Software Design Process: Chrysalis Stage under the Control of Designers

KIYOSHI ITOH*, KOICHI TABATA** and YUTAKA OHNO***

In order to introduce a well-formed mechanism, controlled by software designers into the traditional design stage, we have developed "Dual-View Integration Simulator (Duvis)" based on the "Dual-View Designing (DVD)" methodology for a demand-oriented software system such as an online software system or a database inquiry system. DVD enables designers to construct an executable and evaluatable prototype of a software system. DVD advances the design process by controlling the transformation of design from the users' view to the programmers' view. In DVD, a prototype is constructed in top-down fashion from the flow-oriented viewpoint (appropriate for users' view) at its early design stage. A prototype from the process-oriented viewpoint is obtained at the final stage. During the intermediate design stage, the stepwise substitution of flow-oriented components for process-oriented components in a prototype is taking place under the control of software designers. Duvis performs testing and evaluation of function and performance for an evolving prototype, gradually introducing the actual running environment in parallel with the design process. Final prototypes with process-oriented views are skeletons of programs to be implemented at the succeeding programming phase. Duvis has been implemented on an IBM/370-compatible computer (MVS/TSO).

1. Introduction

Prototyping has been recognized as one of the effective development methods in the requirements analysis phase. A prototype of a software system is operational or executable under an actual or simulated execution environment. Prototyping means that a prototype is constructed in order to correct ambiguities and misunderstandings as early as possible in the software life cycle and to decrease production costs. The merits of prototyping, for example, in [5], are: (1) to decrease the communication gap between the system developer and the user, by the user exercising the system as if it were operating in his own environment, (2) to maximize the user feedback as early as possible in the software life cycle before the requirements analysis is finalized, and (3) to obtain a vehicle for training users on how to use the system before the system has been developed.

We propose a method for prototyping in the software design phase. The feasibility of performance requirements are examined in the requirements analysis phase on whether a software system's performance requirements are realistic. Performance allocation for the components of a software system, i.e., allocation of CPU or resource utilization is determined for the first time in the design phase and the allocation is evaluated to be satisfactory with the performance requirements. A prototype of a software system enables a software designer to design the structure of the software system with consideration of its performance under an actual or simulated execution environment. Moreover, the design process substantially includes a change of system views, i.e., the change from a users' view to a programmers' view. A prototype is effective in avoiding the mismatch between these two views by its operation. Prototyping in the software design process is necessary to accomplish the following criteria: (a) to rapidly and precisely construct a skeleton of a program which is to be developed in the next implementation phase, (b) to adapt itself to user's requirements which were clarified in the requirements analysis phase, (c) to be operational or executable at arbitrary stages of the refinement process in the design phase, and (d) to be able to evaluate both function and performance at arbitrary stages in the design phase.

*Laboratory of Systems Engineering, Faculty of Science and Technology, Sophia University, Kioi-cho 7-1, Chiyoda-ku, Tokyo 102, Japan.

**University of Library and Information Science.

***Department of Information Science, Faculty of Engineering, Kyoto University.

2. Basic Concepts

2.1 Software Development with Prototyping

Fig. 1 shows software development with prototyping. Software development is divided into a construction phase and a testing phase. The former is divided into requirements analysis, design and programming phases. The latter is divided into module testing, inter-module testing and installation testing phases. Fig. 1 shows that user requirements specification, design specification and program specification are examined at installation testing, inter-module testing and module testing phases, respectively. In order to decrease the production costs, it is desirable that errors in any specification are detected and corrected in earlier construction phases before the development is advanced to the later construction phase or testing phase.

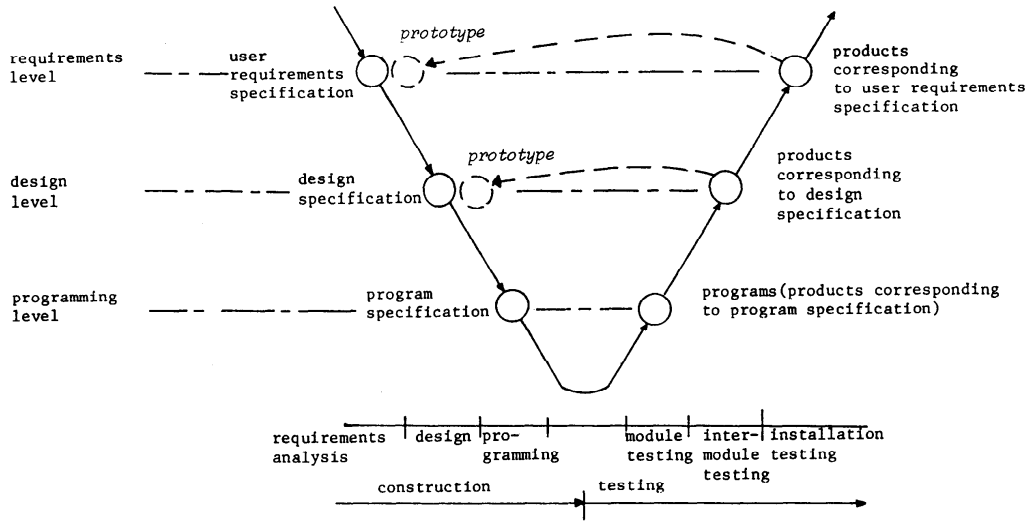


Fig. 1 Software development with Prototyping.

