

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

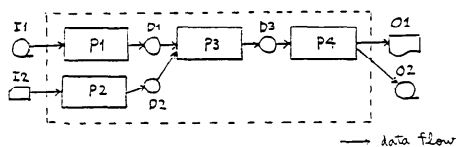


Fig. 1 Data flow modularization.

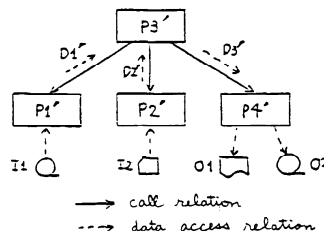


Fig. 2 Transformation of figure 1 to control flow modularization.

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 stream 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

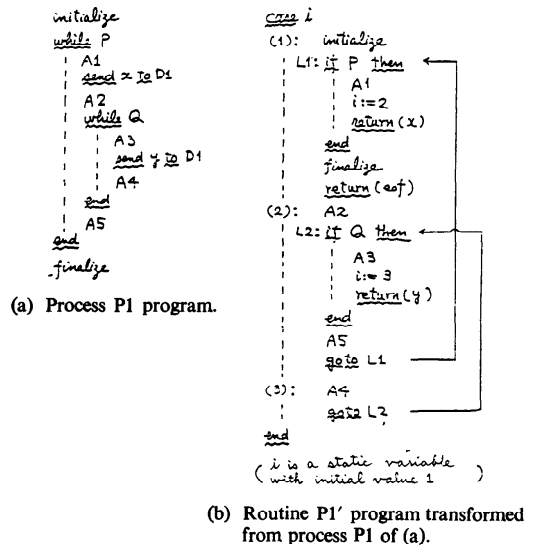
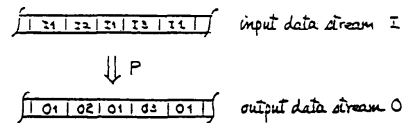
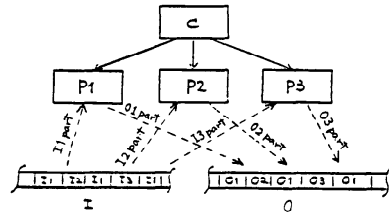


Fig. 3 Process/routine transformation.



(a) Input and output data structures matching. (I_i, O_j indicate data categories).



(b) Control flow realization for (a).

Fig. 4 Control flow modularization.

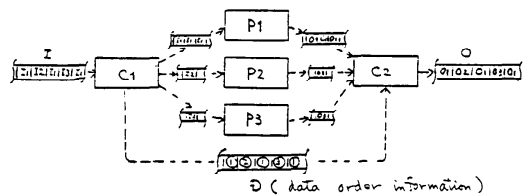


Fig. 5 Transformation of figure 4 to data flow modularization.

*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]).

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.

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 P₁ with respect to the semantic structures of I and D (control flow decomposition), (4) shows a refinement of P₁₁ concerned with the more detailed data structures (control flow decomposition) and (5) shows a refinement of P₁₂

introducing intermediate data IM₁ and IM₂ (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 C₁ and C₁₁ are simple main routines for P₁ and P₁₁, 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 P₁₂ 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-

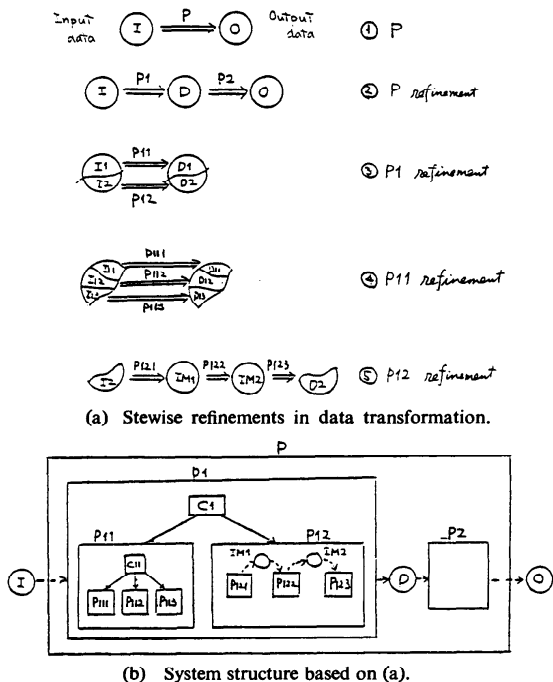


Fig. 6 Integration of data flow and control flow.

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 decomposition. In SDL, each component is described as a design description unit. A component represents a part of a 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

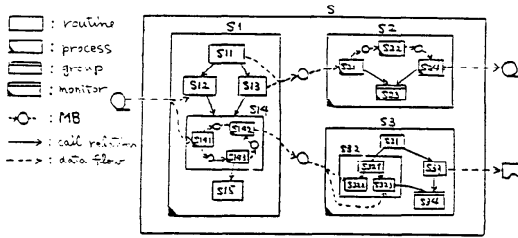


Fig. 7 Components hierarchy in SDL.

