히 하기 위해, 요구 명세서와 설계 명세서사이에 발생 가능한 불일치를 분석한다. 다음, 설계 명세서에 포함된 폴트를 발견하기 위한 새로운 설계 레뷰 방법을 제안한다. 제안한 검증 방법의 특징은 검증될 모든 요소가 대상 프로덕트의 요구 명세서, 안전성 기준, 그리고 설계 명세서에 기반을 두고 추출될 수 있다는 것이다. 이러한 정보가 표의 형식으로 작성되기 때문에, 검증 단계를 단순화 할 수 있다. 여기에서, 컴포넌트 라이브러리, 안전성 기준, 그리고 Booch의 설계 방법에 의해 작성된 설계 명세서가 주어졌다고 가정한다. 먼저, 설계 레뷰를 하는 검증자가 정확성 검증표와 안전성 검증표를 작성한다. 한편 설계자는 설계 검증표를 작성한다. 이렇게 작성된 3개의 검증표를 이용하여, 설계 단계에 포함되어 있는 폴트를 검출한다. 마지막으로 Case study를 통하여 제안한 검증방법의 유효성을 평가하였다.

A Design Verification Method for Object-oriented Design Specification

Eun-Mi Kim†

ABSTRACT

In this paper, we present a first step for developing a method of verifying both safety and correctness of object-oriented design specification. At first, we analyze the discrepancies, which can occur between requirements specification and design specification, to make clear target faults. Then, we propose a new design verification method which aims at detecting faults in the design specification. The key idea of the proposed framework is that all elements to be verified can be extracted based on the requirements specification, safety standards, and design specification given for the target product. These elements are expressed using three kinds of tables for verification, and thus, the verification steps can be greatly simplified. Here, we assume that component library, standards for safety and design specification obtained from the Booch's object-oriented design method are given. At the beginning, the designers construct a design table based on a design specification, and the verifiers construct a correctness table and a safety table from component library and standards for safety. Then, by comparing the items on three tables, the verifiers verification a given design specification and detect faults in it. Finally, using an example of object-oriented design specification, we show that faults concerning safety or correctness can be detected by the new design verification method.

1. Introduction

Object-oriented software development is attractive to software engineers because its use increases productivity throughout the software life cycle[5]. There exist several object-oriented design methods [1,2,10,11]. Some of the typical object-oriented design methods are due to Booch and Rumbaugh. Espe-

※ 본 연구는 호원대학교 교내학술연구비 지원에 의해 진행되었음.
† 정 회 원 : 호원대학교 컴퓨터공학과 교수
논문접수 : 1998년 9월 14일, 심사완료 : 1999년 2월 26일

cially, Booch method is a broad methodology that addresses most aspects of the object-oriented analysis and design technology.

On the other hand, as software is used in the safety critical systems, such as aircrafts, an atomic power plant and so on, the safety has become the most important quality characteristic. Since the requirements for safety are not explicitly described in requirements specification, it is very difficult to assure the safety in the technical reviews and testing. Generally, two types of analysis - dynamic and static - can be used for verification of safety[6]. In dynamic analysis, the code or model of the code is executed and its performance is evaluated. In static analysis, the code and model are examined without being executed. In some ways, static analysis is more complete than dynamic analysis, since general conclusions can be drawn[8,13].

Formal verification and Software Fault Tree Analysis(SFTA) are used as static analysis[3,8]. Formal verification essentially provides a proof of a consistency between two formal specifications of a system. But there are both practical and theoretical limits and the few formal verifications applied to real programs have required massive effort even for relatively small software[4,8]. In addition, even published proofs of small algorithms have been found to be flawed. In the previous researches[3,8], *Fault Tree Analysis*(called FTA), that is often used to verify the safety of hardware system, has been applied to software. This new analysis method is called SFTA. SFTA traces that behavior into the logic of the code and determines whether a path exists through the code that could cause the hazardous output. However, these methods use the reachability graph to analyze statically whether the behavior graph can hold safety assertions, and the reachability graph brings about problems for state explosion.

Moreover for some critical safety-related systems, correctness proofs are a valuable aid in increasing confidence in the system[12]. Unsafe states may be correct if the specification is flawed by including behavior that happens to be unsafe or does not specify anything about a particular hazardous behavior[8]. Formal proofs of correctness attempt to prove that the program meets its specification. Technical reviews(design review, code review and so on) and testing are well known methods for assuring correctness[9]. But, the correctness of software in an early phase cannot be completely verified by reviews and testing. Thus, it is necessary to otherwise verify both safety and correctness of software in the early phase of the software life cycle.

