A Software Design System Based on a Unified Design Methodology

OSAMU SHIGO*, KANJI IWAMOTO* and SHINYA FUJIBAYASHI*

How well to design software systems is a key to the success of software developments. In this paper, a unified design methodology, which integrates module decomposition techniques based on control flow and those based on data flow, is introduced to construct well structured software systems. In this method, a system will be designed as a hierarchy of subsystems, each of which is either a control flow combination or a data flow combination of its subsystems. Since it is possible, under a set of rules offered, to select at each refinement step a best suited decomposition technique, the methodology may be more widely applicable than any other existing ones. A software design system developed to aid the design on the new methodology is described. The system consists of a system design language, to represent the design along the new methodology, and its processor, which generates various design analysis reports. Also, a simple programming tool is provided to realize a system designed through this method in conventional programming language.

1. Introduction

Reliability and maintainability of large scale software system are mainly determined by its design quality. Generally, large scale software design is divided into the following two activities;

(1) Decomposing the whole system, step by step, into sets of simpler components with simple relations (modularization design),
(2) Designing program structures for primitive components which are not further decomposed (inner module design).

The latter is considered as designing many small programs, if each primitive component function is as separate and as simple as realizable by a small program. So, the complexity of large scale software is mostly treated in the former activity. Thus, the modularization method to overcome the complexity are of great importance in software engineering.

This paper introduces a new modularization methodology, which unifies control flow based method (using module call relation) and data flow based method (using data passing relation) to make the best use of each advantageous feature. Then, a design description language for unified design representation, based on the methodology, and its processor that analyzes the design results, are described.

Control flow based modularization has widely been applied in conventional software developments. In this method, the execution sequence of modules must completely be determined by the calling programs. However, there are many cases, in which system function may be represented as a series of mappings from input to output data. The execution sequence of the mapping processes should satisfy only a rather trivial condition: The production of a datum should precede the consumption of the datum. Thus, only partial execution order is required here.

For this case, a model (data flow model) wherein a number of processes are executed in parallel, while passing data to each other, is suitably adapted. If control flow based modularization is applied to the case, the execution order would be forced to change into complete execution order. This often tends to make programs complex and hard to change, because extra variables are embedded in several portion of programs to dynamically control the execution sequence flow. To avoid complexity, which cannot be overcome by conventional control flow based modularization, data flow based modularization is introduced in the methodology.

On the other hand, attempts to realize systems by data flow alone have been made in computer architecture research, such as the data flow machine proposed by Dennis [1]. However, data flow alone is apt to create rather more complex programs than conventional sequential ones with similar functions, because extra data, other than major data streams, are needed to dynamically control the major data flow. From understandability and flexibility viewpoints, data flow alone seems to be insufficient and impractical to resolve all kinds of problems in large scale software.

In general, it can be said that data flow is suited when functions are representable by a series of data stream transformers, while control flow is suited when functions are specified as a combination of procedures and the execution sequence provides information essential to understanding the functions. These two modularization methods are both useful. Selective usage of the two,

*Software Product Engineering Laboratory, Nippon Electric Co., Ltd., Kawasaki, Kanagama 213, Japan.

without any confusion, is desired in constructing a highly modular system.

The modularization methodology, described in this paper, consistently unifies these two modularization methods under functional hierarchical structuring technique to enhance advantageous features of each. Any existing design methodologies are effective in their own best suited cases, but are ineffective (sometimes rather harmful) for other cases, so their applicable ranges are restricted. On the contrary, by the methodology introduced here, a whole problem is divided into sets of functionally independent parts, and the best suited design method is selectively utilized for each part. Thus, the methodology is widely applicable from system programs to application programs.

According to the methodology, a large number of interrelations among components are specified during the refinement steps. Such information is effectively utilized in maintenance and modification phases, as well as in the development phase. However, in large scale software design, if this information is managed in the form of hand written documents, it is very hard to modify the documents along the design change. Also, laborious efforts are required to find any desired bit of information in the huge amount of documents, in order to validate the interrelations or to trace the modification effect. This seems to be one reason for the fact that many conventional design documents, with a large number of pages, have been rather useless, considering the huge amount of manpower needed for writing them.

Instead of document management by hand, a system with design information management facilities, using a design data base, also with design validation and document generation facilities, has been developed by the authors. Recently, similar objective support systems for requirements specification and/or design documentations have been proposed, e.g., ISDOS[2], REVS[3], and SSD [4]. The system described here consists of a design language and its processor tailored for supporting, and also enforcing, practical utilization of the design methodology. The language is aimed mainly at the formal representation of modularization design results, based on the methodology. To gain wide applicabilities, the language is independently designed from any programming languages. The processor provides facilities to: store the information described in the language in the design database, analyze interrelations among components using the database, and generate design documents supporting the methodology utilization.

The language and its processor have been developed as one of the subsystems of a total system SDMS (Software Development and Maintenance System) [5], which aims to support throughout the software life cycle using new methodologies and tools.

A new design methodology is introduced in Sec. 2. The design language and its processor are described in Sec. 3 and 4, respectively. Sec. 5 outlines programming tool features for realizing a system designed on the methodology in conventional programming languages, FORTRAN and COBOL.

2. Design Methodology

At first, problems in modularization design methods, using only a data flow or a control flow, are discussed. Then a new design methodology unifying those two is introduced.