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 descriptions 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

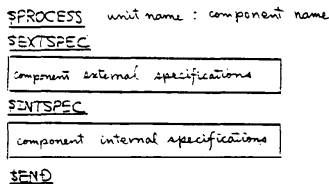


Fig. 8 Process component description skeleton.

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 definition.
\$ 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 SDRS-6 (U: DEL0) PID: 819700
1 $ROUTINE UNIT-SUMMARY-DATA-GEN
2 $EXTSPEC
3 $FUN
4 /* FIND DESIGN DATA BASE AND GENERATE UNIT SUMMARY DATA
5 INTO TOKEN FILE. THIS COMPONENT IS EXECUTED IN BATCH-MOD. */
6 $KEY "DBASE", "SDL", "UNIT-SUMMARY", "USUM", "BATCH", "DATA BASE"
7 $COND "DBASE", "SDL", "UNIT-SUMMARY" FILE MUST BE ATTACHED */
8 IN /* PARM-FILE AND UNIT-NAME-FILE IS GENERATED */
9 OUT /* MESSAGE-FILE AND TOKEN-FILE IS GENERATED */
10 $MB /* IF UNIT IS NOT FOUND, NO TOKENS ARE GENERATED */
11 $EXAMPLE
12 /* IF UNIT-NAME-FILE, UNIT-NAME, UNIT-NAME AND PARM-FILE
13 TOKEN-FILE
14 UNIT-BEGIN(COMP) UNIT-END ... (FUNB) (COND) (OPER) (END) ... (END)
15 UNIT-BEGIN(COMP) UNIT-END ... (FUNB) (COND) (OPER) (END) ... (END)
16 UNIT-BEGIN(COMP) UNIT-END ... (FUNB) (COND) (OPER) (END) ... (END)
17 UNIT-BEGIN(COMP) UNIT-END ... (FUNB) (COND) (OPER) (END) ... (END)
18 (END)
19 WHERE UNIT AND UNB ARE ROUTINE, AND UNB IS GROUP */
20 $PERFORMANCE
21 MEMORY /* LESS THAN 64K WORDS */
22 TIME /* LESS THAN 30 SEC/UNIT */
23 $INTSPEC
24 $ICOMP
25 $CONNECT
26 ROUTINE DEUS21 UNIT-SUMMARY-BATCH-MAIN, DEUS22 UNIT-HEAD-AND-ID-INFO-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=VES */;
34 DEUS21 CALL DEUS24 IF /* INT=VES */;
35 DEUS21 CALL DEUS25 IF /* CALL=VES */;
36 DEUS22 CALL DEUS26 / U;
37 DEUS22 CALL DEUS27 / U;
38 DEUS22 CALL DEUS28 / O;
39 DEUS23 CALL DEUS29 / U;
40 DEUS23 CALL DEUS30 / O;
41 DEUS24 CALL DEUS31 / U;
42 DEUS24 CALL DEUS32 / O;
43 DEUS25 CALL DEUS33 / U;
44 DEUS25 CALL DEUS34 / O;
45 DEUS26 CALL DEUS35 / O;
46 DEUS27 CALL DEUS36 / O;
47 DEUS28 CALL DEUS37 / O;
48 DEUS29 CALL DEUS38 / O;
49 DEUS30 CALL DEUS39 / O;
50 DEUS31 CALL DEUS40 / O;
51 DEUS32 CALL DEUS41 / O;
52 DEUS33 CALL DEUS42 / O;
53 DEUS34 CALL DEUS43 / O;
54 DEUS35 CALL DEUS44 / O;
55 DEUS36 CALL DEUS45 / O;
56 DEUS37 CALL DEUS46 / O;
57 DEUS38 CALL DEUS47 / O;
58 DEUS39 CALL DEUS48 / O;
59 DEUS40 CALL DEUS49 / O;
60 DEUS41 CALL DEUS50 / O;
61 DEUS42 CALL DEUS51 / O;
62 DEUS43 CALL DEUS52 / O;
63 DEUS44 CALL DEUS53 / O;
64 DEUS45 CALL DEUS54 / O;
65 DEUS46 CALL DEUS55 / O;
66 DEUS47 CALL DEUS56 / O;
67 DEUS48 CALL DEUS57 / O;
68 DEUS49 CALL DEUS58 / O;
69 DEUS50 CALL DEUS59 / O;
70 DEUS51 CALL DEUS60 / O;
71 DEUS52 CALL DEUS61 / O;
72 DEUS53 CALL DEUS62 / O;
73 DEUS54 CALL DEUS63 / O;
74 DEUS55 CALL DEUS64 / O;
75 DEUS56 CALL DEUS65 / O;
76 DEUS57 CALL DEUS66 / O;
77 DEUS58 CALL DEUS67 / O;
78 DEUS59 CALL DEUS68 / O;
79 DEUS60 CALL DEUS69 / O;
80 DEUS61 CALL DEUS70 / O;
81 DEUS62 CALL DEUS71 / O;
82 DEUS63 CALL DEUS72 / O;
83 DEUS64 CALL DEUS73 / O;
84 DEUS65 CALL DEUS74 / O;
85 DEUS66 CALL DEUS75 / O;
86 DEUS67 CALL DEUS76 / O;
87 DEUS68 CALL DEUS77 / O;
88 DEUS69 CALL DEUS78 / O;
89 DEUS70 CALL DEUS79 / O;
90 DEUS71 CALL DEUS80 / O;
91 DEUS72 CALL DEUS81 / O;
92 DEUS73 CALL DEUS82 / O;
93 DEUS74 CALL DEUS83 / O;
94 DEUS75 CALL DEUS84 / O;
95 DEUS76 CALL DEUS85 / O;
96 DEUS77 CALL DEUS86 / O;
97 DEUS78 CALL DEUS87 / O;