In this paper, we aim to develop a new method to verify both safety and correctness of object-oriented design specification simultaneously(Concerning the outline, we have already presented it in [5]). At first, this paper proposes a new design verification method to detect faults contained in the design specification by using three kinds of information tables. Here, we assume that component library, standards for safety and design specification obtained from the Booch's object-oriented design method are given, and that designers and verifiers participate to build information tables from the specifications. Based on these tables, the verifiers review a given design specification and detect faults in it. Finally, by applying the proposed framework to a small example of software design specification, we show that faults concerning safety or correctness can be detected in the new design verification.

2. Preliminaries

2.1 Object-oriented design method by Booch

There exist several object-oriented design methods [1,2,10,11]. Some of the typical object-oriented design methods are due to Booch and Rumbaugh. From now on, we will use the term "Booch method" and "OMT(Object Modeling Technique)" to refer to the object-oriented design methods proposed by Booch and Rumbaugh, respectively. Especially, Booch method is a broad methodology that addresses most aspects of the object-oriented analysis and design

technology. Also, Booch and Rumbaugh aim to produce a single, unified method that bring together Booch method and OMT. In addition, it incorporates the lessons learned from real projects as well as from other methodologies. In [1], Booch method adopts a number of elements from OMT such as associations and roles. Similarly, OMT also adopts some elements from Booch method such as categories and object message diagrams. Therefore, if the proposed method can verify the design specification designed by Booch method, we consider the proposed method can be also applied to other methods easily.

In this paper, we aim to verify the object-oriented design specification which results from the Booch method. The Booch method uses the six kinds of diagrams(Class diagram, Object diagram, Module diagram, Process diagram, State transition diagram and Interaction diagram) to describe the strategic and tactical analysis and design decisions that must be made when creating an object-oriented system[1].

2.2 Definition of Correctness and Safety

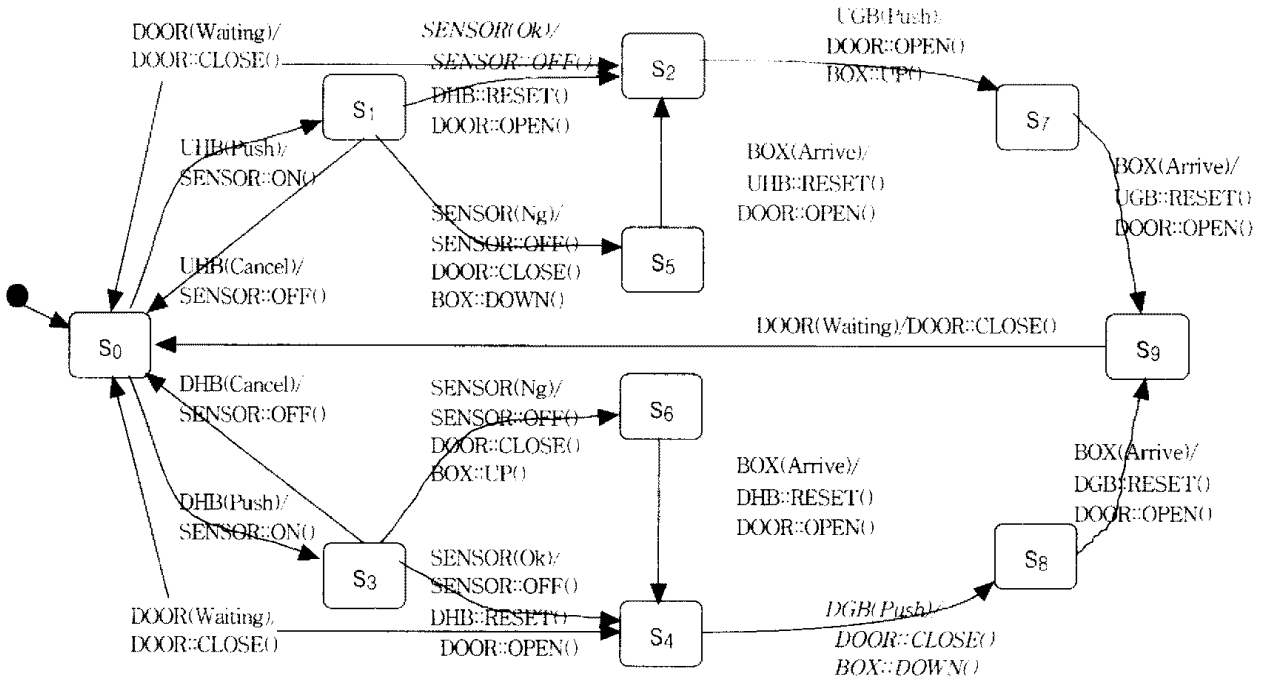
According to IEEE standard, correctness is defined as the degree to which software, documentation, or other items meet specified requirements [14]. Thus, in order to verify the correctness, we check whether design specification meets requirements specification. On the other hands, safety is defined as the degree of freedom from risk in any environment[8]. In order to verify the safety, we check whether design specification has some risk or not. Assume that software is produced correctly according to the requirements specification. So, if the requirements specification was flawed and incomplete, then the software might be not safe[8]. Therefore, it is necessary to verify safety as well as correctness of software.

