Vol. 39 No. 6 Transactions of Information Processing Society of Japan June 1998

Regular Paper

The SOFL Approach: An Improved Principle for Requirements Analysis*

SHAOYING LIU,† A JEFF. OFFUTT,†† MITSURU OHBA† and KEIJIRO ARAKI†††

In this paper we point out three major deficiencies of data flow diagrams (DFDs) for requirements analysis. One is the impracticability of the rule for decomposing processes, another is the inconvenience of drawing data flows for complex DFDs, and the third is the lack of precision in process specifications. We present an improved approach using SOFL (Structured-Object-oriented-Formal Language) to show how these three deficiencies can be addressed. This approach can be applied to make the use of DFDs more practical, scalable, and more accessible to industrial users.

1. Introduction

Data flow diagrams (DFDs) have been widely used in industry as system modeling tools for requirements analysis. They are considered to be easy to comprehend and effective for modeling many applications. However, we have found problems in the following three major aspects of the use of DFDs: *application of decomposition rules*, *drawing of data flows*, and *specification of processes*.

To cope with the complexity of systems and to allow many people to work on one large task, data flow diagrams for a large and complex system are usually drawn in a hierarchical fashion by decomposing high-level processes. Such decompositions are required to conform to a certain rule that *the dataflows coming into and going out of a process at one level must correspond to the dataflows coming into and going out of an entire DFD at the next lower level which describes that process*[1]. Unfortunately, the word *correspond* is not defined precisely for this rule, and can be interpreted in more than one way. However, two examples given by Yourdon on pages 170 and 171[1] show that the meaning of *correspond* is to keep the input data flows (output data flows) of the high-level process the same as the input data flows (output data flows) of the entire DFD at the immediately lower level. Not only does Yourdon advocate this rule, but it has been a well known principle since DFDs were proposed[2] and many researchers have treated this rule as a *structural consistency* of DFDs and worked out algorithms to check it[3),4)].

However, through our experience in the FM-ISEE project**, we have found that this rule is not effective for requirements analysis. More precisely, if we conform to this rule, requirements analysis cannot be conducted easily and clearly. Requirements analysis with DFDs is a complex process and has to be completed gradually through decomposition of processes. It is usually impossible to list all the necessary input data flows and output data flows on a particular abstract level when requirements are derived. Each decomposition may need some new input or output data flows that reflect further user requirements for the functionality of the high-level process. It may sometimes also not be desirable to draw all the input data flows at high-level data flow diagrams because the high-level data flow diagrams need to show clearly the primary idea of the requirements in order to allow a user-friendly validation of the requirements specification against the client. For these

† Hiroshima City University
†† George Mason University
††† Kyushu University

* This work is supported in part by the Ministry of Education of Japan under a Joint Research Grant-in-Aid for International Scientific Research FM-ISEE (08044167) and by Hiroshima City University under a Hiroshima City University Grant for Special Academic Research (International Studies) SCS-FM (A440).

** FM-ISEE stands for Formal Methods and Intelligent Software Engineering Environments. It is an international collaborative project conducted by Hiroshima City University and Kyushu University of Japan, George Mason University of the USA, The Queen's University of Belfast of the UK, and Queensland University of Technology and Monash University of Australia.

reasons, we take a more relaxed approach in SOFL[5] to allow more input and output data flows in the decomposed DFD so that the system can be modeled gradually from an incomplete and abstract level to a complete and detailed level.

Traditional DFDs impose the restriction that all data flows to be processed or produced must be drawn in the DFDs, but fail to provide a mechanism to avoid cross-drawing of data flows and to express data flows from one page to another. This makes the drawing of data flows for complex DFDs difficult or unclear. To overcome these deficiencies, we designed an additional mechanism in SOFL to allow the use of *broken data flows*, thus improving the readability of DFDs without weakening the expressive power of the DFDs.

Traditionally, informal languages (e.g., English and structured English) and semi-formal languages (e.g., flowcharts, PAD, and structure diagrams) are used to describe process specifications for DFDs. Informal specifications are usually imprecise and ambiguous in specifying process functionality. They are also more algorithmic and usually convey some detailed implementation strategy, which should be avoided at the requirements analysis stage. In contrast, formal notations (e.g., Z and VDM-SL) have obvious advantages in this field[6),7),9)]. However, many applications in industry have shown that formal methods are difficult to use and resource-intensive.

One solution to this problem that has been considered is to integrate DFDs and formal notations, and specific integration methods have been explored. Partly to solve this same problem, we have designed SOFL for system development[5),8)]. SOFL has two parts, one for specification and one for implementation. The specification language is an integration of DFDs, Petri Nets, and VDM-SL; the implementation language is an integration of VDM-SL and high-level languages (C++ and Pascal) to allow object-oriented implementations. The specification language allows user requirements to be specified precisely in a hierarchical fashion and the implementation language allows programs to be written on an abstract level. In this paper we present a way of using SOFL for requirements analysis.

The remainder of this paper is organized as follows. Section 2 uses examples to show why the existing decomposition rule is impractical. Section 3 describes an improved decomposition rule and a requirements analysis process based on it. Section 4 explains how SOFL realizes this rule and conforms to this process. Section 5 describes how to use broken data flows to improve the readability of DFDs. Finally, Section 6 presents our conclusions and outlines our plans for future research.

2. Problem with the Decomposition Rule

The rationale for the existing decomposition rule seems to be that, since a high-level process should capture all the necessary input and output data flows and the decomposed DFD should describe the high-level process more precisely, it should keep their input and output data flows consistent. However, this motivation ignores the fact that the high-level process cannot usually completely capture all the necessary input and output data flows because of the complexity of software systems. In fact, the decomposition of a high-level process consists in completing the functional definition of the high-level process, and during this process more user requirements may need to be captured by using more input or output data flows. For example, **Fig. 1** shows a context diagram in which the **HCU Management System** process is an abstraction of the overall system☆. At this early stage the system analyst may not completely know what inputs the system needs to process and what outputs it needs to produce, and therefore the existing decomposition rule is too restrictive.

To cope with this problem and maintain the existing rule, the analyst may add newly occurring data flows in the lower-level DFD to its high-level process once they are created. However, this may result in the high-level process having too many input and output data flows to draw clearly, adding to confusion and encouraging mistakes.

3. Improved Decomposition Rule and Process

3.1 Decomposition Rule

System analysis with DFDs is commonly a successive decomposition process, and each decomposition may realize more functions than its

☆ This system is chosen only as an example to explain the idea of the proposed principle for requirements analysis, it does not necessarily reflect the real management system of Hiroshima City University.

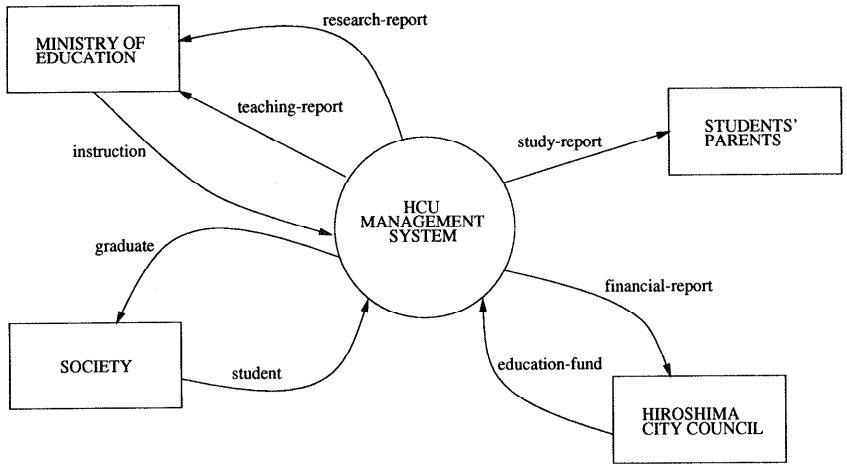

Fig. 1 Context diagram of the HCU Management System.

high-level process. It is very unnatural and difficult to make a high-level process capture all the necessary input and output data flows. Instead, it is natural and easy to let high-level processes capture as many input and output data flows as possible for its potential functionality, and to gradually add more input or output data flows to its decomposed DFD. Of course, decomposed DFDs cannot have fewer input or output data flows than their higher level processes because, otherwise, a decomposed DFD would not realize some behaviors required by its high-level process.

Definition 3.1: Let P be a process and D be a DFD. We then define the following terms:

- $Input(P)$ = the set of all the input data flows of P.
- $Output(P)$ = the set of all the output data flows of P.
- $Input(D)$ = the set of all the data flows coming into D.
- $Output(D)$ = the set of all the data flows going out of D.