98 DEUS79 CALL DEUS88 / O;
99 DEUS80 CALL DEUS89 / O;
100 DEUS81 CALL DEUS90 / O;
101 DEUS82 CALL DEUS91 / O;
102 DEUS83 CALL DEUS92 / O;
103 DEUS84 CALL DEUS93 / O;
104 DEUS85 CALL DEUS94 / O;
105 DEUS86 CALL DEUS95 / O;
106 DEUS87 CALL DEUS96 / O;
107 DEUS88 CALL DEUS97 / O;
108 DEUS89 CALL DEUS98 / O;
109 DEUS90 CALL DEUS99 / O;
110 DEUS91 CALL DEUS100 / O;
111 DEUS92 CALL DEUS101 / O;
112 DEUS93 CALL DEUS102 / O;
113 DEUS94 CALL DEUS103 / O;
114 DEUS95 CALL DEUS104 / O;
115 DEUS96 CALL DEUS105 / O;
116 DEUS97 CALL DEUS106 / O;
117 DEUS98 CALL DEUS107 / O;
118 DEUS99 CALL DEUS108 / O;
119 DEUS100 CALL DEUS109 / O;
120 DEUS101 CALL DEUS110 / O;
121 DEUS102 CALL DEUS111 / O;
122 DEUS103 CALL DEUS112 / O;
123 DEUS104 CALL DEUS113 / O;
124 DEUS105 CALL DEUS114 / O;
125 DEUS106 CALL DEUS115 / O;
126 DEUS107 CALL DEUS116 / O;
127 DEUS108 CALL DEUS117 / O;
128 DEUS109 CALL DEUS118 / O;
129 DEUS110 CALL DEUS119 / O;
130 DEUS111 CALL DEUS120 / O;
131 DEUS112 CALL DEUS121 / O;
132 DEUS113 CALL DEUS122 / O;
133 DEUS114 CALL DEUS123 / O;
134 DEUS115 CALL DEUS124 / O;
135 DEUS116 CALL DEUS125 / O;
136 DEUS117 CALL DEUS126 / O;
137 DEUS118 CALL DEUS127 / O;
138 DEUS119 CALL DEUS128 / O;
139 DEUS120 CALL DEUS129 / O;
140 DEUS121 CALL DEUS130 / O;
141 DEUS122 CALL DEUS131 / O;
142 DEUS123 CALL DEUS132 / O;
143 DEUS124 CALL DEUS133 / O;
144 DEUS125 CALL DEUS134 / O;
145 DEUS126 CALL DEUS135 / O;
146 DEUS127 CALL DEUS136 / O;
147 DEUS128 CALL DEUS137 / O;
148 DEUS129 CALL DEUS138 / O;
149 DEUS130 CALL DEUS139 / O;
150 DEUS131 CALL DEUS140 / O;
151 DEUS132 CALL DEUS141 / O;
152 DEUS133 CALL DEUS142 / O;
153 DEUS134 CALL DEUS143 / O;
154 DEUS135 CALL DEUS144 / O;
155 DEUS136 CALL DEUS145 / O;
156 DEUS137 CALL DEUS146 / O;
157 DEUS138 CALL DEUS147 / O;
158 DEUS139 CALL DEUS148 / O;
159 DEUS140 CALL DEUS149 / O;
160 DEUS141 CALL DEUS150 / O;
161 DEUS142 CALL DEUS151 / O;
162 DEUS143 CALL DEUS152 / O;
163 DEUS144 CALL DEUS153 / O;
164 DEUS145 CALL DEUS154 / O;
165 DEUS146 CALL DEUS155 / O;
166 DEUS147 CALL DEUS156 / O;
167 DEUS148 CALL DEUS157 / O;
168 DEUS149 CALL DEUS158 / O;
169 DEUS150 CALL DEUS159 / O;
170 DEUS151 CALL DEUS160 / O;
171 DEUS152 CALL DEUS161 / O;
172 DEUS153 CALL DEUS162 / O;
173 DEUS154 CALL DEUS163 / O;
174 DEUS155 CALL DEUS164 / O;
175 DEUS156 CALL DEUS165 / O;
176 DEUS157 CALL DEUS166 / O;
177 DEUS158 CALL DEUS167 / O;
178 DEUS159 CALL DEUS168 / O;
179 DEUS160 CALL DEUS169 / O;
180 DEUS161 CALL DEUS170 / O;
181 DEUS162 CALL DEUS171 / O;
182 DEUS163 CALL DEUS172 / O;
183 DEUS164 CALL DEUS173 / O;
184 DEUS165 CALL DEUS174 / O;
185 DEUS166 CALL DEUS175 / O;
186 DEUS167 CALL DEUS176 / O;
187 DEUS168 CALL DEUS177 / O;
188 DEUS169 CALL DEUS178 / O;
189 DEUS170 CALL DEUS179 / O;
190 DEUS171 CALL DEUS180 / O;
191 DEUS172 CALL DEUS181 / O;
192 DEUS173 CALL DEUS182 / O;
193 DEUS174 CALL DEUS183 / O;
194 DEUS175 CALL DEUS184 / O;
195 DEUS176 CALL DEUS185 / O;
196 DEUS177 CALL DEUS186 / O;
197 DEUS178 CALL DEUS187 / O;
198 DEUS179 CALL DEUS188 / O;
199 DEUS180 CALL DEUS189 / O;
200 DEUS181 CALL DEUS190 / O;
201 DEUS182 CALL DEUS191 / O;
202 DEUS183 CALL DEUS192 / O;
203 DEUS184 CALL DEUS193 / O;
204 DEUS185 CALL DEUS194 / O;
205 DEUS186 CALL DEUS195 / O;
206 DEUS187 CALL DEUS196 / O;
207 DEUS188 CALL DEUS197 / O;
208 DEUS189 CALL DEUS198 / O;