2.3 Classification of Faults

Here, we classify the discrepancies, which can occur between requirements and design specifications, into eight types of faults to make clear target faults. <Table 1> shows the results of the classification. We consider that requirements specification includes two items : descriptions about functions and safety of software. In Table 1, ○ in column "Function" means a function of software is described correctly. Similarly, × in column "Safety" means a function of software doesn't violate safety, and ○ in column "Design specification" means the design specification is produced correctly according to the requirements specification. For example, if the descriptions of the function and the safety in the requirements specification are correct and the description of the implementation in the design specification is incorrect, then it belongs to the type *B* fault. Also, in case that the description of a function in the requirement specification is correct, if the description of the safety and the implementation in the design specification are incorrect, then it belongs to the type *D* fault. Here, type *A* fault influences neither correctness nor safety of software, therefore we don't discuss type A. On the other hand, type E fault is due to the incorrect description of the requirements specification. However in this case, the design specification meets the corresponding requirements specification. According to the definition for correctness, we can say that the design specification is correctly implemented. Thus, type E fault also influences neither correctness nor safety of the software, therefore we do not discuss type E fault.

〈Table 1〉 Classification of faults

Requirement specification		Design specification	fault type
Function	Safety		
○	○	○	A
		×	B
○	×	○	C
		×	D
×	○	○	E
		×	F
×	×	○	G
		×	H



(Fig. 1) Design specification of elevator control program

Our proposed method consists of two phases : correctness verification and safety verification.

Types *B*, *D*, *F* and *H* faults can be detected in correctness verification phase and/or safety verification phase. Next, types *C* and *G* faults can be detected only in safety verification phase.

3. Proposed Method

In this section, we propose a new design verification method to verify both safety and correctness simultaneously of object-oriented design specifications.

3.1 Assumptions

In our proposed method, we require the following four assumptions :

(1) The object-oriented design specification to be verified is obtained by the Booch method[1].
The Booch method uses the six kinds of diagrams(class diagram, object diagram, module diagram, process diagram, state transition diagram and interaction diagram) to describe the strategic and tactical analysis and design decisions that must be made when creating an object-oriented system[1]. In this paper, we deal with the faults related only to the action and the state transitions in the state transition diagram [1].

(2) Component library is prepared. In the library, many kinds of component diagrams exist. Each component diagram corresponds to the object diagram in the Booch method, and includes the following information : (1) *name* of the component. (2) a set of *internal states*, (3) a set of *operations* (there are two types of operations, *event* and *action*) and (4) *behavior* of the component described using a state transition diagram.

(3) Standards[8] for safety, which reflect lessons learned from previous accidents, are prepared.

(4) Requirements specification is described using natural language and includes the functional requirements which describe the fundamental behavior of the software.

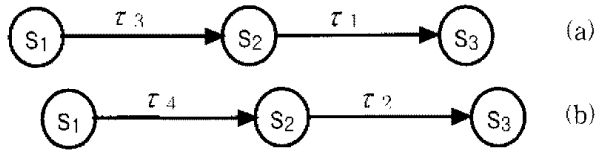
3.2 Design specification

In this Section, we explain the state transition diagram[1] which is considered as the design specification in this paper. The state transition diagram is used to show : (1) state space of a given class, (2) events that cause a transition from one state to another, and (3) actions that result from a state change. If an event can occur at a certain state, then the corresponding actions are executed and the state is changed into the destination of the transition.