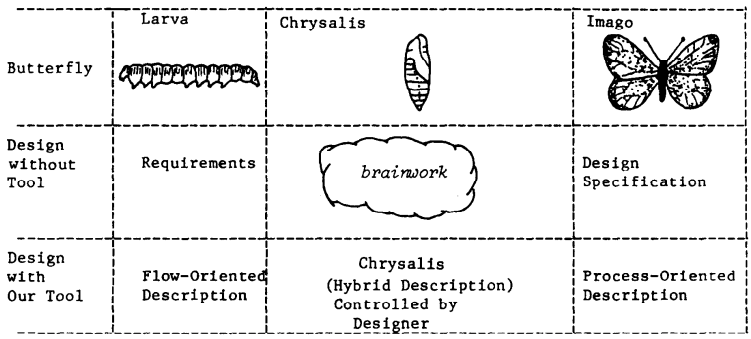


Fig. 2 Software design process.

Prototyping is one of the effective development methods because it introduces an actual or simulated environment. Prototyping in the requirements analysis phase is useful because the users can observe the behavior of a software system by the operation of its prototype under the desired environment. Prototyping in the software design phase is useful because the performance allocation of components of a software system can be determined under the desired environment, and because a mismatch between the users' view and programmers' view can be avoided.

2.2 Dual System Views in the Software Design Process

The software design process is the "chrysalis stage" in the development of a software system. We can associate the design process with a butterfly chrysalis as depicted in Fig. 2. The butterfly chrysalis seems to lie dormant, but a great number of changes are taking place. Most of the organs and other tissues of a larva are replaced by adult structures of an imago such as wings and legs. At last, the adult is ready to leave the chrysalis case [3]. The design process of a software system may correspond to abutterfly chrysalis because during the design process, brainwork seems to lie dormant due to the invisibility of its internal design efforts. It includes a drastic change which is similar to that of a butterfly chrysalis. The change in design process is that of system views for a software system, i.e., the change from a customers' (or users') view to an implementors' (or programmers') view.

We studied the components of software systems in [7] named to-be-processed demands and processing entities. We will briefly describe such components below.

A software system being executed may have one or more processing flows. There are two opposite types of entities in each processing flow. They are a "processing entity" and an entity to be processed, named a "to-be-processed demand". The first processes the second according to a program of a software system. Table 1 shows the correspondence between the two types of entities. Differences in describing the behavior of the two types of entities are shown in Fig. 3. Conceptually, each processing entity is considered to be an autonomous executable entity which can start to work under condi-

Table 1 To-be-processed demand and processing entity.

to-be-processed demand		processing entity
operand data	←→	instruction
parameter data	←→	routine, function, procedure
message	←→	process, task
input data, transaction, query	←→	online software system, database inquiry system

A process STOREs a message to a memory area.
A message ENTERs a memory area.

A process LOADs a message from a memory area.
A message LEAVEs a memory area.

A process SENDs a message to another process.
A process RECEIVEs a message from another process.
A message TRANSFERs from one process to another process.

:Statements in Roman characters represent the behavior of processing entities.
:Statements in italic characters represent the behavior of to-be-processed demands.

Fig. 3 Difference in describing two types of entities in software systems.

tions for its execution, performing its own sequential processing on its own variables and resources and, if necessary, cooperating with other processing entities in terms of shared variables and resources. Such an entity is a functional unit which has been denoted as an ACTOR introduced by Hewitt [6]. Cooperation between processing entities involves transferring a to-be-processed demand from one to another. In an actual environment, one or more processing entities execute concurrently or in parallel with each other and to-be-processed demands coexist with each other in a software system.

We introduced the concept of dual system views for a software system and originally applied it to the System Description and Evaluation System (SDES) [9] [10]. New application of these dual system views is a methodology which advances the software design process smoothly and effectively under the control of software designers.

The condition for the execution of a processing entity means the arrival of all to-be-processed demands necessary for starting its execution. A software system, which is composed of one or more processing entities, is said to be driven by such demands. In this paper, it is called a "demand-oriented software system" and the objective of this research is not a physical, lower level (i.e., instruction level) software system but a higher level software system, such as an online reservation system and a database inquiry system, which consists of routines, functions, procedures, processes, etc.

To-be-processed demands originate from the users of a software system. In the requirements analysis phase, analysts interview customers or users to understand their requirements, determine the feasibility of the requirements, enumerate as object units to-be-processed demands, e.g., input data, messages, or transactions, and determine the general flow of each of the to-be-processed demands by describing what flow occurs in the system. The system view adopted in the requirements analysis phase is called "flow-oriented view" in this paper. This view is considered in the requirements analysis by RSL of SREM [1] for real-time software systems.

Processing entities are used by the implementors to construct a software system while considering its modularity and performance. Modularity and performance sometimes contradict each other. Modularity is one of the measurements of development productivity, system reliability or system maintainability. It represents the clarity of the structure of a software system, i.e., that of logical interrelationships between processing entities. It has the possibility of overhead during execution time or in memory space. Performance is one of the measurements of effectiveness of execution time or memory space of a software system with the aid of parallel or concurrent execution of two or more processing entities, reentrant execution of processing entities, or compactness of processing entities. The trade-offs associated with modularity and performance are determined and estimated for the first time in the design phase and then a software system can be implemented according to such outline and estimation.