209 DEUS190 CALL DEUS199 / O;
210 DEUS191 CALL DEUS200 / O;
211 DEUS192 CALL DEUS201 / O;
212 DEUS193 CALL DEUS202 / O;
213 DEUS194 CALL DEUS203 / O;
214 DEUS195 CALL DEUS204 / O;
215 DEUS196 CALL DEUS205 / O;
216 DEUS197 CALL DEUS206 / O;
217 DEUS198 CALL DEUS207 / O;
218 DEUS199 CALL DEUS208 / O;
219 DEUS200 CALL DEUS209 / O;
220 DEUS201 CALL DEUS210 / O;
221 DEUS202 CALL DEUS211 / O;
222 DEUS203 CALL DEUS212 / O;
223 DEUS204 CALL DEUS213 / O;
224 DEUS205 CALL DEUS214 / O;
225 DEUS206 CALL DEUS215 / O;
226 DEUS207 CALL DEUS216 / O;
227 DEUS208 CALL DEUS217 / O;
228 DEUS209 CALL DEUS218 / O;
229 DEUS210 CALL DEUS219 / O;
230 DEUS211 CALL DEUS220 / O;
231 DEUS212 CALL DEUS221 / O;
232 DEUS213 CALL DEUS222 / O;
233 DEUS214 CALL DEUS223 / O;
234 DEUS215 CALL DEUS224 / O;
235 DEUS216 CALL DEUS225 / O;
236 DEUS217 CALL DEUS226 / O;
237 DEUS218 CALL DEUS227 / O;
238 DEUS219 CALL DEUS228 / O;
239 DEUS220 CALL DEUS229 / O;
240 DEUS221 CALL DEUS230 / O;
241 DEUS222 CALL DEUS231 / O;
242 DEUS223 CALL DEUS232 / O;
243 DEUS224 CALL DEUS233 / O;
244 DEUS225 CALL DEUS234 / O;
245 DEUS226 CALL DEUS235 / O;
246 DEUS227 CALL DEUS236 / O;
247 DEUS228 CALL DEUS237 / O;
248 DEUS229 CALL DEUS238 / O;
249 DEUS230 CALL DEUS239 / O;
250 DEUS231 CALL DEUS240 / O;
251 DEUS232 CALL DEUS241 / O;
252 DEUS233 CALL DEUS242 / O;
253 DEUS234 CALL DEUS243 / O;
254 DEUS235 CALL DEUS244 / O;
255 DEUS236 CALL DEUS245 / O;
256 DEUS237 CALL DEUS246 / O;
257 DEUS238 CALL DEUS247 / O;
258 DEUS239 CALL DEUS248 / O;
259 DEUS240 CALL DEUS249 / O;
260 DEUS241 CALL DEUS250 / O;
261 DEUS242 CALL DEUS251 / O;
262 DEUS243 CALL DEUS252 / O;
263 DEUS244 CALL DEUS253 / O;
264 DEUS245 CALL DEUS254 / O;
265 DEUS246 CALL DEUS255 / O;
266 DEUS247 CALL DEUS256 / O;
267 DEUS248 CALL DEUS257 / O;
268 DEUS249 CALL DEUS258 / O;
269 DEUS250 CALL DEUS259 / O;
270 DEUS251 CALL DEUS260 / O;
271 DEUS252 CALL DEUS261 / O;
272 DEUS253 CALL DEUS262 / O;
273 DEUS254 CALL DEUS263 / O;
274 DEUS255 CALL DEUS264 / O;
275 DEUS256 CALL DEUS265 / O;
276 DEUS257 CALL DEUS266 / O;
277 DEUS258 CALL DEUS267 / O;
278 DEUS259 CALL DEUS268 / O;
279 DEUS260 CALL DEUS269 / O;
280 DEUS261 CALL DEUS270 / O;
281 DEUS262 CALL DEUS271 / O;
282 DEUS263 CALL DEUS272 / O;
283 DEUS264 CALL DEUS273 / O;
284 DEUS265 CALL DEUS274 / O;
285 DEUS266 CALL DEUS275 / O;
286 DEUS267 CALL DEUS276 / O;
287 DEUS268 CALL DEUS277 / O;
288 DEUS269 CALL DEUS278 / O;
289 DEUS270 CALL DEUS279 / O;
290 DEUS271 CALL DEUS280 / O;
291 DEUS272 CALL DEUS281 / O;
292 DEUS273 CALL DEUS282 / O;
293 DEUS274 CALL DEUS283 / O;
294 DEUS275 CALL DEUS284 / O;
295 DEUS276 CALL DEUS285 / O;
296 DEUS277 CALL DEUS286 / O;
297 DEUS278 CALL DEUS287 / O;
298 DEUS279 CALL DEUS288 / O;
299 DEUS280 CALL DEUS289 / O;
300 DEUS281 CALL DEUS290 / O;
301 DEUS282 CALL DEUS291 / O;
302 DEUS283 CALL DEUS292 / O;
303 DEUS284 CALL DEUS293 / O;
304 DEUS285 CALL DEUS294 / O;
305 DEUS286 CALL DEUS295 / O;
306 DEUS287 CALL DEUS296 / O;
307 DEUS288 CALL DEUS297 / O;
308 DEUS289 CALL DEUS298 / O;
309 DEUS290 CALL DEUS299 / O;
310 DEUS291 CALL DEUS300 / O;
311 DEUS292 CALL DEUS301 / O;
312 DEUS293 CALL DEUS302 / O;
313 DEUS294 CALL DEUS303 / O;
314 DEUS295 CALL DEUS304 / O;
315 DEUS296 CALL DEUS305 / O;
316 DEUS297 CALL DEUS306 / O;
317 DEUS298 CALL DEUS307 / O;