2.1 Data Flow vs. Control Flow

Most processes in computer applications deal with sequential data streams, e.g., sequential files, line printers and communication line data, as their input and output data. In this case, a system is often designed as a combination of stepwise data transformers from input to output data, as shown in Fig. 1. Such modularization has commonly been used in conventional business application systems, where intermediate data, transformers and their combinations are realized, respectively, by sequential files, executable programs and job control descriptions. These systems are easy to understand and also to modify. However, they often tend to be so inefficient, according to the work files usage, as to be useless. To avoid the inefficiency, a control flow modularization may be used, where each transformer is realized as a routine and data transfer is implemented by routine call with parameters. However, this control flow modularization often introduces rather complicated control structure incongruent with the data sequence structure, as discussed below.

Consider, for instance, a control flow modularization in Fig. 2 being a transformation result from data flow in Fig. 1. In Fig. 2, main routine P'_3 and subroutines P'_1, P'_2 and P'_4 are resulting transformations of processes P_3, P_1, P_2 and P_4 in Fig. 1, respectively. Also, D'_1, D'_2 and D'_3 represent passing data elements accompanied with corresponding routine calls. Consider that the D_1

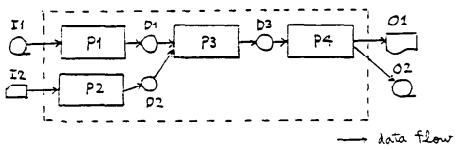


Fig. 1 Data flow modularization.

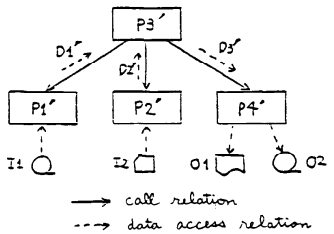


Fig. 2 Transformation of figure 1 to control flow modularization.

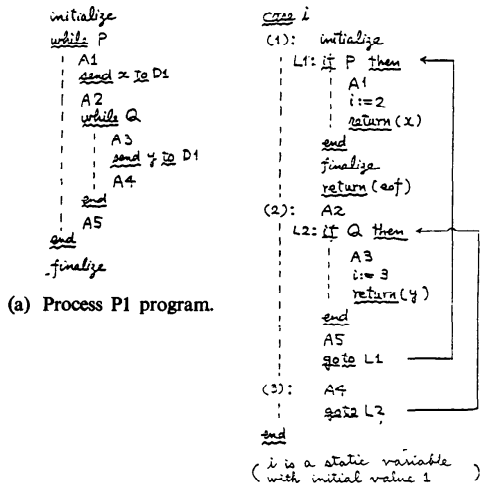
sequence structure, from producer P_1's viewpoint, is a stream of logical units, each of which consists of one x record followed by a variable number of y records. In this case, the program structure of P_1 in Fig. 1 can be represented, independently from how to receive D_1 by P_3, along the data structure as shown in Fig. 3(a). On the other hand, consider also that P_3' in Fig. 2 desires to process x or y records, obtained by P_1' call, independently from the above mentioned D_1 structure. In order that a unit process for generating one logical unit of D_1 should be performed through a series of P_1' calls, the P_1' execution to produce x or y records needs to be selected, at each call, according to the last returning state.

This causes the P_1' program structure to be rather complex and incongruent with the D_1 sequence structure, because a unit process is forced to be scattered over the program,* as shown in Fig. 3(b) [6].

In this case, the data transformers and the intermediate data can be realized as processes and message buffers,** respectively, to eliminate the inefficiency by the work files usage and also to keep the understandability of the control structure. Since, in Fig. 1, there are no requirements to wait for the P_3 execution until all D_1 and D_2 data elements are completely provided, D_1 and D_2 can be realized as message buffers, instead of work files, with no effects on the final results. This enables the programmer to utilize such a simple modularization method with file interface data in system programs without any inefficiencies. Such modularization, where processes are connected through message buffers, is called a data flow based modularization.

On the other hand, as shown in Fig. 4(a), consider that input and output data structures are matched together, and each mapping between corresponding input and output parts can be determined by the category of the parts.*** The control flow modularization, as illustrated in Fig. 4(b), can be suitably adapted now, where the main routine C decides the category of data parts to call corresponding mapping routines P_1, P_2 and P_3.

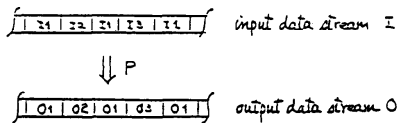
This problem is also solvable with data flow based modularization, as shown in Fig. 5. In the figure, process C_1 divides input streem I according to the category of its parts. Processes P_1, P_2 and P_3 transform each category of input to corresponding output. Process C_2 merges the outputs of the transformers to produce final output stream O. However, in addition to these major data streams another control data stream D is necessary from C_1 to C_2 to inform the input data sequence used for deciding output data sequence. In Fig. 4(b), the main routine C is considered to correspond to a combination of C_1 and C_2, so such an extra control data stream D is not needed.



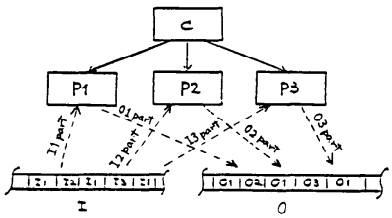
(a) Process P1 program.

(b) Routine P1′ program transformed from process P1 of (a).

Fig. 3 Process/routine transformation.



(a) Input and output data structures matching. (Ii, Oj indicate data categories).



(b) Control flow realization for (a).

Fig. 4 Control flow modularization.

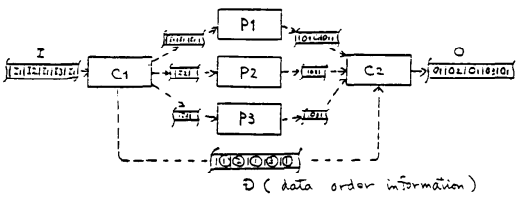


Fig. 5 Transformation of figure 4 to data flow modularization.