The system view adopted in the programming phase is called "process-oriented view" in this paper. The view is adopted in a way that processing entities, e.g., processes, tasks, or modules, are enumerated and constructed one by one by describing how they treat and transform to-be-processed demands such as transactions or messages and by describing what processing steps occur.

The design phase of a software system is considered to be the internal transformation process at the chrysalis stage of a software system from a flow-oriented system view, i.e., requirements or a larva of a software system at an early or preliminary design stage, into a process-oriented system view, i.e., design specification or an imago of a software system at pre-implementation or detailed design stage.

3. Dual-View Designing and Dual-View Integration Simulation

3.1 Overview

In order to organize the software design process into a well-structured framework, we propose a design methodology and its associated computer-aided design tool which enable software designers to manage and control the internal transformation of a development chrysalis stage effectively and smoothly. Our methodology and tool are named "Dual-View Designing (DVD)" and "Dual-View Integration Simulator (Duvis)", respectively. The object software system to which DVD and Duvis are applied is called a "demand-oriented

software system" described in Section 2.2. The example of a demand-oriented software system to which DVD was applied is the online sales order entry software system, handling one hundred to-be-processed demands of seven types of transactions each minute by processing entities of three modules.

DVD is a methodology by which a software designer can construct a prototype of a software system based on four criteria for prototyping described in Section 1. In DVD, the flow-oriented view is applied in constructing a prototype in early or preliminary design, the process-oriented view is applied in final or detailed design, and the hybrid of both views is applied in intermediate design. Duvis is a computer-aided design tool which tests and evaluates the evolving prototype of a software system in the design phase. In parallel with the advancement of the design process, the designer gradually introduces the actual running environment of a software system to be installed. Final prototypes by process-oriented view are skeletons of programs to be implemented at the succeeding programming phase. Fig. 4 shows the advancement of the design phase of a demand-oriented software system by DVD under the support of Duvis.

Fig. 5 shows the contrast of the two views and the gradual change of the relative proportion between flow-oriented and process-oriented factors in keeping steps with advancement of DVD process.

The DVD design phase is a transformation process from a flow-oriented view passed by the requirements analysis phase into a process-oriented view passing to the programming phase. The former view is implementation-free view in which flow of each to-be-processed demand may be considered from the viewpoint of customers or users. On the other hand, the latter view is an implementation-constrained view in which effectiveness must be considered from the viewpoint of implementors or programmers. The latter view advances the design process in the way that flows of to-be-processed demands

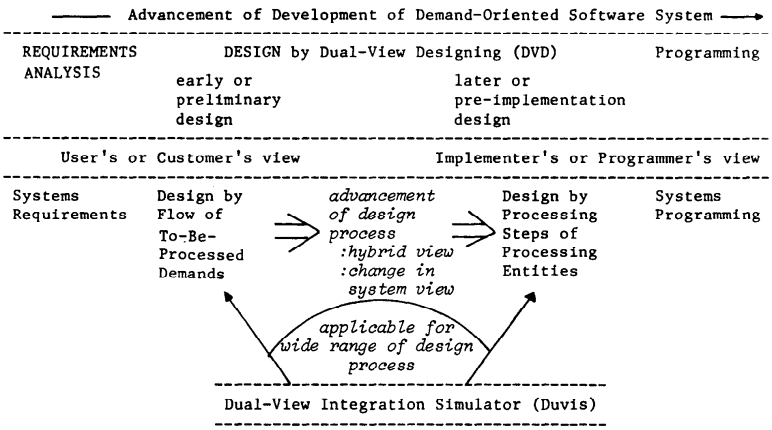


Fig. 4 Design process by DVD and Duvis.

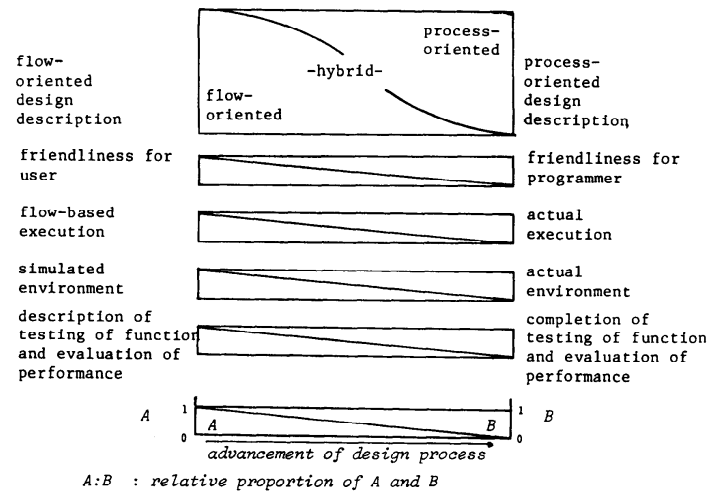


Fig. 5 Variation of design factors with the advancement of design process.

are decomposed into sub-demands, and such sub-demands are grouped into shared ones, reusable ones or independent ones, and then processing entities are assigned in order to work upon such grouped sub-demands.