318 DEUS299 CALL DEUS308 / O;
319 DEUS300 CALL DEUS309 / O;
320 DEUS301 CALL DEUS310 / O;
321 DEUS302 CALL DEUS311 / O;
322 DEUS303 CALL DEUS312 / O;
323 DEUS304 CALL DEUS313 / O;
324 DEUS305 CALL DEUS314 / O;
325 DEUS306 CALL DEUS315 / O;
326 DEUS307 CALL DEUS316 / O;
327 DEUS308 CALL DEUS317 / O;
328 DEUS309 CALL DEUS318 / O;
329 DEUS310 CALL DEUS319 / O;
330 DEUS311 CALL DEUS320 / O;
331 DEUS312 CALL DEUS321 / O;
332 DEUS313 CALL DEUS322 / O;
333 DEUS314 CALL DEUS323 / O;
334 DEUS315 CALL DEUS324 / O;
335 DEUS316 CALL DEUS325 / O;
336 DEUS317 CALL DEUS326 / O;
337 DEUS318 CALL DEUS327 / O;
338 DEUS319 CALL DEUS328 / O;
339 DEUS320 CALL DEUS329 / O;
340 DEUS321 CALL DEUS330 / O;
341 DEUS322 CALL DEUS331 / O;
342 DEUS323 CALL DEUS332 / O;
343 DEUS324 CALL DEUS333 / O;
344 DEUS325 CALL DEUS334 / O;
345 DEUS326 CALL DEUS335 / O;
346 DEUS327 CALL DEUS336 / O;
347 DEUS328 CALL DEUS337 / O;
348 DEUS329 CALL DEUS338 / O;
349 DEUS330 CALL DEUS339 / O;
350 DEUS331 CALL DEUS340 / O;
351 DEUS332 CALL DEUS341 / O;
352 DEUS333 CALL DEUS342 / O;
353 DEUS334 CALL DEUS343 / O;
354 DEUS335 CALL DEUS344 / O;
355 DEUS336 CALL DEUS345 / O;
356 DEUS337 CALL DEUS346 / O;
357 DEUS338 CALL DEUS347 / O;
358 DEUS339 CALL DEUS348 / O;
359 DEUS340 CALL DEUS349 / O;
360 DEUS341 CALL DEUS350 / O;
361 DEUS342 CALL DEUS351 / O;
362 DEUS343 CALL DEUS352 / O;
363 DEUS344 CALL DEUS353 / O;
364 DEUS345 CALL DEUS354 / O;
365 DEUS346 CALL DEUS355 / O;
366 DEUS347 CALL DEUS356 / O;
367 DEUS348 CALL DEUS357 / O;
368 DEUS349 CALL DEUS358 / O;
369 DEUS350 CALL DEUS359 / O;
370 DEUS351 CALL DEUS360 / O;
371 DEUS352 CALL DEUS361 / O;
372 DEUS353 CALL DEUS362 / O;
373 DEUS354 CALL DEUS363 / O;
374 DEUS355 CALL DEUS364 / O;
375 DEUS356 CALL DEUS365 / O;
376 DEUS357 CALL DEUS366 / O;
377 DEUS358 CALL DEUS367 / O;
378 DEUS359 CALL DEUS368 / O;
379 DEUS360 CALL DEUS369 / O;
380 DEUS361 CALL DEUS370 / O;
381 DEUS362 CALL DEUS371 / O;
382 DEUS363 CALL DEUS372 / O;
383 DEUS364 CALL DEUS373 / O;
384 DEUS365 CALL DEUS374 / O;
385 DEUS366 CALL DEUS375 / O;
386 DEUS367 CALL DEUS376 / O;
387 DEUS368 CALL DEUS377 / O;
388 DEUS369 CALL DEUS378 / O;
389 DEUS370 CALL DEUS379 / O;
390 DEUS371 CALL DEUS380 / O;
391 DEUS372 CALL DEUS381 / O;
392 DEUS373 CALL DEUS382 / O;
393 DEUS374 CALL DEUS383 / O;
394 DEUS375 CALL DEUS384 / O;
395 DEUS376 CALL DEUS385 / O;
396 DEUS377 CALL DEUS386 / O;
397 DEUS378 CALL DEUS387 / O;
398 DEUS379 CALL DEUS388 / O;
399 DEUS380 CALL DEUS389 / O;
400 DEUS381 CALL DEUS390 / O;
401 DEUS382 CALL DEUS391 / O;
402 DEUS383 CALL DEUS392 / O;
403 DEUS384 CALL DEUS393 / O;
404 DEUS385 CALL DEUS394 / O;
405 DEUS386 CALL DEUS395 / O;
406 DEUS387 CALL DEUS396 / O;
407 DEUS388 CALL DEUS397 / O;
408 DEUS389 CALL DEUS398 / O;
409 DEUS390 CALL DEUS399 / O;
410 DEUS391 CALL DEUS400 / O;
411 DEUS392 CALL DEUS401 / O;
412 DEUS393 CALL DEUS402 / O;
413 DEUS394 CALL DEUS403 / O;
414 DEUS395 CALL DEUS404 / O;
415 DEUS396 CALL DEUS405 / O;
416 DEUS397 CALL DEUS406 / O;
417 DEUS398 CALL DEUS407 / O;
418 DEUS399 CALL DEUS408 / O;
419 DEUS400 CALL DEUS409 / O;
420 DEUS401 CALL DEUS410 / O;
421 DEUS402 CALL DEUS411 / O;
422 DEUS403 CALL DEUS412 / O;
423 DEUS404 CALL DEUS413 / O;
424 DEUS405 CALL DEUS414 / O;
425 DEUS406 CALL DEUS415 / O;
426 DEUS407 CALL DEUS416 / O;
427 DEUS408 CALL DEUS417 / O;