Figure 1 shows an example of a state transition diagram. For example, if the event "UHB(Push)" occurs at the state "s_0", then the action "SENSOR:: ON()" is executed and the state is changed into "s_1". Many kinds of techniques have been proposed to verify the safety of software designed by state transition diagrams[3,8].

Most of them have used the reachability analysis to verify it. However, the reachability analysis causes state explosion during construction of the reachability graph. In this paper, to resolve this problem, we consider only events and actions in verifying the correctness and safety. In other words, we do not take the state of the system into account explicitly. From now on, we will use the term "design specification" to refer to the state transition diagram.

Here, we assume that for any event e of a transition τ in the state transition diagram, its associated actions are determined by the event e and an event e' of any previous transition τ' which has occurred just before the transition τ.



(Fig. 2) Restriction on transition

Consider two transitions τ_1 from s_2 to s_3 and τ_2 from s_2' to s_3', as shown in Figure 2 (a) and (b), such that τ_1 and τ_2 have the same event e_2. Assume that the previous two transitions τ_3 from s_1 to s_2 and τ_4 from s_1' to s_2' have the same event e_1. Then the new assumption requires that two transitions τ_1 and τ_2 must have the same action. This restriction may be too strict in large scale software. But, in small scale software such as a control program of an electrical pot or microwave oven[3,6], the action strongly depends on the event regardless of what state it was originally in. So, we don't think this restriction is very strict for a small scale software design.

3.3 Overview

Figure 3 shows the overview of the proposed method. In the proposed method, designers construct a *design verification table*. The design verification table includes information related to correctness and safety of the design specification. On the other hand, verifiers construct a *correctness verification table* and a *safety verification table*. The correctness verification table includes conditions to be proved for assuring the correctness, and the safety verification table includes conditions to be proved for assuring the safety. Finally, by using these tables, verifiers review the design specifications to detect the faults which affect the correctness or the safety.

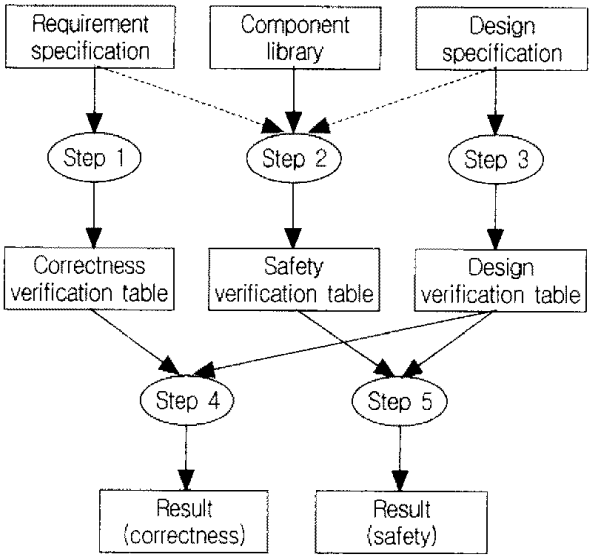
3.4 Verification tables

Next, we describe three kinds of verification tables(CVT, SVT and DVT), used in the proposed method.

(1) Correctness Verification Table(CVT) : Relationships between events and actions, which are extracted from the requirements specification, are described. These give conditions to be proved for assuring the correctness of the deign specification.
(2) Safety Verification Table(SVT) : Relationshipsb among the states of objects, which are extracted from the standards for safety, the experience and

knowledge of verifiers, are described. In other word, SVT includes safety conditions about interaction among objects. These give conditions to be proved for assuring the safety of the design specification.

(3) Design Verification Table(DVT) : Relationships between events and actions, which are extracted from the design specification, are described.



(Fig. 3) Outline of the proposed method

3.5 Verification procedure

The verification procedure consists of the following steps :

Step 1. Construction of CVT and SVT

(1-1) Construction of CVT

(1-1-1) Extract all objects from the requirements specification. Then, take out all component diagrams which correspond to the extracted objects from the component library.

(1-1-2) Extract all events and actions from the component diagrams and put them into the two dimensional array CVT.

(1-1-3) For each event e in CVT, put a ○ on each of event ρ that must have occurred just before the current event e based on the requirements specification and component diagrams extracted in (1-1-2).

(1-1-4) For each event in CVT, put a ○ on each of actions that can occur for the event.

(1-2) Construction of SVT

(1-2-1) Extract all objects from the requirements specification.