Duvis is a computer-aided design support tool which performs flow-based execution for flow-oriented prototypes, actual execution for process-oriented prototypes, and their combined or hybrid execution for hybrid prototypes. Concurrent with the advancement of the design process by DVD, prototypes are fully tested and evaluated by initially providing a simulated environment and gradually introducing an actual environment on which a software system is to be installed. The environment description is included in the prototypes. Duvis can test and evaluate a partially defined software system on a simulated environment where its partially defined parts are replaced by simply-defined stubs or time-advancement descriptions representing estimated time. Duvis provides testing results by hybrid execution of the prototypes and evaluation results by collecting statistics about execution time or resource utilization. While the description of testing and evaluation is gradually decreased in parallel with the DVD design process, the testing and evaluation design data is gradually accumulated.

3.2 Comments on Related Works on Software Development

Jackson Structured Programming (JSP) [11] and System ARchitecture Apprentice (SARA) [2] are representatives of well-structured methods of software design.

In JSP, software design is advanced in the way that the program structure is clarified by the process of studying the correspondence between input data structure and output data structure. The program created by JSP is one that can be used serially and exclusively by input data. JSP designs the intra-program structure rather than the inter-program structure.

SARA provides a hierarchical description mechanism for concurrent software systems in a precise and understandable fashion after preliminary or early design is almost completed.

DVD differs from JSP and SARA by considering the whole of the software design process. DVD begins with analyzing the flow of to-be-processed demands and determines the inter-module structures of concurrent processing entities from the viewpoint of the trade-off between modularity and performance.

Structured Analysis [12] provides a method for requirements specification by which the function of software systems is hierarchically defined with the aid of the decomposition of "data flow" in software systems. The flow-oriented view in DVD resembles the data flow view in the Structured Analysis because the data flow means the flow of users' messages or transactions in the software systems. DVD is a method for software design by nature, which introduces not only flow-oriented view but also process-oriented view and performance estimation in order to effectively produce skeletons of software systems.

3.3 Constructs in Flow-Oriented Design

Flow-oriented design work in DVD is performed in top-down fashion. A flow-oriented prototype has the modularity of a hierarchical structure. This tree structure is easily understood. Such a flow-oriented top-down technique was originally introduced in [8]. It was motivated by the structured programming techniques [4] and the top-down programming techniques [13] for implementing ordinary programs. Design components for constructing a flow-oriented prototype are a transaction, an activity, an "actentity", a "comact", a stub, a facility, a storage, a time-advancement, a queue, a semaphore, a private variable and a common variable.

A transaction acts on other components. It represents a unit of traffic moving through a software system and corresponds to a to-be-processed demand. It enters the system, acts on other components and exits. A private variable represents one of its attributes inaccessible by others. A common variable represents a state of a software system accessible by others. A facility and a storage represent time-shared equipment and space-shared equipment, respectively. The number of transactions which can simultaneously use the equipment is called the capacity. A queue represents an ordered set of transactions that are waiting for a facility or storage. A semaphore represents binary-state equipment for the synchronization of traffic between two or more transactions. These design components are called elementary entities.

A time-advancement component is a transaction that stays in the present place without being affected by other components for a time specified by this component. It is used for the submission of more detailed specifications to the later design stages and the estimated time which it takes in execution of an already specified and yet-to-be-specified part of a prototype. It is one of the key factors enabling the execution and evaluation of a not-fully-specified prototype.

The word "actentity" is derived from "activity entity". It is an abstract entity representing a portion of a software system. Its action—what it does—must be fully defined, but it is not always necessary to know how to construct it. The action will be constructed in the definition of the activity in the next lower level in top-down structure. Actentity and its corresponding activity have an identical name. The word "comact" is derived from "common actentity". It is a special actentity which allows more than one activity to share it. Its action is also specified in the definition of an activity like an ordinary actentity.

A prototype in an arbitrary stage in a top-down process can be easily checked for its syntax and structure by Duvis. Moreover, Duvis executes it for test by providing it with stubs by a designer. A stub simulates the

presence of a yet-to-be-specified actentity and is defined like an activity in simple or even sophisticated fashion.

Elementary entities, except for a transaction, can have an array structure, i.e., a collection of entities with identical properties such as types and capacities. A subscripted elementary entity is used for referring to a component of the array.

3.4 Procedure in Flow-Oriented Design

Behavior of a system can be represented in one or more flows of transactions through parallel components of the system. DVD allows the user to build a hierarchical structure with one activity or several parallel activities each of which may be developed in top-down fashion.

The following procedure shows this top-down process.

(0) Designers enumerate types of transactions for the system and components of the system.

(1) When considering the flow of each type of transaction, the function of the component may be determined by dividing it into several parts. These parts correspond to actentities and other entities. If a part may be shared by two or more components, it may be a comact.

(2) The control for transaction flow may be described as an activity which affects design components. In the case that two activities which represent the control for flow of different types of transactions are identical, they are arranged into a single activity.

(3) Estimation of time may be given to the arbitrary parts.

(4) If necessary, Duvis may perform flow-based execution for the prototype obtained by (3).

(5) Substitute "actentity" for "component" in steps from (1) to (3), and iterate the procedure in all steps until all of the actentities are finally represented by elementary entities.

3.5 Constructs and Procedure in Process-Oriented Design

Process-oriented design work is performed in stepwise refinement fashion so that activities obtained in flow-oriented design work are incrementally replaced by a new design construct, named an actor. It corresponds to a processing entity which is described in a programming language.