*In Jackson's design method [7], the problem in which the input and output data structures are incongruent is recognized as "structure clash", and an intermediate data, such as D_1, is introduced to solve the problem. Also, the transformation from P_1 to P_1' is called "program inversion". The program inversion becomes rather complicated, when the program to be inverted is large and is realized by a number of routines (refer to chapters 7 to 9 in Reference [7]).

**Message buffer is used for communication of two processes executed in parallel. It consists of a FIFO (first-in first-out) message queue with bounded capacity. When process attempts to write a message into a full queue (or read a message from an empty queue), the process is blocked until the queue is not full (or is not empty) [8].

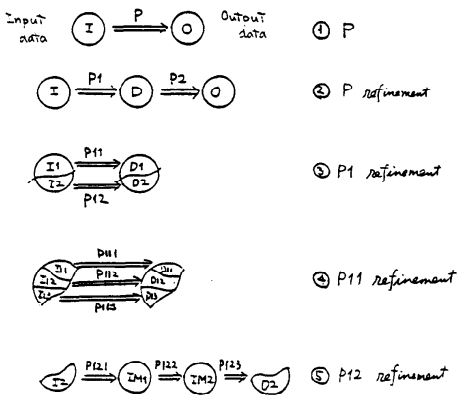
***In Jackson's design method [7], this case corresponds to the problem with input and output data structures matching (refer to Chapter 4 in Reference [7]).

In general, much usage of control variables for controlling the execution sequence for conventional sequential programs causes complex and hard to vary programs. A similar problem will occur in data flow based modularization, when extra data, other than major data to be processed, are introduced. The data flow programs described by Dennis [1] seems to be rather complicated with excessive control data and gate processes. When control flow is used in the case where data flow is more suitable, and vice versa, it can be said that the complexities are always caused by introducing extra control data.

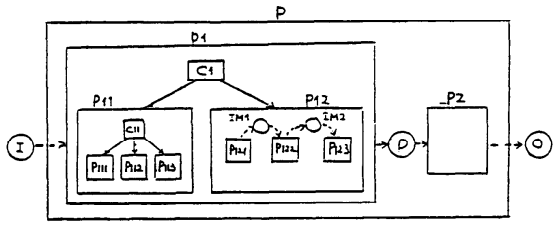
2.2 Unifying Data Flow and Control Flow

Application of the best suited modularization method for each scene, with minimal extra control data, is desired to create understandable and flexible system structures for obtaining high reliability and maintainability. For this purpose, a design methodology unifying data flow and control flow with no confusion is introduced hereafter.

For instance, consider the stepwise refinement steps of data transformer system P in Fig. 6(a), from input I to output O (shown in (1) in the figure). In the figure, (2) shows a refinement of P with intermediate data D (data flow decomposition), (3) shows a refinement of P_1 with respect to the semantic structures of I and D (control flow decomposition), (4) shows a refinement of P_{11} concerned with the more detailed data structures (control flow decomposition) and (5) shows a refinement of P_{12}



(a) Stewise refinements in data transformation.



(b) System structure based on (a).

Fig. 6 Integration of data flow and control flow.

introducing intermediate data IM_1 and IM_2 (data flow decomposition). These refinement steps may be further continued. The system construction for P can be represented hierarchically as illustrated in Fig. 6(b), where C_1 and C_{11} are simple main routines for P_1 and P_{11}, respectively.

The above refinement steps are generalized as follow:

(1) Taking note of major input and output data for a system, define the outlines of their data structure.
(2) Consider the correspondence between the input and the output data structure defined in (1).
   (a) If these two structures correspond to each other, decompose the system into control flow reflecting the data structures.
   (b) Otherwise, introduce intermediate data streams to make correspondences from input to output stepwise, and decompose the system into data flow with these intermediate data.
(3) For each component obtained through decompositions in (2), repeat (1) and (2) until the data structures are defined in detail and the component function seems to be realized by a simple and small program.

The components hierarchy obtained through these refinement steps represents inclusion relations for functions. In step (2), a component of the control flow decomposition in (a) is called *routine*, while a component of the data flow decomposition in (b) is called *process*. In the refinement steps, one can refine a component, whether it is a routine or a process, as either a control flow or a data flow, according to its input and output data structure views.

Thus, a control flow component, i.e., routine, may be realized as a data flow decomposition, as P_{12} in Fig. 6(b). This requires localized data flow linkage mechanism, as will be described in Sec. 5, which is not sufficiently implemented in any existing data flow mechanisms, e.g., PORT [9] and DSLM [10].

Beside routine and process components, so called encapsulation modules can be used for applying the data abstraction or information hiding concept [11] in use of non-sequential data, such as table and stack. The encapsulation module component in control flow is called *group* [12] and the one in data flow is called *monitor* [8], where a number of operations with common resources are integrated into one component to hide implementation details from users of the component. This enables utilizing the data abstraction methodology with the control flow and data flow based modularization methodology, enforcing each advantageous feature.

2.3 Rules for Data Flow and Control Flow Unification

In the prescribed data flow and control flow unification method based on the components functional hierarchy, one can use either a data flow or a control flow, but not both, at each refinement step. So, each method is applic-

able with no confusions, retaining individual understandability.

To make data flow applications easier to understand, the following restrictions for message buffer usages are introduced:

(1) For each message buffer, there is only one producer process sending data, and only one consumer process receiving data.
(2) There is no way an individual process can directly know whether a message buffer is empty or full. That is, there are only two operations, "send" and "receive," provided for accessing the message buffer. The process is always blocked when it attempts to execute receive (send) operation on an empty (full) buffer.