Definition 3.2: If P is decomposed into the DFD D and the following two conditions are satisfied,

- $Input(P) \subseteq Input(D)$
- $Output(P) \subseteq Output(D)$

then we say that P is *structurally consistent* with D.

To check the structural consistency for a given process and its decomposed DFD, we only need to list all the input and output data flows of the process and the DFD, and to compare whether the input and output data flows of the process are the subsets of those of the decomposed DFD respectively. However, if the input or output data flows of a process can be mutually exclusive (i.e., if they cannot be available at the same time), then we need to check it algorithmically; the details of this issue are discussed in Section 4. The most important impact on system development of this change in the decomposition rule is that the new rule indicates an *evolutionary decomposition* methodology, in which more functions may be added to the defined process.

Consider building the HCU Management System as an example. Figure 1 shows the context diagram in which the highest-level process, HCU MANAGEMENT SYSTEM, is used to abstract the functionality of the entire system. The diagram includes four terminators and shows their relations with the system. At the current level we can only propose data that are based on current user requirements. When the process HCU MANAGEMENT SYSTEM is decomposed into the immediately lower-level process in **Fig. 2**, we find that in addition to the input and output data flows of the process HCU MANAGEMENT SYSTEM, one more output data flow, failed-student, is necessary as an output data flow of the entire DFD. Thus this decomposition not only decomposes the functionality of HCU MANAGEMENT SYSTEM, but also allow it to evolve so that it can possess more functions.

3.2 Impact on the Semantics of DFDs

The change of the rule for checking the *structural consistency* of DFDs has an important impact on the semantics of classical DFDs such as DeMarco and Yourdon DFDs[1),2)]. Classical

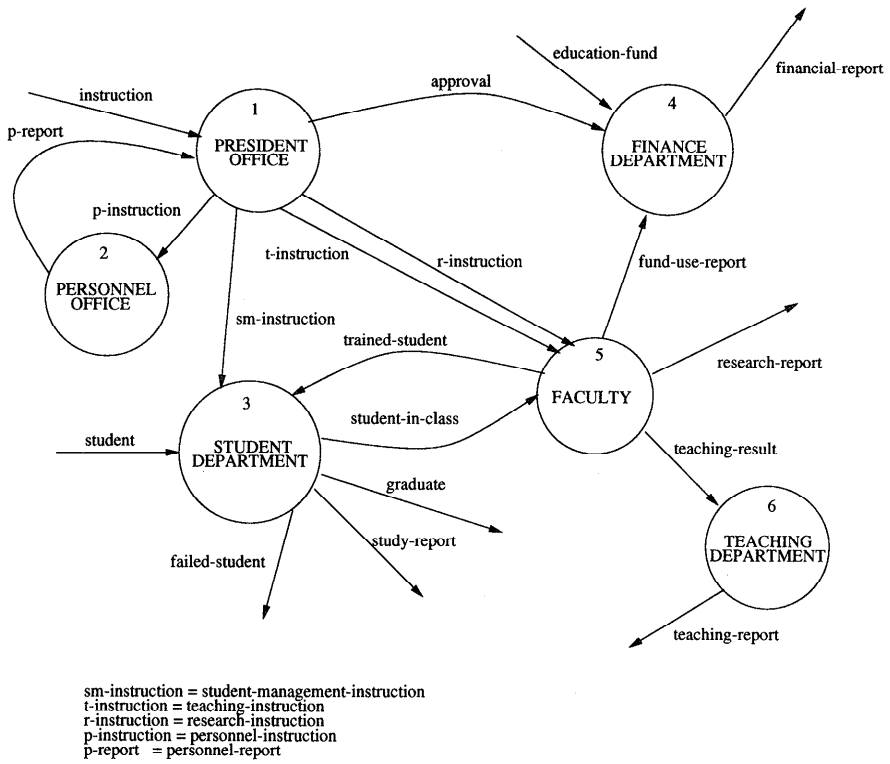

Fig. 2 Decomposition of HCU MANAGEMENT SYSTEM in Fig.1.

DFDs are used to describe a static network of data flows among processes and the decomposition of each process spells out in detail how its input data flows are transformed into its output data flows. In view of this consideration the existing structural consistency rule is reasonable, but it has disadvantages, as we have explained previously. Our newly proposed rule for structural consistency is actually based on a change in the semantics of classical DFDs.

DFDs are used to describe dynamic interactions among processes with data flows. Input data flows of processes have the power to *fire* (or perform) processes once they are available. As a result of a firing, processes generate output data flows. A decomposition of a process is a detailed definition of the process, in which the additional input or output data flows are treated as local data flows of the upper-level process. These additional data flows have one open end, and their source or destination is not given. An advantage of this feature is that a developer can concentrate on the most important and interesting issues and leave the job of determining the exact source or destination to design or implementation (e.g., by calling other functions or reading/writing from or to the display). This provides some additional flexibility for design or implementation.

In fact, this semantics is similar to that of the Pascal language. Consider as an example decomposing process A in the upper-level DFD in **Fig. 3** (1) into the lower level DFD in Fig. 3 (2) as an example. This represents the same semantics as the Pascal procedures given below.

```
procedure G(var x,y,z: Type1)
begin ...
A(x,y);
B(y, z);
...
end;
procedure A(var x,y: Type1)
var
a,b,d,c: Type2;
begin
...
A1(x,a);
Obtain(b);
A2(a,b,d,c);
A3(c, y);
...
end;
```

The definition of procedure G corresponds to the DFD in Fig. 3 (1) and the procedure calls

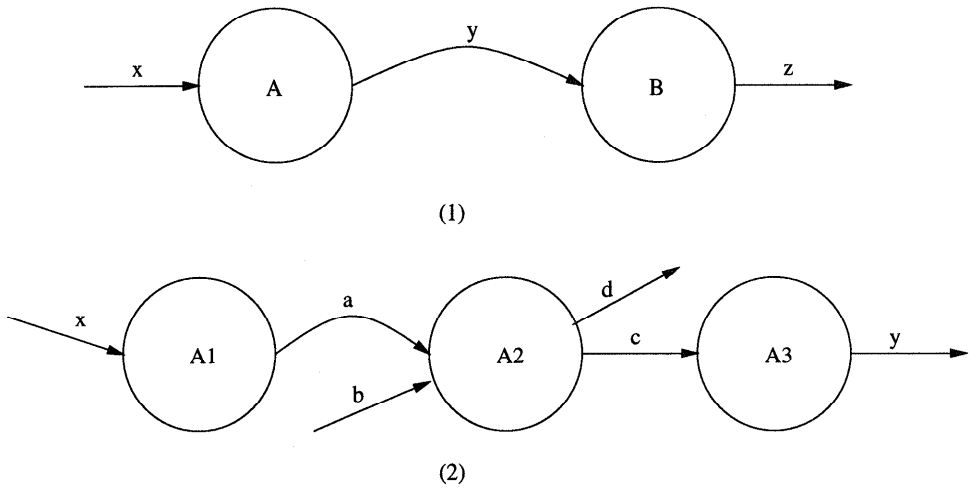

Fig. 3 Example of process decomposition.

A(x,y) and B(y, z) correspond to the occurrence of processes A and B, respectively, in this DFD. The definition of procedure A corresponds to the DFD in Fig. 3 (2), in which procedure calls A1(x,a), A2(a,b,d,c), and A3(c, y) correspond to the occurrences of processes A1, A2, and A3, respectivcly, in Fig. 3 (2). Since the DFD in Fig. 3 (2) is a decomposition of process A, the additional input data flow b and output data flow d are allowed. They are treated as local (or internal) variables in the definition of procedure A, and can be implemented in several different ways. For example, Obtain(b) can be implemented in several different ways, according to the designer. One way is to use the read(b) statement if b is a basic variable such as an integer or a real number. Another way is to call a procedure or function to obtain b if b is a compound variable (e.g., array, linked list, or file). Since process A (or procedure A) is responsible for obtaining b, b is allowed to occur in the decomposition of process A rather than as an input data flow to A in the DFD of Fig. 3 (1). The same principle is applicable to the additional output data flow d in the DFD of Fig. 3 (2), with the difference that the value of d is output to the standard output device.

In summary, the occurrence of processes in upper level DFDs is treated as a use of the processes, and their decompositions are treated as their definitions. This changed semantics of DFDs sets up the foundation for SOFL semantics.