(1-2-2) For each object, extract all states from the component diagram and put them into the two dimensional array SVT.

(1-2-3) For each state s of an object in a column of SVT, put an × on each of states t of objects in a row of SVT such that both state s and state t cannot occur simultaneously based on the standards for safety and the experience and knowledge of the verifiers.

Step 2. Construction of DVT

(2-1) Extract all actions and events from the design specification, and put them into the two dimensional array DVT.

(2-2) For each event e in DVT, put a ○ on each of events ρ that have occurred just before the current event e based on the design specification.

(2-3) For each event in DVT, put a ○ on each of actions that occur for the event.

Step 3. Verification of correctness and safety

(3-1) Verification of correctness :

Check whether the design specification meets the requirements specification. That is, check if the DVT includes all ○'s in the CVT at the same positions as in the CVT.

(3-2) Verification of safety :

Check whether the design specification has some risk. That is, determine whether the DVT violates restrictions in the SVT.

3.6 Comparison with respect to conventional methods

Here, we compare the proposed method with conventional review methods and the safety verification method proposed by Fukaya et al.

(1) Comparison with conventional design review methods : The conventional design review mainly aims to prove the correctness of the design specification obtained by the structured design method. For detecting design faults effectively, some kind of check-list is usually used in the design review. But, in order to extend applicabilities, the items in the check list are generally described abstractly. So, for checking faults which depend on the target design specification, a check-list is neither sufficient nor useful. Moreover, since conditions for safety are not explicitly described in the requirements specification, it is very difficult to prove the safety conditions.

On the other hand, the correctness and safety tables in the proposed method are substituted for check-lists, and the information in these tables is much more dependable. Moreover, since conditions for safety are extracted from the knowledge of verifiers and standards for safety, which reflect lessons learned from similar projects, we can expect that the proposed method makes the design review more effective and efficient than conventional methods.

(2) Comparison with safety verification by Fukaya et al.[3] : Fukaya et al. have proposed a method to verify the safety of the design specification. In order to avoid software design faults, they derive safety assertions using Fault Tree Analysis, compute a reachability graph of the specification, and analyze statically whether this graph satisfies safety assertions. They assume that the requirements specification is correct. However, since the majority of safety problems arise from software requirements errors[8], their assumption is strict. That is, though the software correctly implements the requirements, the requirements sometimes specify behavior that is not safe from a system perspective[8].

In the proposed method, we don't assume that the requirements specification is correct. If the design specification includes the faults against safety due to the incorrect requirements specification, we can detect them in the safety verification step.

4. Case Study

In this section, we apply the proposed method to the design specification which includes two faults. We consider the development of the control program for a two story elevator system[7]. Assume that the requirements specification shown in Figure 4 and the design specification shown in Figure 1 are given.

First, objects are extracted from the given requirements specification. Here, the objects BOX, DOOR, BUTTON, and SENSOR are extracted. Then take out all component diagrams which correspond to the extracted objects from the component library. Figure 5 shows the component diagrams which correspond to the components DOOR and BOX respectively.

4.1 Application of Step 1 and Step 2

At Step 1, verifiers construct the CVT and SVT, shown in Tables 2 and 3, based on the component diagram shown in Figure 5, and the descriptions about the system behavior in the requirements specification(Step (1-1)). For example, consider the statement "If the sensor detects that the elevator is on the first floor, then the sensor is turned off, the button is turned off, and the door of the elevator opens" in the requirements specification. Judging from the previous statement in the requirements specification and in reviewing the component diagrams, in order to confirm whether the box is on the floor or not, the box should be called by pushing the button. Since the button is pushed on the first floor, the button means Up-Hall-Button. Therefore, ○'s are placed at the intersection of the current event (SENSOR, Ok) and the previous event (UHB, Push), and at the intersections of the current event (SENSOR, Ok) and actions [UHB, RESET], [DOOR, OPEN], and [SENSOR, OFF], as shown in Table 2. This means that if two events (UHB, Push) and

(SENSOR, Ok) occur successively, then the actions [UHB, RESET], [DOOR, OPEN], and [SENSOR, OFF] are performed.