By these two restrictions, the data flow system function is guaranteed to be determined independently from the process scheduling method [13].

For component decompositions in the methodology, the following four rules are offered to guarantee unification method consistency;

(1) If a data flow connection component C is refined as data flow decomposition, each message buffer accessed by C is accessed by only one internal component (internal process) of C.
(2) If a data flow connection component C is refined as control flow decomposition, any message buffers accessed by C can be accessed by any number of internal components of C.

In (1) and (2), message buffer access method (send or receive) by internal component must be the same as the method by C (e.g., a message buffer sent by C can only be sent by internal component of C).

(3) If a control flow connection component C is refined as data flow decomposition, each routine or operation for the encapsulation module called by C is called by only one internal component of C.
(4) If a control flow connection component C is refined as control flow decomposition, any routines or operations called by C can be called by any number of internal components of C.

The above (1) and (3) are necessary to guarantee the functional determinancy of the whole system. That is, the system function is determined independently from the process scheduling method.

If no real parallel executions are performed, and a routine or an operation called by C is functional with no internal states kept, restriction (3) can be removed. That is, the routine or the operation can be called by any number of internal processes of C.

As in (2), when the message buffer is accessed from conventional sequential processing routines, the message buffer can be treated like a conventional sequential file. This enables designers and programmers, who have been used to designing conventional sequential programs, to easily utilize message buffers.

3. Design Language

Design language performs important roles, not only for uniformly representing design results, but also for guiding design considerations, and for enforcing design standardization. Design language SDL (system design language), described here, is a language for entering various kinds of design information, mainly the component decomposition results based on the prescribed design methodology, into a design database. Since readable design documents and many interrelated bits of design information can be automatically generated by the language processor, SDL is rather concerned with writability and avoiding duplicate description. The following describes specific SDL features.

3.1 Component

According to the methodology in Sec. 2, design proceeds mainly along stepwise component decomp[illegible] In SDL, each component is described as a desi[illegible]cription unit. A component represents a part of [illegible] system with functionally classified features. An entire system can also be seen as a component. In component hierarchy, a component is said to be *nonprimitive* if it is realized as a combination of its components. Otherwise, it is a *primitive*. The parent-child relations are defined as conventional in component hierarchy. Thus, a primitive component can be said to be a component having no children.

In the programming stage, a primitive component corresponds to a program module, while a nonprimitive component at most corresponds to a part of linkage control description. However, in the design stage, a nonprimitive component description in SDL has more important roles than a primitive component. It can be said that nonprimitive components represent the classification of the final program modules according to functional hierarchy.

There are six component types. Four of them have already been denoted in Sec. 2, i.e., *routine* (providing conventional subroutine facility), *process* (being a parallel processing unit), *group* (representing abstract data or machine) and *monitor* (representing commonly used abstract data among processes). In addition to these component types, *root* and *data* component types are also prepared in SDL. Root component is a topmost component in components hierarchy of a system, which describes the connections method between the entire system to be designed and its environments. Data component represents concrete data area in memory, files and databases, commonly used from other components.

These six component types indicate the component functional features, and are independent from their realization methods, such as primitive or nonprimitive. Figure 7 illustrates a component hierarchy example in SDL, where components S, S_1, S_2, S_3, S_{11}, S_{12} and

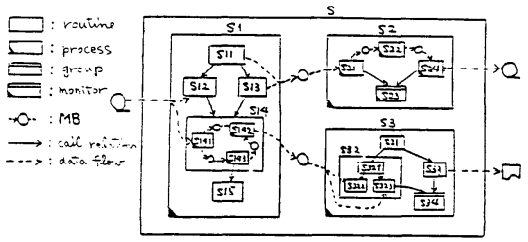


Fig. 7 Components hierarchy in SDL.

other have their own description units.

3.2 Component Description

The design description in SDL consists of a number of statements, each with a statement header word beginning with "$" character. A component description unit is enclosed by a header statement with header word denoting its component type (i.e., $PROCESS, $ROUTINE etc.) and an end statement, i.e., $END. A header statement contains a *unit name* (unique in the whole system) for managing the description unit, and *component name* (unique in a set of components of a decomposition) for representing its function. In a large system developed by many designers and programmers, these two names are necessary to identify a component from managerial and functional viewpoints.

The component description body is divided into the following two divisions:

* *External specification division*, consisting of discriptions necessary for the use of the component, e.g., function description, parameter specification, input and output conditions etc.
* *Internal specification division*, consisting of its internal structure descriptions necessary to implement the component, e.g., its internal (child) components declarations, the connection method among the internal components etc.

Figure 8 illustrates a skeleton of a process component description. Table 1 shows an SDL statements list classified in the above two divisions. In Table 1, the statements with asterisk mark (*) are composed by narrative descriptions, with or without reserved words, such as IN (input), OUT (output) and ABT (abort) words of $COND (condition) statement, to indicate their contents. In SDL, since in the narrative, informal descriptions are classified according to their contents or

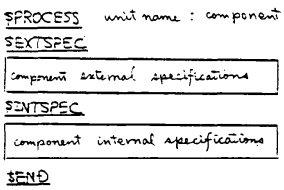


Fig. 8 Process component description skeleton.

Table 1 SDL statements list.