3.3 Analysis Process

On the basis of the evolutionary decomposition methodology, we propose a *successive requirements analysis process* using DFDs, as shown in **Fig. 4**. The essence of this process is that *requirements analysis is a successive process in terms of evolving an initial specification modeled as a hierarchical DFD into a satisfactory specification (though not necessarily a complete one) step by step.*

The first activity for requirements analysis is to capture user requirements as accurately as possible through communication with the user. Such requirements are usually initially written in natural languages together with special terminology used informally in the application domain. After the developer has reached a general understanding, the requirements are then modeled as a hierarchical DFD at an abstract level and a sufficient validation of this DFD against the user requirements is necessary. The analysis then proceeds through successive evolutionary decompositions of this hierarchical DFD until a sufficiently concrete hierarchical DFD is obtained (i.e., each lowest-level process of the hierarchical DFD is simple enough and well specified within a certain formality). Each level of a hierarchical DFD may consist of many inter-related components, and each decomposition transforms abstract requirements into concrete requirements. During a decomposition, the abstract level hierarchical DFD is considered to provide functional constraints for constructing the concrete hierarchical DFD. In principle, the decomposition decisions about adding new functional behaviors in the concrete hierarchical DFD must be made by the developer in consultation with the user, because the system is developed for the user (who may be

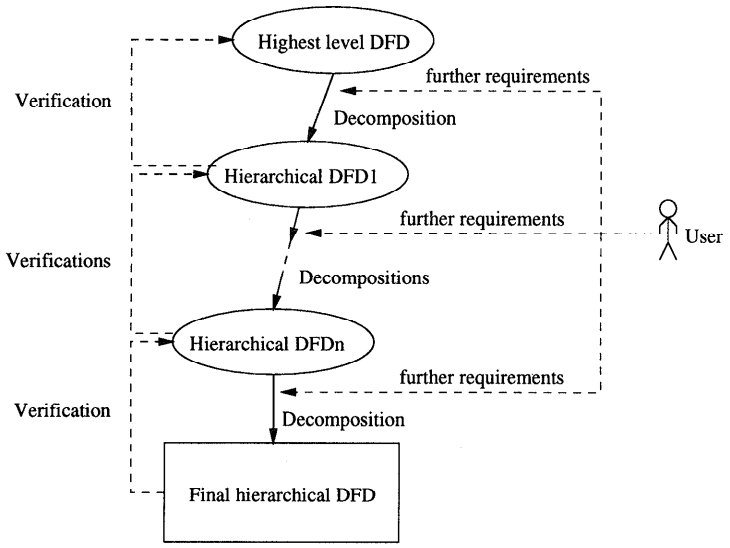

Fig. 4 Successive requirements analysis process.

the same person as the developer). However, when system development reaches a detailed level, it is usually difficult to involve the user. In that case, the person in charge of the abstract hierarchical DFD should play the role of the user, to help make decomposition decisions. In order to ensure the correctness of the concrete hierarchical DFD with respect to the abstract hierarchical DFD, each decomposition must be justified through consistency checking.

To provide technical support for this analysis process, we designed SOFL by integrating DFDs, Petri Nets, and VDM-SL. DFDs are used to provide a friendly mechanism for describing connections among processes in terms of data flows; the concept of *firing* used in Petri Nets provides a precise operational semantics for the DFD; VDM-SL-like notation is employed to specify precise functionality of processes in *specification modules* (**s-module**) corresponding to DFDs, as illustrated in **Fig. 5**.

In fact, SOFL provides an integrated approach for using structured methodology, formal methods, and object-oriented methodology. Since SOFL and its methodology are not the focus of this paper, we discuss only how SOFL is used for requirements analysis. The details of the SOFL language and methodology are described elsewhere [10].

4. SOFL Decomposition Rules

In SOFL we use Condition Data Flow Diagrams (CDFD), which are DFDs that are formalized in the manner as described previously. Processes in CDFD are called *condition processes*, because they are specified with pre- and post-conditions. A condition process is represented graphically as a rectangle with five parts: *name*, *inputs*, *outputs*, *pre-condition*, and *post-condition*. These five parts are located at the center, left, right, top, and bottom of the rectangle, as indicated by examples given later (e.g., **Fig. 6**). The graphical representation of a condition process also shows how to express the input and output data flows graphically. All pairs of data flows (variables) connected with $\odot$ flow into (or leave from) one small rectangular box; small boxes are separated by a short horizontal line, which denotes the operator $\oplus$. The expression a $\odot$ b means that both a and b are required to fire a process, while a $\oplus$ b means that either a or b, but not both, is required to be consumed by the condition process (if they are input data flows) or to be produced by the condition process (if they are output data flows). Each high-level condition process is decomposed into a CDFD under its functional constraints expressed by the pre- and post-conditions. the decomposed CDFD spells out the details of how the functionality of the high-level condition process is realized by lower-level condition processes. In addition to its functional specification in the form of pre- and post-conditions, each lowest-level condition process is implemented in an object-oriented manner, using the executable part of SOFL.

Considering the **HCU Management System** given above as an example, we describe

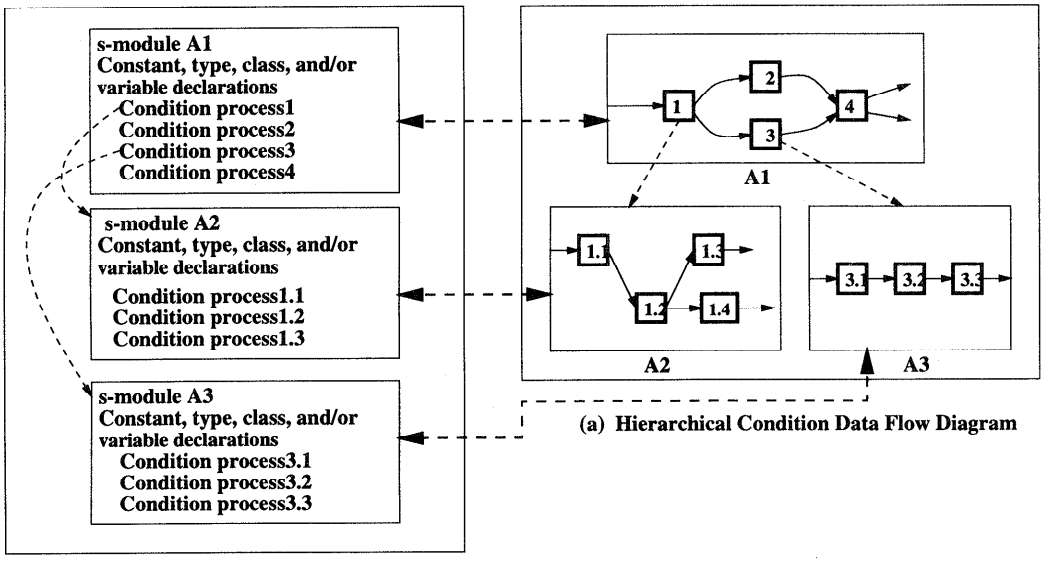

Fig. 5 The structure of a SOFL system.

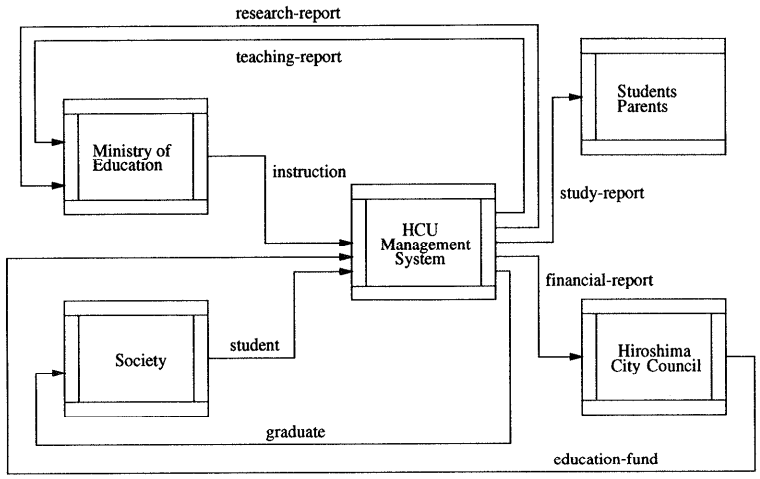

Fig. 6 Highest-level CDFD for the HCU Management System.

how to use SOFL to model this system with the new decomposition rule and how to check the structural consistency. Since the focus of this paper is not SOFL itself, we do not explain all the constructs and notions unless they are necessary for this paper. Readers interested in the details of SOFL can consult our previous paper [8].