A time-advancement design component may remain or be improved by giving to a refined part of a prototype a more precise estimated time, according to refinement level, until the end of process-oriented design work. Design components such as a facility, a storage and a queue may also remain for collecting performance information. These enable the designer to get improved performance evaluation results by Duvis.

The following procedure shows this stepwise process.

(0) An activity in flow-oriented design may be arbitrarily selected.

(1) The selected activity is rewritten as an actor in a programming language, not in the way that a transaction uses any other components but in the way that a processing entity receives a transaction and works upon it. Such work may be performed on either the whole or a part of the selected activity.

(2) An improved estimation of time may be given to the rewritten part according to refinement level (e.g., execution time in proportion to the number of statements).

(3) If necessary, Duvis performs hybrid execution for the prototype obtained by (2).

(4) Incrementally rewriting the process iterates the procedure in all steps until the whole of a flow-oriented description becomes a process-oriented description.

3.6 DVD Language

A prototype description language in DVD is named "DVD Language (DVD/L)" in which the behavior of to-be-processed demands and the processing steps of processing entities are described. In DVD/L, language components for the two main types of entities are named a transaction and an actor as mentioned above. A transaction description and an actor description are flow-oriented and process-oriented forms, respectively.

We have designed a transaction-type language as a flow-oriented language. It is a GPSS[14]-like language enhanced by structured programming and top-down methods [8] [10] with a transaction identifier in order to distinguish variour types of to-be-processed demands and describe their flow in a well-structured form. We adopt PL/I, with multi-task option, as a language for processing entities because it enables the designer to test and evaluate a currently-designed software system under a PL/I-written actual installation environment, and because final prototypes in detailed design are skeletons of programs in the programming phase.

Both entities for flow-oriented design and PL/I variables for process-oriented design have their own scope of references. Their scope consists of the activity and actors in which they are declared and all encom-

Table 2 Statements of DVD/L.

Structure Statement

statement	structure
IF,CASE, CONDITION	selection
REPEAT,DO	repetition

Non-Structure Statements

statement	entity
ENTER,LEAVE	storage
USE,PREEMPT, SEIZE,RELEASE	facility
HOLD	time-advancement
SET,RESET, WAIT UNTIL	semaphore
assignment	private/common variable
ACT, USE ACT	activity
GENERATE TERMINATE	transaction
compound	statements enclosed by BEGIN and END

passed activities and actors without another declaration of the same identifiers. Once entities are declared in the definition of an activity, they may be used in any activity and actor included in the subtree whose root is the activity with their declaration. The entities, which are used in an activity of a comact and are not local ones to the activity, have to be declared in the root activity of any one subtree which includes all the activities invoking the comact.

Statements of a transaction language are grouped into structure and non-structure statements. Non-structure statements are simple and compound statements. Table 2 shows these statements.

3.7 Testing and Evaluation Functions of Duvis

Dynamic testing of function is carried out by flow-based, actual or hybrid execution of a prototype. Table 3 shows items for testing and evaluation of function and performance.

3.8 Script for Interaction between Designers and Duvis

Prototypes may be executed in step (4) of the procedure in Section 3.4 and step (3) of the procedure in Section 3.5. Designers can interact with Duvis according to the

Table 3 Items for testing and evaluation.

Items for Testing

invalid flow	invalid flow of transaction (violating user specified assertion)
invalid status	invalid status of entity (violating user specified assertion)
dead lock	suspention of transaction flow (automatic)
abnormal access	improper access for storage/facility (automatic)

Items for Evaluation of Facility/Storage

total entries	the total number of transactions already entered
average contents	the rate of contents per elapsed time
average utilization	the rate of average contents per capacity

Items for Evaluation of Queue: the above statistics except for average utilization

Items for Evaluation of Transaction

generation count	the total number of transactions already generated
termination count	the total number of transactions already terminated
average duration	the mean of intervals from generation to termination