Statement Header	Statement Contents
External Specifications	
$ FUNCTION*	component function description.
$ KEY	key words and key phrases.
$ PARM	parameter specification (in/out, type, meaning etc.).
$ OPER	operation specification (O/V/OV type, operation parameter spec, function, effect description etc.).
$ SYSPARM	system integration parameter declaration (e.g., table size).
$ SYSENV	system environment (interfaces between system and operating system or hardware) declaration.
$ COND*	input output condition description.
$ EXAMPLE*	example descriptions for usage or dynamic behavior.
$ PERFORMANCE*	performance specification.
Internal Specifications	
$ CONV*	convention declaration (naming rule, message form etc.).
$ SYNONYM	synonym difinition.
$ TYPE	data type definition (similar to PASCAL).
$ VAR	common variable declaration.
$ MB	message buffer declaration.
$ FILE	file declaration.
$ ICOMP	internal (child) component declaration.
$ CONNECT	inter-components relation definition.
$ ASSERT*	general condition to be satisfied in all internal components.
$ ALGORITHM*	outline algorithm description.

(*: Statement contents are informal narrative description)

```
54-10-17  16:27:53        S D R S - 6  (V : DE1.0)          PID: 81970M
      1   $ROUTINE
      2       DEUS2     : UNIT-SUMMARY-DATA-GEN
      3   $EXTSPEC
      4   $FUN
      5       /* FIND DESIGN DATA BASE AND GENERATE UNIT SUMMARY DATA
      6          INTO TOKEN FILE. THIS COMPONENT IS EXECUTED IN BATCH-EMU. */
      7   $KEY
      8       'DEMO', 'SDL', 'UNIT SUMMARY', 'USUM', 'BATCH', 'DATA BASE'
      9   $COND
     10       IN /* PARM-FILE AND UNIT-NAME-FILE MUST BE ATTACHED */;
     11       OUT /* MESSAGE-FILE AND TOKEN-FILE IS GENERATED */;
     12       BRK /* IF UNIT IS NOT FOUND, NO TOKENS ARE GENERATED */
     13   $EXAMPLE
     14       /*      IF UM1:=NAME-FILE-UM1,UM2,UM3 AND PARM=NOFUM,
     15                    TOKEN=FILE>
     16       <UM1-BEGIN><ROUT><UM1><COM1>...<FUNB><COMD><PERF><END>...<END>
     17       <UM2-BEGIN><GROU><UM2><END>...<FUNB><COMD><PERF><END>...<END>
     18       <UM3-BEGIN><ROUT><UM3><END>...<FUNB><COMD><PERF><END>...<END>
     19                 <EOF>
     20            WHERE UM1 AND UM3 ARE ROUTINE, AND UM2 IS GROUP      */
     21   $PERFORMANCE
     22       MEMORY /* LESS THAN 64K WORDS */;
     23       TIME /* LESS THAN 30 SEC/UNIT */
     24   $INTSPEC
     25   $ICOMP
     26       ROUTINE DEUS21:   UNIT-SUMMARY-BATCH-MAIN, DEUS22   UNIT-HEAD-AND-ID-INF-GEN,
     27          DEUS23   UNIT-FUNCTION-GEN, DEUS24   UNIT-INT-STRUCTURE-GEN, DEUS25
     28          CALL-STRUCTURE-GEN
     29   $MAIN
     30       DEUS21
     31   $CONNECT
     32       DEUS21   CALL DEUS22;
     33       DEUS21   CALL DEUS23 IF /* FUN=YES */;
     34       DEUS21   CALL DEUS24 IF /* INT=YES */;
     35       DEUS21   CALL DEUS25 IF /* CALL=YES */;
     36       DEUS22   CALL DED81 / U;
     37       DEUS22   CALL DED83 / U;
     38       DEUS22   CALL DEUSTKNS / 0;
     39       DEUS23   CALL DED81 / U;
     40       DEUS23   CALL DEUSTKNS / 0;
     41       DEUS24   CALL DED81 / U;
     42       DEUS24   CALL DED82 / U;
     43       DEUS24   CALL DEUSTKNS / 0;
     44       DEUS25   CALL DED81 / U;
     45       DEUS25   CALL DED82 / U;
     46       DEUS25   CALL DEUSTKNS / 0;
     47       DEUS21   RD   PARM-FILE;
     48       DEUS21   RD   UNIT-NAME-FILE;
     49       DEUS21   WT   MESSAGE-FILE
     50   $ALGORITHM
     51       /*      OPEN FILES; INPUT PARM-FILE;
     52            WHILE: UNIT-NAME FILE NOT EOF
     53               INPUT UNIT NAME FILE( UN );
     54               GENERATE UNIT SUMMARY DATA FOR UN
     55            END;
     56            CLOSE FILES          */
     57   $END
     58
```

Fig. 9 SDL component description example (listed by SDL processor output command).

meanings, the language itself can be used as a check list for design documentation. The processor can provide the automatic design document generators classified by contents, e.g., performance specification generator in a set of components. Figure 9 shows a component description example in SDL.

3.3 Inter-component Relation Description

One of the most important features of SDL appears in inter-component relations description in a $CONNECT statement. By the methodology shown in Sec. 2, the relations between a component C and its environments are determined for a time when C is recognized as a component of its parent, that is, when internal structure of C's parent is designed. For instance, consider component S_{12} in Fig. 7. It is called by S_{11}, reads input file and calls S_{14}. Such relations between S_{12} and its environment are determined when S_{12}'s parent S_1 is designed as a control flow decomposition. In SDL, these relations (except parameter specifications) are described in S_1 internal specification, not in S_{12} external specification. In conventional design documents, these relations are described as S_{12}'s interface information in S_{12}'s own description unit. However, such interface information can be automatically selected from its parent description, and such conventional design documents can be automatically generated by SDL processor. This enables minimizing the amount of design descriptions which must be entered into the design database.

There are two kinds of interrelations among components:

(1) Routine and operation call relations.
(2) Data and message buffer access relations.