4.1 System Modeling

The system is modeled as the CDFD in Fig. 6, and the condition process HCU Management System is decomposed into the CDFD in **Fig. 7**. This has more inputs and outputs than the HCU Management System; these are evolved from the functional constraints of the HCU Management System based on further requirements. Furthermore, the condition process Faculty is decomposed into the CDFD in **Fig. 8**. For each level of CDFD, we construct a specification module to precisely specify the functionalities of all the condition processes in that CDFD and to give informal comments about each condition process. For reasons of space, we give only the formal specification for some condition processes in this CDFD.

Note that the condition process Students-Parents in Fig. 6 is called the *output condition process*, because it takes input data flows and

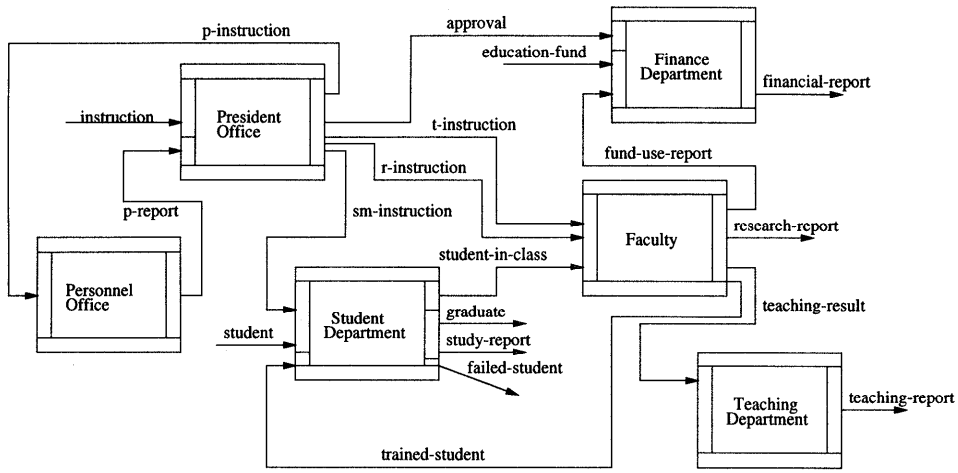

Fig. 7 Decomposition of the HCU Management System.

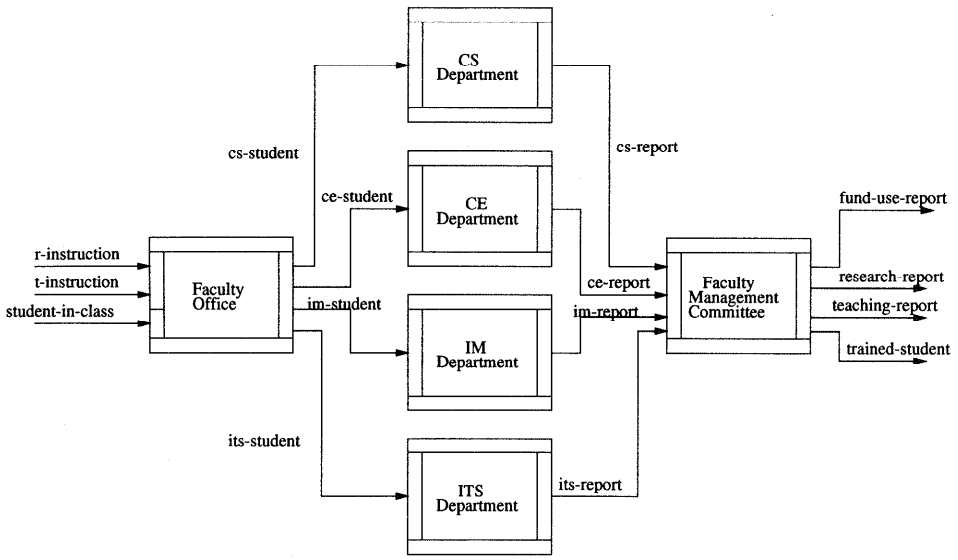

Fig. 8 Decomposition of Faculty.

produces no output data flows for the system—all the outputs go outside the system. In this sense it is an output condition process. In contrast, SOFL uses *input condition processes*, which take no input data flows from the system and produce output data flows to the system.

s-module TOP: Fig. 6;

/* This specification module corresponds to Fig. 6, i.e., this module defines all the necessary data flows (variables) and condition processes in Fig. 6. */

type
Publication = **composed of**
authors: **seq of string**;
title: **string**;
type-of-publication:
{BOOK, PAPER};
publisher: **string**;
volume: **natural**;
number: **natural**;
year: **natural**;
end;

Project = **composed of**
representative:
string * string * string;
title: **string**;
sponsors: **seq of string**;

```
        grant:      real;
        end;

/*A representative consists of name, faculty, and position */

R-Report = composed of
        pub-list:       seq of Publication;
        proj-list:      seq of Project;
        end;

Course = composed of
        title:          string;
        teacher:        string;
        teaching-hours: real;
        end;

Course-info = composed of
        course:         Course;
        pass-stu-no:    natural;
        failed-stu-no:  natural;
        end;

/* SOFL is case-sensitive, and thus course is different from Course */

T-Report = seq of Course-info

Student-info = composed of
        name: string;
        id-no: natural;
        sex: {M, F};
        date-of-birth:
        natural * natural * natural;
        course-score-list:
                map Course to real;
        end;

F-Report = composed of
        item:           string;
        expense:        real;
        end;

var
  research-report:      R-Report;
  teaching-report:      T-Report;
  instruction:          string;
  student:              seq of Student-info;
  graduate:             seq of Student-info;
  study-report:         map Course to real;
  education-fund:       real;
  finance-report:       set of F-report;

c-process Ministry-of-Education (teaching-report ⊙ research-report) instruction
  pre true
  post true
comment
This condition process gives instruction to Hiroshima City University based on the annual research-report and teaching-report from the university. However, it is difficult to model precisely how this instruction is made at the abstract level. For this reason, we let both the pre- and post-conditions be true. This means that there are currently no constraints on the input and output data flows, but they may be added as the system evolves.

The operator ⊙ between research-report and teaching-report means that when both research-report and teaching-report are available, this condition process can be fired (or executed or carried out).
end-process;

c-process HCU-Management-System (instruction ⊙ education-fund ⊙ student) teaching-report ⊙ research-report ⊙ study-report ⊙ financial-report ⊙ graduate
pre  len(student) > 0 and education-fund > 10000 and instruction ≠ " "
post len(research-report.pub-list) > 5 and len(research-report.proj-list) ≥ 1 and len(teaching-report) > 15 and forall[x inset elems(teaching-report) | x.pass-stu-no * 100 /(x.pass-stu-no + x.failed-stu-no) > 80]
decomposition Dec-HCU
comment
The pre-condition of this condition process requires that student (a sequence of Student-info) not be an empty sequence that education-fund be greater than 10,000 US dollars, and that instruction be given.

The post-condition requires that in the research-report more than five publications be provided (len(research-report.pub-list) > 5) that in the teaching-report more than fifteen courses be offered for students (len(teaching-report) > 15), and that for every course in the report the passing rate be greater than 80% (forall[x inset elems (teaching-report) | x.pass-stu-no * 100 /(x.pass-stu-no + x.failed-stu-no) > 80]). In SOFL we use forall[$P_1$, $P_2$] instead of $\forall_{P_1} \cdot P_2$ to achieve better readability.

The post-condition of this condition process does not specify anything concerning the variables study-report and financial-report.

Since it is still unclear precisely what relationship exists between input data flows (variables) or output data flows at this moment, we use ⊙ to abstract their relationship. Once this condition process is decomposed into the immediately lower-level, this relationship may be
```

refined to ⊕ if it accurately fits the situation.

end-process;

...

end-module;

Note that we can add comments anywhere, enclosed between /* and */, as shown in the module TOP, to provide an explanation or interpretation of any relevant matters (e.g., the meanings of types and variables). In addition to this, each condition process is also provided with a comment part starting with the keyword **comment**, in which anything concerned with the functionality of the condition process can be explained.

s-module Dec-HCU: Fig. 7;