428 DEUS409 CALL DEUS418 / O;
429 DEUS410 CALL DEUS419 / O;
430 DEUS411 CALL DEUS420 / O;
431 DEUS412 CALL DEUS421 / O;
432 DEUS413 CALL DEUS422 / O;
433 DEUS414 CALL DEUS423 / O;
434 DEUS415 CALL DEUS424 / O;
435 DEUS416 CALL DEUS425 / O;
436 DEUS417 CALL DEUS426 / O;
437 DEUS418 CALL DEUS427 / O;
438 DEUS419 CALL DEUS428 / O;
439 DEUS420 CALL DEUS429 / O;
440 DEUS421 CALL DEUS430 / O;
441 DEUS422 CALL DEUS431 / O;
442 DEUS423 CALL DEUS432 / O;
443 DEUS424 CALL DEUS433 / O;
444 DEUS425 CALL DEUS434 / O;
445 DEUS426 CALL DEUS435 / O;
446 DEUS427 CALL DEUS436 / O;
447 DEUS428 CALL DEUS437 / O;
448 DEUS429 CALL DEUS438 / O;
449 DEUS430 CALL DEUS439 / O;
450 DEUS431 CALL DEUS440 / O;
451 DEUS432 CALL DEUS441 / O;
452 DEUS433 CALL DEUS442 / O;
453 DEUS434 CALL DEUS443 / O;
454 DEUS435 CALL DEUS444 / O;
455 DEUS436 CALL DEUS445 / O;
456 DEUS437 CALL DEUS446 / O;
457 DEUS438 CALL DEUS447 / O;
458 DEUS439 CALL DEUS448 / O;
459 DEUS440 CALL DEUS449 / O;
460 DEUS441 CALL DEUS450 / O;
461 DEUS442 CALL DEUS451 / O;
462 DEUS443 CALL DEUS452 / O;
463 DEUS444 CALL DEUS453 / O;
464 DEUS445 CALL DEUS454 / O;
465 DEUS446 CALL DEUS455 / O;
466 DEUS447 CALL DEUS456 / O;
467 DEUS448 CALL DEUS457 / O;
468 DEUS449 CALL DEUS458 / O;
469 DEUS450 CALL DEUS459 / O;
470 DEUS451 CALL DEUS460 / O;
471 DEUS452 CALL DEUS461 / O;
472 DEUS453 CALL DEUS462 / O;
473 DEUS454 CALL DEUS463 / O;
474 DEUS455 CALL DEUS464 / O;
475 DEUS456 CALL DEUS465 / O;
476 DEUS457 CALL DEUS466 / O;
477 DEUS458 CALL DEUS467 / O;
478 DEUS459 CALL DEUS468 / O;
479 DEUS460 CALL DEUS469 / O;
480 DEUS461 CALL DEUS470 / O;
481 DEUS462 CALL DEUS471 / O;
482 DEUS463 CALL DEUS472 / O;
483 DEUS464 CALL DEUS473 / O;
484 DEUS465 CALL DEUS474 / O;
485 DEUS466 CALL DEUS475 / O;
486 DEUS467 CALL DEUS476 / O;
487 DEUS468 CALL DEUS477 / O;
488 DEUS469 CALL DEUS478 / O;
489 DEUS470 CALL DEUS479 / O;
490 DEUS471 CALL DEUS480 / O;
491 DEUS472 CALL DEUS481 / O;
492 DEUS473 CALL DEUS482 / O;
493 DEUS474 CALL DEUS483 / O;
494 DEUS475 CALL DEUS484 / O;
495 DEUS476 CALL DEUS485 / O;
496 DEUS477 CALL DEUS486 / O;
497 DEUS478 CALL DEUS487 / O;
498 DEUS479 CALL DEUS488 / O;
499 DEUS480 CALL DEUS489 / O;
500 DEUS481 CALL DEUS490 / O;
501 DEUS482 CALL DEUS491 / O;
502 DEUS483 CALL DEUS492 / O;
503 DEUS484 CALL DEUS493 / O;
504 DEUS485 CALL DEUS494 / O;
505 DEUS486 CALL DEUS495 / O;
506 DEUS487 CALL DEUS496 / O;
507 DEUS488 CALL DEUS497 / O;
508 DEUS489 CALL DEUS498 / O;
509 DEUS490 CALL DEUS499 / O;
510 DEUS491 CALL DEUS500 / O;
511 DEUS492 CALL DEUS501 / O;
512 DEUS493 CALL DEUS502 / O;
513 DEUS494 CALL DEUS503 / O;
514 DEUS495 CALL DEUS504 / O;
515 DEUS496 CALL DEUS505 / O;
516 DEUS497 CALL DEUS506 / O;
517 DEUS498 CALL DEUS507 / O;
518 DEUS499 CALL DEUS508 / O;
519 DEUS500 CALL DEUS509 / O;
520 DEUS501 CALL DEUS510 / O;
521 DEUS502 CALL DEUS511 / O;
522 DEUS503 CALL DEUS512 / O;
523 DEUS504 CALL DEUS513 / O;
524 DEUS505 CALL DEUS514 / O;
525 DEUS506 CALL DEUS515 / O;
526 DEUS507 CALL DEUS516 / O;
527 DEUS508 CALL DEUS517 / O;
528 DEUS509 CALL DEUS518 / O;
529 DEUS510 CALL DEUS519 / O;
530 DEUS511 CALL DEUS520 / O;
531 DEUS512 CALL DEUS521 / O;
532 DEUS513 CALL DEUS522 / O;
533 DEUS514 CALL DEUS523 / O;
534 DEUS515 CALL DEUS524 / O;
535 DEUS516 CALL DEUS525 / O;
536 DEUS517 CALL DEUS526 / O;