These relations are uniformly represented by a series of statements with the following syntax form;

⟨subject⟩ ⟨verb⟩ ⟨object 1⟩ [TO ⟨object 2⟩]
[IF ⟨condition description⟩],

where [—] denotes the optional clause. Statements are delimited by semicolon (;). ⟨Subject⟩ is a unit name of an internal component. ⟨Verb⟩ denotes the relation category, such as CALL (routine or operation call), SND/RCV (send/receive message buffer), RD/WT (read/write file), CPY (copy data value), and so on. ⟨Object 1⟩ and ⟨Object 2⟩ represent components or data which are related to ⟨subject⟩ in ⟨verb⟩ relation. The "TO ⟨object 2⟩" clause is necessary for CPY verb.

If the CALL verb object is group or monitor operation, a set of operations of a group or monior called by the subject component can be represented, with the group or monitor component unit name, as follows:

U1 CALL G1/(OP1, OP2, OP3).

This denotes that U1 calls operations OP1, OP2 and OP3 of group or monitor G1. For data object, "/" can be used to restrict the part of the data area to be accessed, as follows:

U1 REF D1/PART1.

This means that U1 refers only to PART1 in D1. Also, to represent the dynamic accessing methods for file and and message buffer, the restriction in data contents to be processed can be described as value class description. For instance,

U1 WT F1 {page header};
U2 WT F1 {page body}

says that U1 and U2 write page header part and page body part of file F1, respectively. The IF clause in the syntax form can be used to describe the condition for the verb, such as:

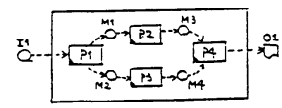
U1 CALL U99 IF {database overflow occurs}.

Figure 10 illustrates examples of inter-component relation description in SDL.

These inter-component relationships represent the static configuration of a system. To understand the system structure more completely, the dynamic configuration information may also be necessary in some cases. For example, when a number of routines are called and a number of data are accessed by a component, the execution order for the routine calls and the data accesses in the component may become dynamic configuration information, which is sometimes necessary to understand the system structure in more detail. A program list can be considered as a complete description of such dynamic behavior.

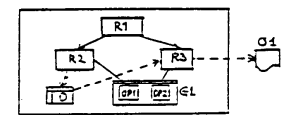
However, these dynamic behavior descriptions are excluded from $CONNECT statement description, for two reasons. They are:

(a) The dynamic behavior in most cases can be easily presumed from the component function and inter-component relationship descriptions.




```
$CONNECT
  P1 RD I1; P4 WT O1;
  P1 SND M1, M2;
  P2 RCV M1; P2 SND M3;
  P3 RCV M2; P3 SND M4;
  P4 RCV M3, M4
```

(a) $ CONNECT statemen texample for data flow.




```
$CONNECT
  R1 CALL R2 IF {---};
  R1 CALL R3 IF {---};
  R2 CALL G1/OP1;
  R3 CALL G1/OP2;
  R2 SET D; R3 REF D;
  R3 WT O1
```

(b) $ CONNECT statementexample for control flow.

Fig. 10 Inter-component relationship.

(b) The detailed dynamic behavior in most cases should be concerned in the programming stage, not in the design stage.

The value class description and if clause in $CONNECT statement, expressed before, are considered as compensations for the lack of dynamic behavior descriptions. More detailed dynamic behavior is described in $ALGORITHM statement, if necessary, in narrative form.

3.4 Value History Description

It is very important to grasp the meaning of data contents which will be processed, in order to understand the component function and its processing manner. For file or message buffer, the contents of each data element (data for one record in the file) are insufficient, but the contents of a whole data stream is rather important, to understand the meaning of data.

In SDL, a *value history* concept is introduced to represent the data stream contents produced and consumed by components. The value history is a description of data streams, which are possibly produced or can be consumed by components, in regular expression or BNF expression form, like formal language grammar. The value history for one file or message buffer is divided into two categories; the description from the producer's viewpoint and the description from the consumer's viewpoint.

For example, message buffer M1 is declared with value history descriptions, as follows;

```
$MB M1: CHAR
  VHP {M1=line* eof, line=char+ eol, char=
       nonb | blk}
  VHC {M1=(word | blks)* eof, word=nonb+,
       blks=(blk | eol)+},
```

where, "CHAR" denotes M1's element data type, and descriptions enclosed by { } followed by VHP and VHC represent value histories from a producer's viewpoint and from a consumer's viewpoint, respectively. When VHP and VHC descriptions are not distinguished, VH is used to identify the value history.

4. Design Language Processor

A design language processor facilitates entering design descriptions into design database, analyzing overall design consistency and acquiring a variety of design information from the database.

The processor consists of the following four parts:

(1) Design database: maintaining the design descriptions with relational form.
(2) SDL editor: facilitating entering design descriptions into the database, modifying and deleting the database contents.
(3) SDL analyzer: analyzing overall design consistency and completeness on the design database.

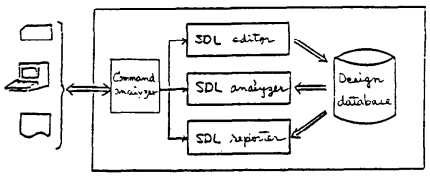


Fig. 11 Design language processor organization.

(4) SDL reporter: reporting a variety of design documents, and retrieving bits of design information from the design database.

The processor organization is shown in Fig. 11.

Although SDL includes many informal narrative statements, these statements could be represented with unified and formal description, depending upon the problem areas or projects. Therefore, the processor is constructed as expandable so that analyzers for newly introduced formal description can be easily added. In this section, only fundamental processor functions are described.