/* Below we only give the declarations of the types and variables to be used in the condition process specification shown in this module. According to SOFL, this module can inherit (i.e., use) all the types and variables declared in the higher-level module (e.g., TOP) */

var

t-instruction: **string**;
student-in-class: **seq of** Student-info;
fund-use-report: **set of** F-Report;

c-process FACULTY (t-instruction ⊙ r-instruction ⊙ student-in-class) fund-use-report ⊙ research-report ⊙ teaching-result ⊙ trained-student

pre **len**(student-in-class) > 200 **and** t-instruction ≠ " " **and** r-instruction ≠ " "
post **post**-HCU-Management-System

decomposition Dec-FACULTY

comment

This condition process takes t-instruction, r-instruction, and student-in-class as its input data and produces fund-use-report, research-report, teaching-result, and trained-student. The pre-condition requires that more than 200 students be contained in the input variable student-in-class, and that neither t-instruction nor r-instruction be empty. The post-condition is the same as that of the condition process HCU Management System (which is indicated by **post**-HCU Management System).

end-process;

...

end-module

s-module Dec-FACULTY: Fig. 8;

/* The contents of this module are omitted. */

...

end-module;

4.2 Consistency Checking

We need to check whether a decomposed CDFD is consistent with its abstract condition process in terms of input and output data flows. Since input or output data flows of a condition process may have the *exclusive relationship* denoted by ⊕, we cannot simply check the following conditions as given above:

(1) $Input(P) \subseteq Input(D)$
(2) $Output(P) \subseteq Output(D)$

where P is a condition process and D is its decomposed CDFD. The reason for this is that variables representing data flows in $Input(P)$ and $Input(D)$ or $Output(P)$ and $Output(D)$ may have either the ⊙ or ⊕ relationship. We therefore modify this rule as follows:

About input data flows:

(1) $Input(P) \subseteq Input(D)$
(2) $\forall_{x,y \in Input(P)} \cdot x \odot y \Rightarrow x, y \in Input(D) \wedge x\ op\ y$
(3) $\forall_{x,y \in Input(P)} \cdot x \oplus y \Rightarrow x, y \in Input(D) \wedge x \oplus y$

About output data flows:

(4) $Output(P) \subseteq Output(D)$
(5) $\forall_{x,y \in Output(P)} \cdot x \odot y \Rightarrow x, y \in Output(D) \wedge x\ op\ y$
(6) $\forall_{x,y \in Input(P)} \cdot x \oplus y \Rightarrow x, y \in Input(D) \wedge x \oplus y.$

where *op* represents either ⊙ or ⊕.

These rules state three things. First, if x is an input data flow to the high-level condition process P, then it should also be an input data flow to its decomposed CDFD D (likewise, if it is an output of P, it should be an output of D). Second, if x and y are two input or output data flows of P and their relationship is ⊙, then their relationship in D can be either ⊙ or ⊕. In other words, we consider either of $x \odot y$ and $x \oplus y$ in D as a refinement of $x \odot y$ in P. It is not difficult to understand that $x \odot y$ in D is a refinement of itself in P, but perhaps it is difficult to understand that $x \oplus y$ in D is also a refinement of $x \odot y$ in P. In fact, $x \odot y$ in P may be used to represent two situations: one is that x and y do have a clear relationship ⊙ in P (both the designer and the user agree on this), while the other is that the precise relationship between x and y is unclear (or uncertain) in P for some reason. Therefore, once the precise

Table 1 Relationships between input data flows.

	instruction	student	education-fund
instruction		$\odot$	$\odot$
student	$\odot$		$\odot$
education-fund	$\odot$	$\odot$	

Table 2 Relationships between output data flows.

	research-report	teaching-report	finance-report	study-report	graduate	failed-student
research-report		$\odot$	$\odot$	$\oplus$	$\oplus$	$\oplus$
teaching-report	$\odot$		$\odot$	$\oplus$	$\oplus$	$\oplus$
financial-report	$\odot$	$\odot$		$\oplus$	$\oplus$	$\oplus$
study-report	$\oplus$	$\oplus$	$\oplus$		$\odot$	$\oplus$
graduate	$\oplus$	$\oplus$	$\oplus$	$\odot$		$\oplus$
failed-student	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	

relationship becomes clear as $\oplus$, $x \odot y$ can be refined into $x \oplus y$.

An algorithm for obtaining *Input*(D) and *Output*(D) as well as the relationships between input or output variables have been developed [3),8)] (the algorithm appears in the Appendix A). These two sets of input and output data flows and their relationships are expressed by *ext-inp*(D) and *ext-outp*(D) in the algorithm. With this information we can check the consistency of *Input*(P) and *Input*(D) and also that of *Output*(P) and *Output*(D) according to the above rules.

For example, by applying this algorithm to the CDFD in Fig. 7, we obtain the following results:

ext-inp(D) = instruction $\odot$ student $\odot$ education-fund,
ext-outp(D) = research-report $\odot$ teaching-report $\odot$financial-report $\oplus$ study-report $\odot$ graduate $\oplus$ failed-student

Since the process of deriving *ext-inp*(D) and *ext-outp*(D) is complicated, we include it in Appendix B for reference.

By extracting all the data variables from *ext-inp*(D) and *ext-outp*(D) we obtain the two sets:

Input(D) = {instruction, student, education-fund}
Output(D) = {research-report, teaching-report, finance-report, study-report, graduate, failed-student}.

The relationships between the data variables in *ext-inp*(D) and *ext-outp*(D) are expressed in **Tables 1** and **2**, which facilitate the algorithm for consistency-checking given below.

The first table, called M_{in}, shows the relationships between input data flows (variables). It is constructed on the basis of *ext-inp*(D) by using the following rules:

- Each input data flow corresponds to a row and column.
- For $x_1 \odot x_2 \odot ... \odot x_n$, let $M_{in}[x_i, x_j] = \odot$ and $M_{in}[x_j, x_i] = \odot$, where $i \neq j \wedge i, j \in [1..n]$.
- For $x_1 \odot x_2 \ ... \ \odot x_n \oplus y_1 \odot y_2 \ ... \ \odot y_m$, let $M_{in}[x_i, y_j] = \oplus$ and $M_{in}[y_j, x_i] = \oplus$, where $1 \leq i \leq n$, $1 \leq j \leq m$. Note that y_j can be x_i.

The second table, called M_{out}, can be similarly constructed on the basis of *ext-outp*(D), but M_{in} should be replaced with M_{out} in the rules.

An algorithm for checking the structural consistency by using the new decomposition rule is given below. In this algorithm we use the matrices M_{in}^P and M_{out}^P to represent the relationships between the input data flows and output data flows, respectively, of the upper level condition process P, and the matrices M_{in}^D and M_{out}^D to represent the relationships between the input data flows and output data flows, respectively, of its decomposition (i.e., the decomposed CDFD of P).

Algorithm 4.1:

(1) X := *Input*(P); *consistent* := **false**; *checkdone* = **true**; $M_1 := M_{in}^P$; $M_2 := M_{in}^D$;

(2) For any $x, y \in X$,
if $M_1[x, y] = \odot \wedge (M_2[x, y] = \odot \vee M_2[x, y] = \oplus)$
then goto (3)
else begin *checkdone* := **false**; **goto (4) end**;

(3) **if** for every $x, y \in X$ **(2)** is performed
then goto (4)
else begin $(x, y) := GETNEW(X)$;
goto (2) end;

(4) **if** *checkdone* = **true**
then if $X = Input(P)$
then begin $X := Output(P)$;
$M_1 := M^P_{out}$; $M_2 := M^D_{out}$;
goto (2) end
else *consistent* = **true**
else *consistent* = **false**

where $(x, y) := GETNEW(X)$ is a function for obtaining an unchecked pair of data flow variables from X and assigning them to x and y respectively.

This algorithm first checks the relationships between the input data flows of the upper-level condition process and its decomposition and then checks that between their output data flows. Only when both relationships satisfy the decomposition rules can the structural consistency of the hierarchical CDFDs be confirmed.

Compared with the input and output data flows of the condition process HCU-Management-System in Fig. 6, it is obvious that the new decomposition rule given above is satisfied by this CDFD and the condition process.

4.3 Evaluation of the Decomposition Rules

The new decomposition rules substantially affect the use of traditional DFDs in several respects. First, they change the way in which people use DFDs hierarchically. The decomposition of an upper-level condition process is in fact its precise definition, while its own occurrence on the upper-level CDFD is treated as a reference (or calling). Thus, only the necessary input and output data flows (or parameters) for the upper-level condition process are given in the diagram, while other input or output flows (i.e., additional data flows) for defining its functionality are given in its decomposition. When attempting to understand an upper-level condition process, one needs to read both its own occurrence and its decomposition. The function of the lowest-level condition process is completely specified by its own occurrence in the diagram and by the formal specification given in the specification module. In other words, such a decomposition is a refinement rather than just a detailed description.