537 DEUS518 CALL DEUS527 / O;
538 DEUS519 CALL DEUS528 / O;
539 DEUS520 CALL DEUS529 / O;
540 DEUS521 CALL DEUS530 / O;
541 DEUS522 CALL DEUS531 / O;
542 DEUS523 CALL DEUS532 / O;
543 DEUS524 CALL DEUS533 / O;
544 DEUS525 CALL DEUS534 / O;
545 DEUS526 CALL DEUS535 / O;
546 DEUS527 CALL DEUS536 / O;
547 DEUS528 CALL DEUS537 / O;
548 DEUS529 CALL DEUS538 / O;
549 DEUS530 CALL DEUS539 / O;
550 DEUS531 CALL DEUS540 / O;
551 DEUS532 CALL DEUS541 / O;
552 DEUS533 CALL DEUS542 / O;
553 DEUS534 CALL DEUS543 / O;
554 DEUS535 CALL DEUS544 / O;
555 DEUS536 CALL DEUS545 / O;
556 DEUS537 CALL DEUS546 / O;
557 DEUS538 CALL DEUS547 / O;
558 DEUS539 CALL DEUS548 / O;
559 DEUS540 CALL DEUS549 / O;
560 DEUS541 CALL DEUS550 / O;
561 DEUS542 CALL DEUS551 / O;
562 DEUS543 CALL DEUS552 / O;
563 DEUS544 CALL DEUS553 / O;
564 DEUS545 CALL DEUS554 / O;
565 DEUS546 CALL DEUS555 / O;
566 DEUS547 CALL DEUS556 / O;
567 DEUS548 CALL DEUS557 / O;
568 DEUS549 CALL DEUS558 / O;
569 DEUS550 CALL DEUS559 / O;
570 DEUS551 CALL DEUS560 / O;
571 DEUS552 CALL DEUS561 / O;
572 DEUS553 CALL DEUS562 / O;
573 DEUS554 CALL DEUS563 / O;
574 DEUS555 CALL DEUS564 / O;
575 DEUS556 CALL DEUS565 / O;
576 DEUS557 CALL DEUS566 / O;
577 DEUS558 CALL DEUS567 / O;
578 DEUS559 CALL DEUS568 / O;
579 DEUS560 CALL DEUS569 / O;
580 DEUS561 CALL DEUS570 / O;
581 DEUS562 CALL DEUS571 / O;
582 DEUS563 CALL DEUS572 / O;
583 DEUS564 CALL DEUS573 / O;
584 DEUS565 CALL DEUS574 / O;
585 DEUS566 CALL DEUS575 / O;
586 DEUS567 CALL DEUS576 / O;
587 DEUS568 CALL DEUS577 / O;
588 DEUS569 CALL DEUS578 / O;
589 DEUS570 CALL DEUS579 / O;
590 DEUS571 CALL DEUS580 / O;
591 DEUS572 CALL DEUS581 / O;
592 DEUS573 CALL DEUS582 / O;
593 DEUS574 CALL DEUS583 / O;
594 DEUS575 CALL DEUS584 / O;
595 DEUS576 CALL DEUS585 / O;
596 DEUS577 CALL DEUS586 / O;
597 DEUS578 CALL DEUS587 / O;
598 DEUS579 CALL DEUS588 / O;
599 DEUS580 CALL DEUS589 / O;
600 DEUS581 CALL DEUS590 / O;
601 DEUS582 CALL DEUS591 / O;
602 DEUS583 CALL DEUS592 / O;
603 DEUS584 CALL DEUS593 / O;
604 DEUS585 CALL DEUS594 / O;
605 DEUS586 CALL DEUS595 / O;
606 DEUS587 CALL DEUS596 / O;
607 DEUS588 CALL DEUS597 / O;
608 DEUS589 CALL DEUS598 /
```

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 S₁₂ in Fig. 7. It is called by S₁₁, reads input file and calls S₁₄. Such relations between S₁₂ and its environment are determined when S₁₂'s parent S₁ is designed as a control flow decomposition. In SDL, these relations (except parameter specifications) are described in S₁ internal specification, not in S₁₂ external specification. In conventional design documents, these relations are described as S₁₂'s interface information in S₁₂'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 monitor 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:

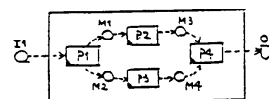
U1 CALL U99 IF {database overflow occurs}.

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.

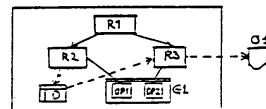
However, these dynamic behavior descriptions are excluded from \$CONNECT statement description, for two reasons. They are:

- (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 statement example 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 statement example 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;