4.1 SDL Editor

The SDL editor analyzes syntax and semantics of design descriptions written in SDL, then enters the descriptions, if no errors exist, into the design database. It also facilitates updating and deleting the design database contents.

These editing actions can be accomplished for the following units:

(1) Component: identifying a unit name (or a set of unit names) for a component (components), entry, replacement, deletion or print of the component description will be performed.
(2) Statement: identifying a unit name (or a set of unit names) for a component (components) and a statement header word (or a set of statement header words), entry, replacement, deletion or print of the statement descriptions will be performed.
(3) Data: identifying a data name or a set of data names and the name scope, if the data name is not unique, the data definition will be modified. This case differs from (2) in that the modification object is indicated only by the data name without identifying the data defining component.

So, unlike usual text editors, SDL editor makes it possible to edit according to the language syntax.

Design information is maintained in a form with inter component relations in the design database. Thus, a modification in a part of the design automatically causes modifications in the related items. For instance, deleting a component from the database causes an effect to delete the component from the internal component declaration statement in its parent component. This assures overall design consistency among components.

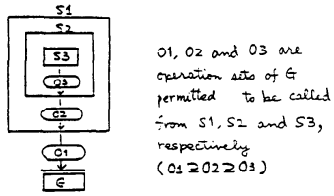


Fig. 12 Restriction for inter-component relationship.

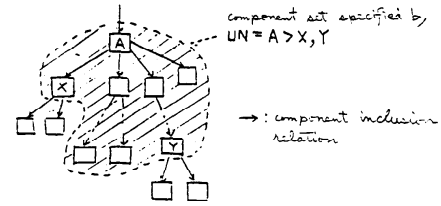


Fig. 13 Components set representation.

4.2 SDL Analyzer

The SDL analyzer mainly analyzes consistency and completeness of inter component relationships. Usually, errors in inter component relationships are more difficult to detect and correct, and require more effort, in a later development phase. The analyzer detects these errors earlier in validating consistency of the design information in the design database.

As an example, validating checks on access right of component to its external resources are illustrated. Assume that, as shown in Fig. 12, component S_1 includes component S_2, S_2 includes S_3, and operations of external group G are called from S_1, S_2 and S_3. Since S_2 and S_3 can not access beyond the rights permitted for S_1 and S_2, respectively, for operations sets O_1, O_2 and O_3 of G that are accessible from S1, S_2 and S_3, respectively, the relationship

$$O_1 \supseteq O_2 \supseteq O_3$$

should be held. For data accessing, similar access right is checked.

4.3 SDL Reporter

Although design information is mainly entered in a unit of a component or a statement into the design database using SDL editor, design results retrieval and output of a variety of design documents are mainly performed among more than one component using SDL reporter.

To restrict the component domain to be operated by the reporter, a set of components is specified by combining component inclusion relationship and many kinds of component conditions. For example, the following conditions:

UN = A > X, Y · · · inclusion relation specification by unit name.
CT = ROUTINE · · · component type identification.
MID = SHIGO · · · designer identification.
KEY = "SDL", "DATAFLOW" · · · key words identification.

designate a set of routine components that are successors of component A, except successors of X and Y (hashed area in Fig. 13), whose designer is "SHIGO," and for which "SDL" and "DATAFLOW" are described as key words.

For the components designated above, the following documents can be reported:

* Component summary.
* Detailed design reports, including related external information.
* A list of component interfaces.
* A list of performance specifications.

Also, concerning inter-component relationship information, the following documents can be reported:

* System organization diagram showing component inclusion relationships.
* Module call hierarchy diagram and data flow diagram showing module interconnection relationships.
* A list of data access cross references.
* A list of routine and operation call cross references.

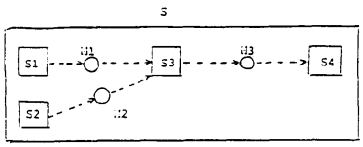
In addition to these batch outputs, many kinds of functions to retrieve bits of design information are provided. Moreover, since the results of these functions are usable as command parameters of SDL editor and SDL reporter, elaborated design supports, combining report generating functions and editing functions, may be provided.

5. Data Flow Mechanisms in FORTRAN and COBOL

To realize a system designed on the preseribed methodology as conventional sequential programs, how to realize inter processes connection with message buffers is a problem. For this, the authors have developed a message buffer simulating mechanism so that message buffers are easily used in conventional FORTRAN and COBOL [14]. Here, taking FORTRAN as an example, its function is briefly described.

A component S is assumed, as in Fig. 14(a), to be realized by a data flow decomposition connecting inter processes S_1, S_2, S_3 and S_4 by message buffers M_1, M_2 and M_3. In this instance, it makes no difference whether S is a process or a routine, and whether S_1, S_2, S_3 and S_4 are primitives or nonprimitives. For this case, the aspect of inner components connection in S should be declared as shown in Fig. 14(b). Processing the declaration statements generates a DF (data flow) table for S. Combining this DF table and a simple scheduler program (about two hundred codes in an assembly language) with object programs of processes S_1, S_2, S_3 and S_4, an executable program of S is obtained (Fig. 15).

Operations on message buffers will be accomplished by CALL statements in FORTRAN. For instance, to send data into M_1, write



(a) Data flow decomposition.

```
DFBEGIN  S              ·····declare component name,
PROC.C   4              ·····declare number of processes,
PROC     1,S1          ┐
PROC     2,S2          ├····· ··declare process code and
PROC     3,S3          ┤              process name,
PROC     4,S4          ┘
MB.C     3              ·····declare number of message buffers,
MB       :i1,1,3,d1,q1 ┐·····declare name, producer code, consumer
MB       :i2,2,3,d2,q2 ┤              code, data size(di) and queue
MB       :i3,3,4,d3,q3 ┘              capacity(qi) of message buffer.
DFEND
```

(b) Declarations for (a).