Second, it provides an effective way of capturing the most essential input and/or output data flows at each level so that unnecessary input or output data flows can be hidden away from the current level for readability and maintainability.

Third, application of the new rules can be automatically supported by means of the algorithms given in this section and Appendix A, although the efficiency of the algorithms needs to be improved.

It is, however, worth mentioning that the proposed decomposition rules may not be correctly used if they are applied to traditional DFDs, because of the difference in semantics between CDFDs and DFDs, as explained in Section 3.2. Only if DFDs used to describe dynamic relation between processes based on an operational semantics, is it sensible to apply the proposed decomposition rules in guiding and checking the construction of hierarchical DFDs.

5. Broken Data Flows

A *broken data flow* is a data flow that exists in a CDFD but is drawn in a *broken* fashion, using *connecting points*. The motivation behind broken data flows is to avoid *cross-drawing* of data flows and to express data flows that must be drawn over one page, therefore allowing CDFDs to more neatly express the most important and relevant information flows among condition processes. For example, in **Fig. 9** (a) data y is needed by condition processes B, C, D, and E. Therefore, we need to draw all the arcs from condition process A to B, C, D, and E, but it may be impossible to avoid cross drawing with the data flow q, which is cumbersome and unclear. This problem will become more serious for more complex CDFDs. However, if we draw y as a broken data flow to C, D, and E using the connecting point (1) as shown in Fig. 9 (b), the DFD looks much neater than the one in Fig. 9 (a). This advantage will become more obvious for more complex CDFDs.

The broken data flows are in fact semantically the same as normal data flows, but different representationally (i.e., syntactically). Therefore, their occurrences do not affect the definition of related condition processes in the specification module.

6. Conclusions

In this paper we have suggested that the existing decomposition rule for data flow diagrams be not effective for realistic systems analysis, and have proposed an improved rule. The improved rule does not require that the data flows coming into and going out of a process at one level correspond to the data flows coming into and going out of an entire DFD at the next lower level. Instead, additional data flows

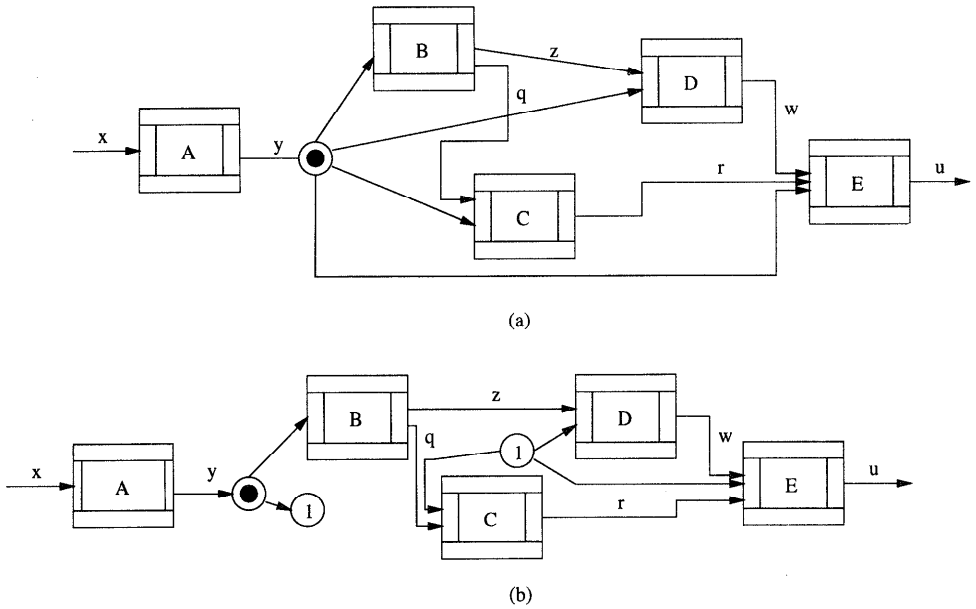

Fig. 9 An example of broken data flows.

in the DFD at the immediatelly lower-level are allowed. This change leads to an *evolutionary decomposition* methodology for systems analysis using DFDs, which we believe is more practical.

We are currently interested in investigating the rules for checking whether the pre- and post-conditions of a high-level condition process are consistent with the pre- and post-conditions of its decomposed CDFD. To this end, we need to first work out how to generate the pre- and post-conditions of a CDFD based on its semantics and then design an algorithm to check the consistency between a condition process and its decomposed CDFD.

Acknowledgments We would like to thank all of our colleagues who have provided us with help in various ways during this research, especially M. Sato and N. Arai of Hiroshima City University, and C. Ho-Stuart of Queensland University of Technology in Australia. We also thank the Ministry of Education of Japan and Hiroshima City University for funding this research under a Joint Research Grant-in-Aid for International Scientific Research FM-ISEE (08044167) and a Hiroshima City University Grant for Special Academic Research (International Studies) SCS-FM (A440), respectively.

References

1) Yourdon, E.: *Modern Structured Analysis*, Prentice Hall International (1989).
2) DeMarco, T.: *Structured Analysis and System Specification*, Yourdon, New York (1978).
3) Tse, T.H. and Pong, L.: Towards a Formal Foundation for DeMarco Data Flow Diagrams, *Comput. J.*, Vol.32, No.1, pp.1–11 (1989).
4) Adler, M.: An Algebra for Data Flow Diagram Process Decomposition, *IEEE Trans. Softw. Eng.*, Vol.14, No.2 (1988).
5) Liu, S. and Sun, Y.: Structured Methodology + Object-Oriented Methodology + Formal Methods: Methodology of SOFL, *Proc. First IEEE International Conference on Engineering of Complex Computer Systems* (Ft.Landerdale, FL, USA, November 6–10), pp.137–144, IEEE Computer Society Press (1995).
6) Hall, A.: Seven Myths of Formal Methods, *IEEE Software*, pp.11–19 (1990).
7) Craigen, D., Gerhart, S. and Ralston, T.: Formal Methods Reality Check: Industrial Usage, *FME'93: Industrial-Strength Formal Methods*, (Odense, Denmark), pp.250–267, Springer-Verlag (1993).
8) Liu, S.: A Formal Requirements Specification Method Based on Data Flow Analysis, *Journal of Systems and Software*, Vol.21, pp.141–149 (1993).
9) Liu, S., Stavridou, V. and Dutertre, B.: The Practice of Formal Methods in Safety Critical Systems, *Journal of Systems and Software*, Vol.28, pp.77–87 (1995).
10) Liu, S., Offutt, A.J., Ho-Stuart, C., Sun, Y. and Ohba, M.: SOFL: A Formal Engineering Methodlogy for Industrial Applications, *IEEE Trans. Softw. Eng.*, Vol.24, No.1 (1998).

Appendix A

Algorithm:

1. Let $P_1, P_2, \ldots, P_n$ be the condition processes of the CDFD D. Relate each P_i with a data transformation of form:
   $L(P_i) \longrightarrow R(P_i)$
   Hence we represent D by a combined transformation:
   $L_1 \longrightarrow R_1 \odot L_2 \longrightarrow R_2 \odot \cdots \odot L_n \longrightarrow R_n$.
2. For every transformation of the form $(L_1 \oplus L_2 \oplus \ldots \oplus L_k) \longrightarrow R$, expand it to:
   $L_1 \longrightarrow R \oplus L_2 \longrightarrow R \oplus \ldots \oplus L_k \longrightarrow R$
3. For every transformation of the form:
   $L \longrightarrow (R_1 \oplus R_2 \oplus \ldots \oplus R_l)$
   expand it to:
   $L \longrightarrow R_1 \oplus L \longrightarrow R_2 \oplus \ldots \oplus L \longrightarrow R_l$
4. For every transformation of the form
   $L \longrightarrow (R_1 \odot R_2 \odot \ldots \odot R_t)$,
   expand it to:
   $L \longrightarrow R_1 \odot L \longrightarrow R_2 \odot \ldots \odot L \longrightarrow R_t$
5. Using the distributive property of the operator "$\odot$" over the operator "$\oplus$", expand the expressions of transformations. For example,
   $L_1 \longrightarrow R_1 \odot (L_2 \longrightarrow R_2 \oplus L_3 \longrightarrow R_3)$
   becomes
   $L_1 \longrightarrow R_1 \odot L_2 \longrightarrow R_2 \oplus L_1 \longrightarrow R_1 \odot L_3 \longrightarrow R_3$
6. We define a path as a combination of transformations joined together only by "$\odot$"s but not "$\oplus$"s. Do the following for each path:
   (6.1) For any $L' \longrightarrow R'$ such that R' is a subexpression of L for some $L \longrightarrow R$ in the same path.
   $\langle a \rangle$ Substitute L' for R' in the transformation $L \longrightarrow R$;
   $\langle b \rangle$ Remove $L' \longrightarrow R'$ from the path.
   (6.2) Remove all transformations $L \longrightarrow R$ such that
   $\langle a \rangle$ $R \notin ext\text{-}outpset(D)$ or
   $\langle b \rangle$ $\exists_{d \in S(L)} \cdot d \notin ext\text{-}inpset(D)$.
7. Combine transformations within a path into a single transformation by converting
   $L_1 \longrightarrow R_1 \odot L_2 \longrightarrow R_2 \odot \ldots \odot L_k \longrightarrow R_k$
   into
   $(L_1 \odot L_2 \odot \ldots \odot L_k) \longrightarrow (R_1 \odot R_2 \odot \ldots \odot R_k)$.
8. Combine all the transformations into a single transformation by performing the following two operations:
   (8.1) Convert
   $L_1 \longrightarrow R_1 \oplus L_2 \longrightarrow R_2 \oplus \ldots \oplus L_t \longrightarrow R_t$
   into
   $(L_1 \oplus L_2 \oplus \ldots \oplus L_t) \longrightarrow (R_1 \oplus R_2 \oplus \ldots \oplus R_t)$.
   (8.2) Within $(L_1 \oplus L_2 \oplus \ldots \oplus L_t) \longrightarrow (R_1 \oplus R_2 \oplus \ldots \oplus R_t)$, for every L_i ($i \in [1..t]$) (or R_i), if $L_j = L_i$ ($j \neq i \wedge j \in [1..t]$) (or $R_j = R_i$), then delete L_j (or R_j) and the left $\oplus$ to L_j (or R_j) (if applicable).
9. Let $L \longrightarrow R$ be the resulting transformation. Then
   (a) $ext\text{-}inp(D) = L$;
   (b) $ext\text{-}outp(D) = R$;

where $ext\text{-}inp(D)$(ext−outp(D)) is a data expression in which all the input (output) variables are in the set $Input(D)$ ($Output(D)$) and are connected with the operators $\odot$ or $\oplus$. For example, let $ext\text{-}inp(D) = x \odot y \oplus z$, then $Input(D) = \{x, y, z\}$.

Table 3 Simpler variables for data flows.

x_1	=	instruction
x_2	=	student
x_3	=	education-fund
y_1	=	research-report
y_2	=	teaching-report
y_3	=	financial-report
y_4	=	study-report
y_5	=	graduate
y_6	=	failed-student
z_1	=	p-report
z_2	=	p-instruction
z_3	=	approval
z_4	=	t-instruction
z_5	=	r-instruction
z_6	=	sm-instruction
z_7	=	student-in-class
z_8	=	fund-use-report
z_9	=	teaching-result
z_{10}	=	trained-student

Appendix B

For the sake of readability, we use simpler variables to replace the data flow variables as given in **Table 3** in the expressions derived by applying the algorithm given in Appendix A.

We use $\models (n)$ to mean that the following ex-

pression is derived by applying step n in the algorithm. For the sake of readability, we also omit some steps that are not important in deriving the final expected result.

$\models$ (1)
$(x_1 \oplus z_1 \longrightarrow z_2 \odot z_3 \odot z_4 \odot z_5 \odot z_6) \odot$
$(z_2 \longrightarrow z_1) \odot$
$(z_6 \odot x_2 \oplus z_{10} \longrightarrow z_7 \oplus y_5 \odot y_4 \oplus y_6) \odot$
$(z_3 \oplus x_3 \odot z_8 \longrightarrow y_3) \odot$
$(z_4 \odot z_5 \odot z_7 \longrightarrow z_8 \odot y_1 \odot z_9 \odot z_{10}) \odot$
$(z_9 \longrightarrow y_2)$
$\models$ (2)
$((x_1 \longrightarrow z_2 \odot z_3 \odot z_4 \odot z_5 \odot z_6) \oplus$
$(z_1 \longrightarrow z_2 \odot z_3 \odot z_4 \odot z_5 \odot z_6)) \odot$
$(z_2 \longrightarrow z_1) \odot$
$((z_6 \odot x_2 \longrightarrow z_7 \oplus y_5 \odot y_4 \oplus y_6) \oplus$
$(z_{10} \longrightarrow z_7 \oplus y_5 \odot y_4 \oplus y_6)) \odot$
$((z_3 \longrightarrow y_3) \oplus (x_3 \odot z_8 \longrightarrow y_3)) \odot$
$(z_4 \odot z_5 \odot z_7 \longrightarrow z_8 \odot y_1 \odot z_9 \odot z_{10}) \odot$
$(z_9 \longrightarrow y_2)$
$\models$ (3)
$((x_1 \longrightarrow z_2 \odot z_3 \odot z_4 \odot z_5 \odot z_6) \oplus$
$z_1 \longrightarrow z_2 \odot z_3 \odot z_4 \odot z_5 \odot z_6)) \odot$
$(z_2 \longrightarrow z_1) \odot$
$((z_6 \odot x_2 \longrightarrow z_7) \oplus (z_6 \odot x_2 \longrightarrow y_5 \odot y_4) \oplus$
$(z_6 \odot x_2 \longrightarrow y_6) \oplus$
$(z_{10} \longrightarrow z_7) \oplus (z_{10} \longrightarrow y_5 \odot y_4) \oplus (z_{10} \longrightarrow y_6) \odot$
$((z_3 \longrightarrow y_3) \oplus (x_3 \odot z_8 \longrightarrow y_3)) \odot$
$(z_4 \odot z_5 \odot z_7 \longrightarrow z_8 \odot y_1 \odot z_9 \odot z_{10}) \odot$
$(z_9 \longrightarrow y_2)$
$\models$ (4)
$((x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot$
$(x_1 \longrightarrow z_4) \odot (x_1 \longrightarrow z_5) \odot$
$(x_1 \longrightarrow z_6) \oplus (z_1 \longrightarrow z_2) \odot$
$(z_1 \longrightarrow z_3) \odot$
$(z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6)) \odot$
$(z_2 \longrightarrow z_1) \odot$
$((z_6 \odot x_2 \longrightarrow z_7) \oplus (z_6 \odot x_2 \longrightarrow y_5) \odot$
$(z_6 \odot x_2 \longrightarrow y_4) \oplus (z_6 \odot x_2 \longrightarrow y_6) \oplus$
$(z_{10} \longrightarrow z_7) \oplus (z_{10} \longrightarrow y_5) \odot (z_{10} \longrightarrow y_4) \oplus$
$(z_{10} \longrightarrow y_6)) \odot$
$((z_3 \longrightarrow y_3) \oplus (x_3 \odot z_8 \longrightarrow y_3)) \odot$
$((z_4 \odot z_5 \odot z_7 \longrightarrow z_8) \odot (z_4 \odot z_5 \odot z_7 \longrightarrow y_1) \odot$
$(z_4 \odot z_5 \odot z_7 \longrightarrow z_9) \odot (z_4 \odot z_5 \odot z_7 \longrightarrow z_{10})) \odot$
$(z_9 \longrightarrow y_2)$
$\models$ (5)
$((A) \odot (z_6 \odot x_2 \longrightarrow z_7) \oplus (A) \odot (z_6 \odot x_2 \longrightarrow y_5) \odot$
$(z_6 \odot x_2 \longrightarrow y_4) \oplus (A) \odot (z_6 \odot x_2 \longrightarrow y_6) \oplus$
$(A) \odot (z_{10} \longrightarrow z_7) \oplus (A) \odot (z_{10} \longrightarrow y_5) \odot$
$(z_{10} \longrightarrow y_4) \oplus$
$(A) \odot (z_{10} \longrightarrow y_6)) \odot$
$((B) \odot (z_3 \longrightarrow y_3) \oplus (B) \odot (x_3 \odot z_8 \longrightarrow z_1) \odot$
$(x_3 \odot z_8 \longrightarrow y_3)) \odot$
$(z_4 \odot z_5 \odot z_7 \longrightarrow z_8) \odot (z_4 \odot z_5 \odot z_7 \longrightarrow y_1) \odot$
$(z_4 \odot z_5 \odot z_7 \longrightarrow z_9) \odot (z_4 \odot z_5 \odot z_7 \longrightarrow z_{10}) \odot$
$(z_9 \longrightarrow y_2)$
$\models$ (5)
$(B) \odot (x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot$
$(x_1 \longrightarrow z_4) \odot (x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow z_7) \odot (z_3 \longrightarrow z_1) \odot (z_3 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow z_7) \odot (z_3 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2) \longrightarrow y_5 \odot (z_6 \odot x_2 \longrightarrow y_4) \odot$
$(z_3 \longrightarrow z_1) \odot (z_3 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow y_5) \odot (z_6 \odot x_2 \longrightarrow y_4) \odot$
$(z_3 \longrightarrow z_1) \odot (z_3 \longrightarrow y_3) \oplus (x_1 \longrightarrow z_2) \odot$
$(x_1 \longrightarrow z_3) \odot (x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow y_6) \odot (z_3 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot$
$(z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow y_6) \odot (z_6 \odot x_2 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow z_7) \odot$
$(z_3 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow y_6) \odot (z_3 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot (x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow z_7) \odot$
$(z_3 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow z_7) \odot$
$(z_3 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot$
$(x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow y_5) \odot$
$(z_{10} \longrightarrow y_4) \odot (z_3 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$