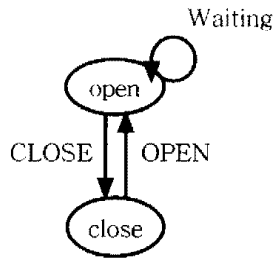
Next, the unusual situations among the components are analyzed by referring to the component diagrams. As a result, the SVT shown in Table 3 is constructed(Step (1-2)). For example, from the knowledge and experience of the verifiers and the standard about safety, an × is placed at the intersection of state 'down' of BOX in the column and state 'open' of DOOR in the row. This implies that 'down' of BOX and 'open' of DOOR are not permitted simultaneously.

Design of the program that controls an elevator in a 2 story building.

There are a button for requesting the elevator on each of the two floors, and a sensor for detecting if the elevator is on the floor. When the button is pushed on the first floor, the button is turned on and the sensor is activated. The sensor checks if the elevator is already on the first floor.(If the button on the first floor is pushed again, then the request for the elevator is canceled and the sensor is turned off.) If the sensor detects that the elevator is on the first floor, then the sensor is turned off, the button is turned off, and the door of the elevator opens.(x)
If the sensor detects that the elevator is not on the first floor, then the elevator moves to the first floor automatically. Once the elevator arrives at the first floor, the sensor is turned off, the button is turned off, and the door opens.....
When the button is pushed on the second floor, the button is turned on and the sensor is activated..... Once the elevator arrives at the second floor, the sensor is turned off, the button is turned off, and the door opens. The door closes automatically when a fixed amount of time elapses. If the elevator is on the second floor and the DOWN button is pushed, then the door opens and the elevator moves to the first floor.(y) Once the elevator arrives at the first floor, the DOWN button is turned off and the door opens. The door closes automatically when a fixed amount of time elapses.

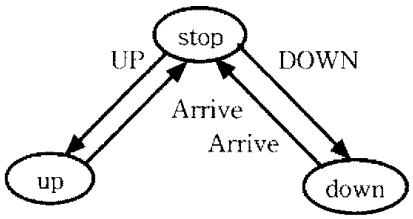
(Fig. 4) Requirements specification for elevator control system

Component DOOR
state { open, close }
method { A:OPEN, A:CLOSE, E:Waiting }
state transition {



}

Component BOX
state { stop, up, down }
method { A:UP, A:DOWN, E:Arrive
state transition {



}

(Fig. 5) Component diagrams

Based on the design specification in Figure 1, designers construct the DVT shown in Table 4 (Step 2). For example, consider the transition from the state s_4 to the state s_8 in Figure 1. That is, if event DGB (Push) occurs at state s_4, then the actions DOOR::OPEN() and BOX::DOWN() are executed and the state is changed into s_8. In this case, there are two possible previous events. One is the BOX(Arrive) from state s_6 to state s_4, and the other is SENSOR (Ok) from state s_7 to state s_4. Therefore, ○'s are placed at the intersection of the current event (DGB, Push) and the previous event (SENSOR, Ok), and at the intersections of the current event (DGB, Push) and the actions [BOX, DOWN] and [DOOR, OPEN]. Similarly, ○'s are placed at the intersection of the current event (DGB, Push) and the previous event (BOX, Arrive), and at the intersections of the current event (DGB, Push) and the actions [BOX, DOWN] and [DOOR, OPEN].

〈Table 2〉 Correctness Verification Table(CVT) for elevator control system

Current event	Previous event: (UHB, Push)	(UHB, Cancel)	(DHB, Push)	(DHB, Cancel)	...	(SENSOR, Ok)	(SENSOR, Ng)	(BOX, Arrive)	Action: [UHB, RESET]	[DHB, RESET]	...	[BOX, UP]	[BOX, DOWN]	[DOOR, OPEN]	...	[SENSOR, OFF]
(UHB,Push)		○														
				○												○
(UHB,Cancel)	○															
(DHB,Push)				○												
		○														
(DHB,Cancel)			○													○
(UGB,Push)						○						○				
								○				○				
(DGB,Push)						○							○	○		
								○					○	○		
(DOOR,Waiting)						○				(x)						
								○								
(SENSOR,Ok)	○								○					○		○
			○							○				○		○
(SENSOR,Ng)	○												○			○
			○									○				○
(BOX,Arrive)							○		○					○		
							○			○				○		
														○		
														○		

4.2 Application of Step 3

1The requirements specification in Figure 4 includes two faults. In the requirements specification, (x) represents a statement where the functions are described correctly, and (y) represents a statement where the functions are described incorrectly. But, neither (x) nor (y) say anything about safety. The two faults in this example are explained in detail as follows :