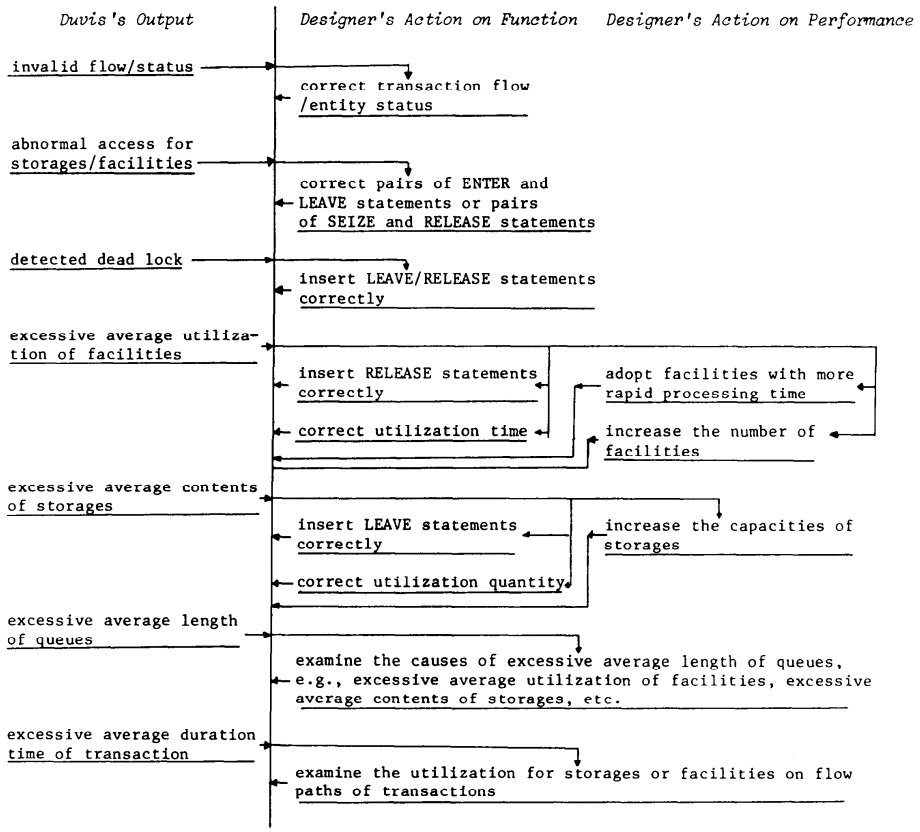


Fig. 6 Script for interaction between designers and Duvis.

script as shown in Fig. 6. There are both function and performance aspects of the interaction. The function aspect means the correction of functional errors of prototypes. The performance aspect means the improvement of performance of prototypes to satisfy the performance requirements for software systems.

4. DVD Command System

DVD Command System controls the progress of design and evaluation work in a TSS environment. Four modes are provided: design specification, design translation, design simulation and batch processing modes. (See Fig. 7.)

In design specification mode, a prototype is built with the use of a built-in, line-based editor whose functions are appending, deleting, listing, etc. In design translation mode, if there are some syntax errors, error messages are outputted and the control must be returned to design specification mode. A designer can get the arbitrary level of translation output with/without a source list of a prototype, a list of its internal form, a table of utilized entities or error messages. Batch processing mode is provided for the case in which a prototype requires large CPU-time for its test and evaluation. A prototype is submitted to the computer from a TSS terminal and executed off-line. The simulation results can be obtained at the terminal after its execution.

In design simulation mode, a designer can simulate, test and evaluate his prototype with the use of interactive simulation commands which specify the manner of simulation and the format of statistics about entities. The interactive simulation commands are grouped into simulation control commands and simulation report commands. Each simulation control command is specified with its function as listed in Table 4. Simulation execution is performed in an arbitrary sequence of the simulation control commands. When it is suspended, a designer may obtain the state of a system with the use of simulation report commands. These allow a designer to inquire into any statistics about transactions or entities according to his choice. The report has decimal representation of statistics described in Table 3.

5. Implementation of Duvis

The flow-based execution for a flow-oriented prototype is implemented in terms of an interpreter for the description and a scheduler for transactions. The actual execution for process-oriented parts is implemented in the ordinary cycle of program execution, i.e., a cycle of PL/I compilation, linking and execution after such descriptions, i.e., PL/I-written part, are extracted and assembled into a group of PL/I procedures. The hybrid execution for hybrid prototype is implemented in terms of the interpreter for flow-based execution linked with the compiled PL/I parts. A combined program of the scheduler and interpreter is called a Duvis flow-based transaction simulator. The scheduler manages the event-occurring time at which transactions start, pause, restart or terminate. Such events may occur concurrently or in parallel, so the simulator performs quasi-parallel control of events by quasi-parallel execution of all of the

Table 4 Simulation control command and its function.

Command	Function
Step-by-Step	Every time one statement is executed, the simulation run is suspended.
Clock-by-Clock	Every time the simulation clock time is updated, the simulation run is suspended.
Run	At the simulation clock time specified by designers, the simulation run is suspended.
Condition	When the number of terminated transactions becomes the number specified by designers, the simulation run is suspended.
Roll-Back	The simulation run is returned to the previous simulation clock time specified by designers.

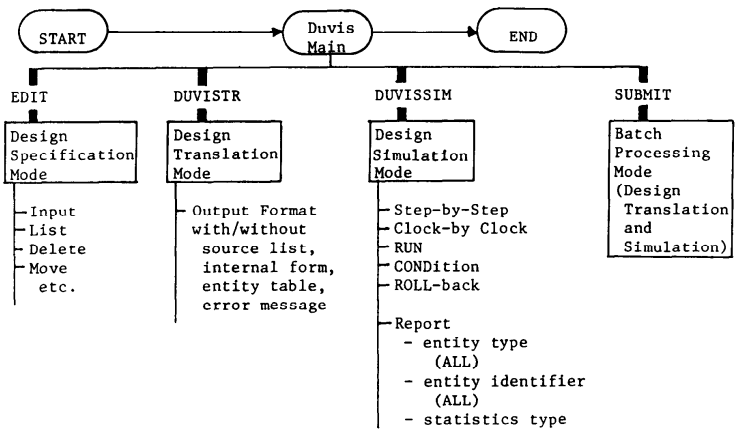


Fig. 7 DVD command system.