```
$MBS 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.

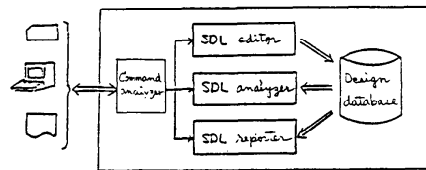


Fig. 11 Design language processor organization.

- (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.

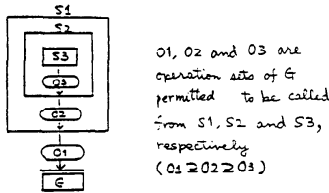


Fig. 12 Restriction for inter-component relationship.

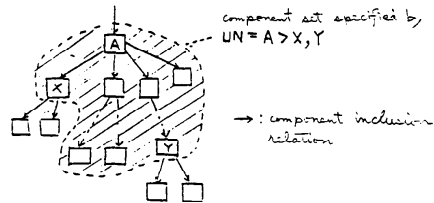


Fig. 13 Components set representation.

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 S_1 , 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 prescribed 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

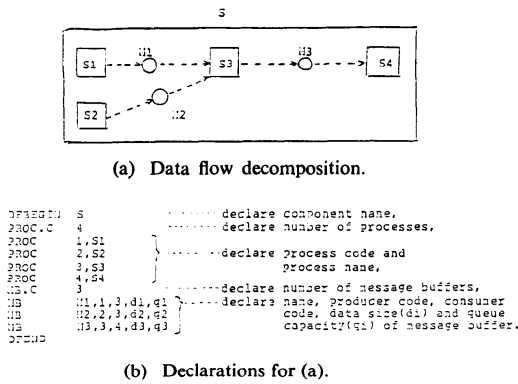
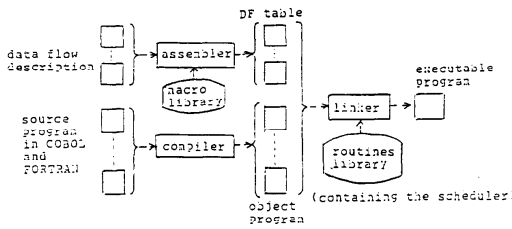


Fig. 14 Data flow combination declaration.



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 transferred 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 meth-

odology, 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's 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 HERSHEY 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).