(1) Correctness fault (underlined by (x)) : Consider a transition from state "s_1" to state "s_1" in Figure 1 such that the event and actions on the transition are defined as "SENSOR(Ok)/SENSOR::OFF(), DHB::RESET(), DOOR::OPEN()". However, the require-

〈Table 3〉 Safety Verification Table(SVT) for elevator control system

Object(row)																Object column	
UHB		DHB		UGB		DGB		BOX			DOOR		SENSOR				
on	off	on	off	on	off	on	off	stop	down	up	open	close	on	off			
															on	UHB	
															off		
															on	DHB	
															off		
															on	UGB	
															off		
															on	DGB	
															off		
												(y)			stop	BOX	
											×				down		
											×				up		
															open	DOOR	
															close		
															on	SENSOR	
															off		

ments specification shown in Figure 4 says "If the sensor detects that the elevator is on the first floor, then the sensor is turned off, the button is turned off, and the door of the elevator opens". From the requirements specification, we can see that the button is pushed before the elevator box is on the first floor. Since the button is pushed on the first floor, the button means Up-Hall-Button. Thus, the action DHB::RESET() on the transition from s_1 to s_2 is a correctness fault, that is, an incorrect implementation for the statement (x) in Figure 4, and belongs to the type D fault.

(2) Safety fault(underlined by (y)) : Consider a transition from state "s_4" to state "s_8" in Figure 1 such that the event and action on the transition are DGB (Push)/DOOR::OPEN(), BOX::DOWN(). Clearly, this is a correct implementation for the statement (y) in Figure 4. However, when an elevator is moving, its door must be closed. If the box moves continuously even if the door is opening, then it can cause trouble, and it may cause damage to the user. This is a safety fault, and belongs to the type G fault.

At Step(3-1), according to the CVT shown in Table 2, for the current event (SENSOR, Ok) with the previous event (UHB, Push), the intersection of (SENSOR, Ok) and [UHB, RESET] is marked with a ○. However, in the DVT shown in Table 4, for the current event (SENSOR, Ok) with the previous event (UHB, Push), the intersection of (SENSOR, Ok) and [UHB, RESET] is not marked with a ○. Thus, the correctness fault is detected.

Next, at Step (3-2), according to the SVT shown in Table 3, the intersection of the 'down' state of BOX in the column and the 'open' state of DOOR in the row is marked with an ×. This means that while BOX is in the 'down' state, DOOR can not change to the 'open' state. However, in the DVT shown in Table 4, the intersections of (DGB, Push) and [BOX, DOWN], (DGB, Push) and [DOOR, OPEN] are marked with ○s. This implies that 'down' of BOX and 'open' of DOOR are permitted simultaneously. Thus, the safety fault is detected.

5. Limitations of the proposed method

In the CVT and DVT shown in Tables 2 and 4, there are current events whose previous events are the same, but the actions are different. In this case, if the design specification is implemented by exchanging the actions to be performed, then the proposed method cannot detect the fault. For example, consider the statement "If the sensor detects that the elevator is not on the first floor, then the sensor is turned off, the button is turned off, and the door of the elevator opens" in the requirements specification. Then in the CVT, ○'s are placed at the intersection of the current event (BOX, Arrive) and the previous event (SENSOR, Ng), and at the intersection of the current event (BOX, Arrive) and the actions [UHB, RESET] and [DOOR, OPEN].

On the other hand, consider the other statement "If the sensor detects that the elevator is not on the second floor, then the sensor is turned off, the button is turned off and the door of the elevator opens" in the requirements specification. Then in the CVT, ○'s are placed at the intersection of the current event (BOX, Arrive) and the previous event (SENSOR, Ng), and at the intersections of the current event (BOX, Arrive) and the actions [DHB, RESET] and [DOOR, OPEN].

For example, consider the case in which the design specification in Figure 1 is implemented incorrectly such that the actions on the two transitions from s_5 to s_2 and from s_6 to s_4 are exchanged. Then, in the DVT, ○'s are placed at the intersection of the current event (BOX, Arrive) and the previous event (SENSOR, Ng), and at the intersections of the current event (BOX, Arrive) and the actions [UHB, RESET] and [DOOR, OPEN]. Also, ○'s are placed at the intersection of the current event (BOX, Arrive) and the previous event (SENSOR, Ng), and at the intersections of the current event (BOX, Arrive) and the actions [DHB, RESET] and [DOOR, OPEN]. As a result, all of the ○'s for this case in CVT are included at the same