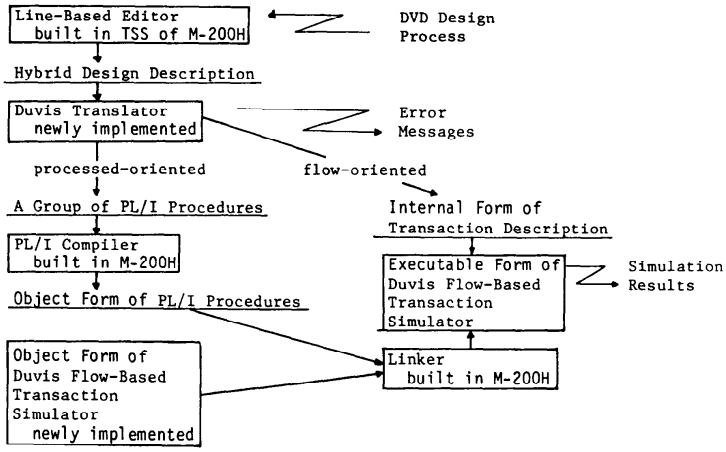
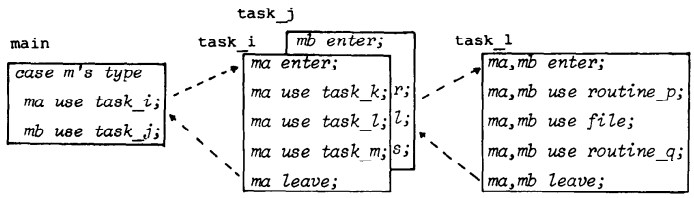
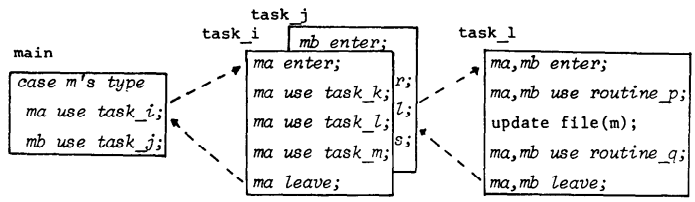


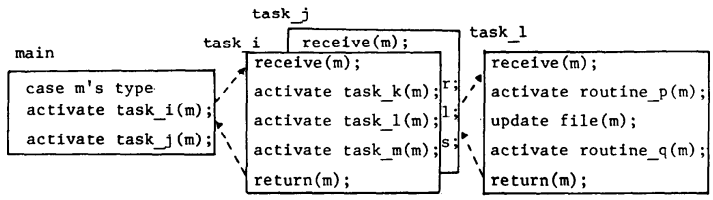
Fig. 8 Internal structure of Buvis.



(a) Flow-oriented design description



(b) A snapshot of hybrid design description



(c) Process-oriented design description

left of (a) ⟶ right of (a) : Top-Down Process

(a) ⟶ (b) ⟶ (c) : Incremental Substitution Process

Fig. 9 Advancement of design process of DVD (to be continued).

activities and actors.

We have developed Duvis on a HITAC M-200H, a large-scale IBM/370-compatible computer, under VOS3 roughly equivalent to MVS/TSO, installed in University of Tokyo. Fig. 8 shows the internal structure of Duvis. A line editor, a PL/I compiler and a linker are built in a computer and utilized without modification. A Duvis translator and a Duvis flow-based transaction simulator have newly been implemented in Fortran IV.

6. Example

Fig. 9 shows the advancement of DVD design process, where the example software system is composed of eight modules named main, task-i, task-j, task-k, task-m, task-r, task-s and task-l and two types of to-be-processed demands named ma and mb. In Fig. 9(a)–(c), italic and Roman characters represent flow-oriented and process-oriented descriptions, respectively. Fig. 9(a)

shows the top-down process by flow-oriented view where the flow of ma is determined by the sequence of main, task-i, task-k, task-i, task-l, task-i, task-m, task-i and main and the flow of mb is determined by the sequence of main, task-j, task-r, task-j, task-l, task-j, task-s, task-j and main. Fig. 9(b) shows a snapshot of an incremental substitution process where an actual environment such as a file structure is introduced in task-l. Fig. 9(c) shows the end of such process where all of the descriptions are process-oriented, i.e., eight modules (processing entities) handle the to-be-processed demands concurrently. Fig. 9(d) shows a description in DVD/L for Fig. 9(b). The PL/I-written part is processed-oriented as an actor; parts underlined with dashed lines represent estimated time for execution of the parts; and a transaction generation part represents that two types of to-be-processed demands generate with exponential distribution and a mean of two hundred units of time. Fig. 9(e) shows the simulation results for Fig. 9(d).

```
DUVIS ONLINE(MAIN,OPENF:MA,MB,OPN);
  FACILITY ROUTINE_P(1),ROUTINE_Q(1);
  FACILITY FILE(1),F_TASK_L(1);
  FACILITY F_TASK_I(1),F_TASK_J(1);
  PRIVATE SORT;
 ACTIVITY MAIN;
  ACTENTITY TASK_I,TASK_J;
  CONACT TASK_L;
 ENTRY;
    CASE SORT OF 2;
     M1: <MA> USE F_TASK_I FOR TASK_I;
     M2: <MB> USE F_TASK_J FOR TASK_J;
 EXIT;
END MAIN;
  DECLARE I FIXED DEC(9,0) STATIC;
  DECLARE WKFILE FILE RECORD KEYED DIRECT
   ENV(REGIONAL(1) F RECSIZE(5) BLKSIZE(5));
  ACTOR OPENF:0;
   I=1;
   OPEN FILE(WKFILE) UPDATE;
  END OPENF;
ACTIVITY TASK_I;
  FACILITY F_TASK_K(1),F_TASK_M(1);
  ENTRY;
    <MA> USE F_TASK_K FOR EXPO(10);
    <MA> USE F_TASK_L FOR TASK_L;
    <MA> USE F_TASK_M FOR EXPO(15);
  EXIT;
END TASK_I;
ACTIVITY TASK_J;
  FACILITY F_TASK_R(1),F_TASK_S(1);
  ENTRY;
    <MB> USE F_TASK_R FOR EXPO(15);
    <MB> USE F_TASK_L FOR TASK_L;
    <MB> USE F_TASK_S FOR EXPO(10);
  EXIT;
END TASK_J;
ACTOR TASK_L:0;                                  Actor Written
   <MA MB> USE ROUTINE_P FOR EXPO(20);           in PL/I
  REWRITE FILE(WKFILE) KEY(PRM(I)) FROM(I);
  I=I+1;
   <MA MB> USE ROUTINE_Q FOR EXPO(15);
END TASK_L;

GENERATE(200,EXPO(200),200);
SORT=1;
EXEC MAIN;
TERMINATE;
STOP(200,30000);
GENERATE(200,EXPO(200),200);
SORT=2;                                  transaction
EXEC MAIN;                               generation
TERMINATE;                               part
STOP(200,30000);
GENERATE(1,EXPO(1),1);
EXEC OPENF;
TERMINATE;
STOP(200,30000);
END ONLINE;
```