$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow y_5) \odot$
$(z_{10} \longrightarrow y_4) \odot (z_3 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot (x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow y_6) \odot$
$(z_3 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow y_6) \odot$
$(z_3 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot (x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow z_7) \odot (x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow z_7) \odot (x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot (x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow y_5) \odot (x_7 \odot x_2 \longrightarrow y_4) \odot$
$(x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow y_5) \odot (x_7 \odot x_2 \longrightarrow y_4) \odot$
$(x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot (x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow y_6) \odot (x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot$
$(z_6 \odot x_2 \longrightarrow y_6) \odot (x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot (x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow z_7) \odot$
$(x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow z_7) \odot$
$(x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot (x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow y_5) \odot$
$(z_{10} \longrightarrow y_4) \odot (x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow y_5) \odot$
$(z_{10} \longrightarrow y_4) \odot (x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot (x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow y_6) \odot$
$(x_3 \odot z_8 \longrightarrow y_3) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot$
$(z_2 \longrightarrow z_1) \odot (z_9 \longrightarrow y_2) \odot (z_{10} \longrightarrow y_6) \odot$
$(x_3 \odot z_8 \longrightarrow y_3)) \odot (z_4 \odot z_5 \odot z_7 \longrightarrow z_8) \odot$
$(z_4 \odot z_5 \odot z_7 \longrightarrow y_1) \odot (z_4 \odot z_5 \odot z_7 \longrightarrow z_9) \odot$
$(z_4 \odot z_5 \odot z_7 \longrightarrow z_{10}) \odot (z_9 \longrightarrow y_2)$
$\models (5,6)$
$(x_1 \odot x_2 \odot x_3 \longrightarrow y_1) \odot (x_1 \odot x_2 \odot x_3 \longrightarrow y_2) \odot$
$(x_1 \odot x_2 \odot x_3 \longrightarrow y_3) \oplus$
$(x_1 \odot x_2 \longrightarrow y_4) \odot (x_1 \odot x_2 \longrightarrow y_5) \odot$
$(x_1 \odot x_2 \longrightarrow y_6)$
$\models (7)$
$(x_1 \odot x_2 \odot x_3 \odot x_1 \odot x_2 \odot x_3 \odot x_1 \odot x_2 \odot x_3$
$\longrightarrow y_1 \odot y_2 \odot y_3) \oplus$
$(x_1 \odot x_2 \odot x_1 \odot x_2 \longrightarrow y_4 \odot y_5) \oplus$
$x_1 \odot x_2 \longrightarrow y_6)$
$\models (8.1)$
$x_1 \odot x_2 \odot x_3 \odot x_1 \odot x_2 \odot x_3 \odot x_1 \odot x_2 \odot x_3 \oplus x_1 \odot x_2 \odot$
$x_1 \odot x_2 \oplus x_1 \odot x_2 \longrightarrow y_1 \odot y_2 \odot y_3 \oplus y_4 \odot y_5 \oplus y_6$
$\models (8.2)$
$x_1 \odot x_2 \odot x_3 \longrightarrow y_1 \odot y_2 \odot y_3 \oplus y_4 \odot y_5 \oplus y_6$
$\models (9)$
$ext\text{-}inp(D) = x_1 \odot x_2 \odot x_3$
$ext\text{-}outp(D) = y_1 \odot y_2 \odot y_3 \oplus y_4 \odot y_5 \oplus y_6$
where
$A =$ $((x_1 \longrightarrow z_2) \odot (x_1 \longrightarrow z_3) \odot x_1 \longrightarrow z_4) \odot$
$(x_1 \longrightarrow z_5) \odot (x_1 \longrightarrow z_6) \odot (z_2 \longrightarrow z_1) \odot$
$(z_9 \longrightarrow y_2) \oplus$
$(z_1 \longrightarrow z_2) \odot (z_1 \longrightarrow z_3) \odot (z_1 \longrightarrow z_4) \odot$
$(z_1 \longrightarrow z_5) \odot (z_1 \longrightarrow z_6) \odot (z_2 \longrightarrow z_1) \odot$
$(z_9 \longrightarrow y_2))$
$B =$ $((A) \odot (z_6 \odot x_2 \longrightarrow z_7) \oplus (A) \odot$
$(z_6 \odot x_2 \longrightarrow y_5) \odot (z_6 \odot x_2 \longrightarrow y_4) \oplus (A) \odot$
$(z_6 \odot x_2 \longrightarrow y_6) \oplus$
$(A) \odot (z_{10} \longrightarrow z_7) \oplus (A) \odot (z_{10} \longrightarrow y_5) \odot$
$(z_{10} \longrightarrow y_4) \oplus$
$(A) \odot (z_{10} \longrightarrow y_6))$

(Received June 21, 1996)
(Accepted March 6, 1998)

Shaoying Liu is an associate professor in the Computer Science Department at Hiroshima City University. He holds B.S. and M.S. degrees in Computer Science from Xi'an Jiaotong University, The People's Republic of China, and a Ph.D. in Formal Methods from the University of Manchester, the United Kingdom. His research interests include formal engineering methods, software development methodology, software evolution, software testing, software engineering environments, formal languages, and safety-critical systems. Dr. Liu received an "Outstanding Paper Award" at the Second IEEE International Conference on Engineering of Complex Computing Systems (ICECCS'96) and has over 35 publications in refereed journals and international conferences. He served as the General Chair of First IEEE International Conference on Formal Engineering Methods (ICFEM'97) and Co-Chair of formal methods track of Third IEEE International Conference on Engineering of Complex Computing Systems (ICECCS'97). He is a member of IEEE Computer Society and IEICE Japan.

A Jeff. Offutt is an Associate Professor of Information and Software Systems Engineering at George Mason University. His current research interests include program testing and automatic test data generation, software reliability, module and integration testing, formal methods, and change-impact analysis. He has published over fourty research papers in refereed computer science journals and conferences. Offutt received a Ph.D. degree in computer science from the Georgia Institute of Technology, and is a member of the ACM and IEEE Computer Society. He previously held a faculty position in the Department of Computer Science at Clemson University.

Mitsuru Ohba received the M.S. and B.S. degrees from Aoyama Gakuin University, Tokyo, in 1973 and 1971, respectively. Ohba has been a professor in the Computer Science Department at the Hiroshima City University since 1994. Ohba was a co-founder of the Software CALS national project of Japan that aimed to establish a new framework for international collabtorations using the internet. He developed his 20-year professional career with IBM. His experience with IBM involved the study of software reliability analysis, study of design notation and specifications, development of a high-speed prolog compiler, study of software testing practices, and development of software test tools.

Keijiro Araki was born in Fukuoka City, Japan. He received B.S., M.S., and Dr. degrees in Computer Science and Communication Engineering from Kyushu University in 1976, 1978, and 1982 respectively. He became a professor at Nara Institute of Science Technology in 1993, and since 1996 he is a professor at Graduate School of Information Science and Electrical Engineering, Kyushu University. His research interests include software development methods, formal specification, programming languages, internetworking, multi-media communication systems. He is a member of ACM, IEEE Computer Society, IPSJ, JSSST, SEA, etc.