<Table 4> Design Verification Table(DVT) for elevator control system

Current event	Previous event: (UHB, Push)	(UHB, Cancel)	(DHB, Push)	(DHB, Cancel)	...	(SENSOR, Ok)	(SENSOR, Ng)	(BOX, Arrive)	Action: [UHB, RESET]	[DHB, RESET]	...	[BOX, UP]	[BOX, DOWN]	[DOOR, OPEN]	...	[SENSOR, OFF]
(UHB, Push)																
(UHB,Cancel)																
(DHB,Push)																
(DHB,Cancel)																
(UGB,Push)																
(DGB,Push)																
(DOOR,Waiting)																
(SENSOR,Ok)																
(SENSOR,Ng)																
(BOX,Arrive)																

position in the DVT. Thus, the fault is not detected using the proposed method. However, verifiers can identify the cases or the parts of the CVT and DVT in which both the current event and the previous events on different transitions are the same. Therefore verifiers can be instructed to examine the parts in detail and detect the faults.

6. Conclusion

In this paper, we classify the discrepancies, which can occur between requirements specification and design specification, into eight types of faults and make clear target faults. Then, we propose a new design verification method to detect faults that are contained in the design specification using three kinds of information tables(CVT, SVT and DVT).

In this paper, CVT and DVT don't include the state dependencies. Because, in small scale software such as a control program of an electrical pot or microwave oven[3], the state is strongly depend on the event regardless of what state it was originally in. Currently, as the future research work, we are extending the proposed method to relax this assumption. We are also planning to extend the proposed method to deal with six diagrams (state transition diagram, class diagram, object diagram, interaction diagram, module diagram, and process diagram) by using the information of each diagram and evaluate the proposed method using practical software development projects.

Reference

[1] G. Booch, "Object Oriented Analysis and Design with Applications," The Benjamin/Cummings, 1994.

[2] P. Coad and E. Yourdon, "Object Oriented Analysis, 2nd ed.," Yourdon Press, 1991.

[3] T. Fukaya, M. Hirayama and Y. Mihara, "Software specification verification using FTA," Proceedings of *the 24th FTCS*, pp.131-133, 1994.

[4] S. Gerhart, D. Craigen and T. Ralston, "Experience with formal methods in critical systems," IEEE Softw. 11, 1, pp.21-28, 1994.

[5] J. Jeftcoate, K. Hales and V. Downes, "Object Oriented System : the Commercial Benefits," Ovum Ltd, 1989.

[6] E. M. Kim, S. Kusumoto and T. Kikuno, "An approach to safety and correctness verification of software design specification," Proceedings of *the 6th ISSRE*, pp.78-83, 1995.

[7] E. M. Kim, S. Kusumoto and T. Kikuno, "A new verification framework of object-oriented design specification for small scale software," IEICE Transactions on Information and Systems, E80-D, 1, pp. 51-56, 1997.

[8] N. G. Leveson, "Safeware : System Safety and Computers," Addison Wesley, 1995.

[9] G. J. Myers, "The Art of Software Testing," Wiley-Interscience, 1979.

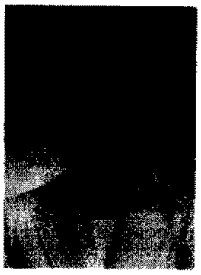
[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, "Object Oriented Modeling and Design," Prentice Hall, 1991.

[11] S. Schlaer and S. Mellor, "Object Oriented System Analysis," Prentice Hall, 1988.

[12] I. Sommervill, 'Software engineering,' Addison-Wesley, 1992.

[13] L. G. Williams, "Assessment of safety-critical specifications," IEEE Softw. 11, 1, pp.51-60, 1994.

[14] "IEEE Standard Glossary of Software Engineering Terminology," IEEE, ANSI/IEEE Std 610.12-1990, 1990.



김 은 미

e-mail : ekim@sunny.howon.ac.kr

1991년 2월 전북대학교 전산통계학과(이학사)

1993년 8월 전북대학교 대학원 전산통계학과(이학석사)

1997년 3월 일본 오사카대학 정보공학과(공학박사)

1997년 3월~8월 한국전자통신연구원에서 Post-Doc.

1997년 9월~현재 호원대학교(구 전북산업대학교) 컴퓨터공학과 전임강사

관심분야 : 객체 지향 소프트웨어 척도 및 검증 방법, 시큐리티 등