Fig. 9 (d) Description in DVD/L of (b).

7. Concluding Remarks

We have proposed a method named DVD for prototyping in software design based upon the study of basic concepts, flow-oriented design view by to-be-processed demands and process-oriented design view by processing entities. Prototypes of software systems such as online software systems and database inquiry systems are designed in top-down fashion from the flow-oriented view at its early design stage and then they are designed in stepwise substitution fashion from the process-oriented view at its later stage. Duvis is a computer-aided design support tool for DVD which performs testing and evaluation of function and performance for evolving prototypes, while gradually introducing an actual running environment in parallel with the design process. Final prototypes are skeletons of programs to be implemented. We expect the to assess the applicability of DVD and Duvis with additional software engineering environments.

Acknowledgements

This research was supported in part by the Foundation of Hattori-Hohkoh-Kai of Japan under Grant KOHGAGU SHOHREI 55 and 56 and also supported

STATISTICS **

IDENTIFIER	CAPACITY	AVERAGE CONTENTS	AVERAGE UTILIZATION	NUMBER ENTRIES	AVERAGE TIME/TRANS
ROUTINE_P	1	0.18	0.18	229	18.72
ROUTINE_Q	1	0.13	0.13	228	12.95
F_TASK_L	1	0.87	0.87	229	88.97
F_TASK_I	1	0.86	0.86	115	175.58
F_TASK_J	1	0.86	0.86	115	175.13
F_TASK_K	1	0.05	0.05	115	9.85
F_TASK_M	1	0.08	0.08	114	16.39
F_TASK_R	1	0.06	0.06	115	12.95
F_TASK_S	1	0.04	0.04	114	8.82

*** MEAN FLOW TIME ***

GENERATION LIST	GENERATION COUNT	TERMINATION COUNT	MEAN FLOW TIME	STANDARD DEVIATION	MAX FLOW TIME
1	146	143	682.203	593.661	2644
2	146	143	627.490	487.193	2465

Fig. 9 (e) Simulation results.

in part by the Ministry of Education, Science and Culture of Japan under Grant SHOHREI 56790034.

References

1. BELL, T. et al. An Extendable Approach to Computer-Aided Software Requirements Engineering, *IEEE Trans. Softw. Eng.* 3, 1, (Jan. 1977), 49–60.
2. CAMPOS, I. M. et al. Concurrent Software System Design Supported by SARA at the Age of One, *Proc. 3rd ICSE*, (May 1978), 230–242.
3. Compton's Encyclopedia, F. E. Compton Company, 1972.
4. DAHL, O. J. et al. Structured programming, Academic Press, 1972.
5. GOMAA, H. et al. Prototyping as a Tool in the Specification of User Requirements, *Proc. 5th ICSE*, (Mar. 1981), 333–339.
6. HEWITT, C. Viewing Control Structure as Patterns of Passing Messages, *Artif. Intell.* 323–364, North-Holland Publishing Company, 1977.
7. ITOH, K. et al. An Evaluation System for Concurrent Processes by the Traversing Method, *Proc. 3rd USA-Japan Computer Conference*, (Oct. 1978), 41–45.
8. ITOH, K. et al. Interactive Modeling and Simulation System, *Proc. 1978 International Conference on Cybernetics and Society*, (Nov. 1978), 1247–1252.
9. ITOH, K. et al. System Description and Evaluation System: SDES, *Trans. IPSJ* **20**, 4 (July 1979), 352–362.
10. ITOH, K. A Study on Systems for Functional Verification and Performance Evaluation in Software Design, Doctoral Thesis, Dept. of Information Science, Kyoto Univ., 1980.
11. JACKSON, M. A. Principles of Program Design, Academic Press, 1975.
12. DEMARCO, T. Structured Analysis and System Specification, Prentice Hall, 1978.
13. MILLS, H. Top-Down Programming in Large Systems, Debugging Technique in Large Systems, Prentice-Hall, 41–55, 1971.
14. General Purpose Systems Simulator (GPSS) User's Manual, IBM.

(Received April 26, 1982: revised August 26, 1983)