Fig. 14 Data flow combination declaration.

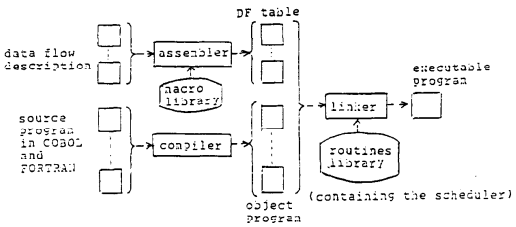


Fig. 15 System implementation processes.

CALL MPUT (M_1, d)

and, to receive a data from M_1 into x, write

CALL MGET (M_1, x, isend).

here, "isend" is a variable to be noted as the end of a data stream.

In executing these operations, the scheduler program controls data passing timing among processes, using the DF table for S. For instance, when executing "CALL MPUT (M_1, d)" in S_1, if the buffer M_1 is full, then the address of the successor statement of the CALL statement (i.e. reexecution address) together with the status wait for S_1, are written into DF table for S and the control is transfered to execute a ready process, say S_3. Also, in executing "CALL MGET (M_1, x, isend)" in S_3, S_1 is set to ready status.

Assigning a DF table as described for each component to realize with a data flow, local execution management is possible. Hence, a hierarchically constructed system with a combination of control flows and data flows, as noted in the design methodology in Sec. 2.2, is easily realizable in FORTRAN and COBOL.

6. Conclusion

A new modularization design methodology, which unifies data flow method based on concepts of concurrent processes or message buffers and control flow method based on traditional caller-callee relationship, a design language and its processor to enforce the methodology, and a simple mechanism to write programs in a natural way for the design result, are described.

Although intermediate data decision criteria in the data flow decomposition and a design method in use of non-sequential bulk data, such as database, are left to further research, the methodology including consistently a variety of existing design methodologies (e.g., Jackson method [7], Warnier method [15], Parnas's modularization [11], data flow decompositions by Morrison [10], etc.) and systematically integrating them has a wide range applicability. Also, due to the new methodology, in the process of hierarchical components decomposition, one can select in best suited scene for each methodology and utilize that methodology that is most effective.

Using the design language and its processor oriented to the design methodology, designers can enter only the minimum necessary information into the design database, and from it analyze the developing target system from many viewpoints. The design system has been implemented on NEC's operating system ACOS-6 in time sharing mode to be used interactively. Its effectiveness will be further evaluated through application to actual software developments.

The unification technique for data flow and control flow described in this paper will not only be applicable to software developments, but will also act as a guide line in new computer architecture research to utilize advantageous features of both traditional sequential machines and topical data flow machines, with attention to their duality.

Acknowledgments

The authors are grateful to Takashi Nishimura and their colleagues for their assistance in developing this research. In particular, concerning the representation of inter modules connection and the design of the programming tool, Mishimura' ideas are reflected. Thanks are also due to Koichi Nezu and Kiichi Fujino of NEC for their helpful advice and encouragement during this work.

References

1. Dennis, J. B. First version of a data flow procedure language, *Lecture Notes in Computer Science*, 19 (1974) 362–476.
2. Teichroew, D. and Herskey I, E. A. PSL/PSA: A computer-aided techniques for structured documentation and analysis of information processing systems, *IEEE Trans. on Software Engineering*, SE-3, 1 (1977) 16–33.
3. Bell, T. E. et al. An extendable approach to computer-aided software requirements engineering, *ibid.*, SE-3, 1 (1977) 49–60.
4. Uchida, Y. Software development support system SSD, *Software Engineering Notes of IPSJ*, 5–2 (1978) (in Japanese).
5. Iwamoto, K., Fujibayashi, S., Shigo, O. and Nishimura, T. SDMS: Software development and maintenance support system, *20th Annual Conf. of IPSJ*, 327–328 (1979) (in Japanese).
6. Shigo, O., Fujibayashi, S. and Iwamoto, K. On a modularization using a concept of concurrent processes, *16th Annual Conf. of IPSJ*, 573–574 (1975) (in Japanese).
7. Jackson, M. A. Principles of program design, Academic Press (1975).
8. Hansen, B. Operating system principles, Prentice Hall (1973).

9. BALZER, M. PORTS-a method for dynamic interprogram communication and job control, *AFIPS Conf. Proceedings* **38** (SJCC) (1971) 485–489.
10. MORRISON, J. P. Data stream linkage mechanism, *IBM System J.*, **17**, 4 (1978) 383–408.
11. PARNAS, D. L. On the criteria to be used in decomposing systems into modules, *Commun. of ACM*, **15**, 12 (1972) 1053–1058.
12. SHIGO, O., SHIMOMURA, T., IWAMOTO, K. and MAEJIMA, T. Implementing the abstraction technique in software development, *2nd USA-Japan Computer Conf.*, (1975) 517–522.
13. SHIGO, O. and IWAMOTO, K. On deadlock of a restricted concurrent processing system, *Annual Conf. of IECE of Japan* (Information Processing Area), (1977) 68 (in Japanese).
14. SHIGO, O. and NISHIMURA, T. Simulating message buffer in FORTRAN, *19th Annual Conf. of IPSJ* (1978) 263–264 (in Japanese).
15. WARNIER, J. D. and FLANAGAN, B. M. Entrainement a la programmation, Les Editions d'Organisation, Paris (1971).