

SDL '95 WITH MSC IN CASE

*Proceedings of the Seventh SDL Forum
Oslo, Norway, 26-29 September, 1995*

edited by

Rolv BRÆK
*SINTEF DELAB
Trondheim, Norway*

Amardeo SARMA
*EURESCOM
Heidelberg, Germany*



1995

ELSEVIER

Amsterdam • Lausanne • New York • Oxford • Shannon • Tokyo

High Quality Design Using SDL Technology

Zenya Koono, and Behrouz H. Far

Department of Information and Computer Sciences, Saitama University,
255 Shimo-okubo, Urawa 338, Saitama, Japan
koono@cit.ics.saitama-u.ac.jp

This paper describes a design procedure with documentation for embedded software systems aiming at high quality. It starts from process level design. A system is decomposed to several orthogonal processes by data flow division. Making each block or process mono-functional and repeating the division, the system is reduced to a hierarchical cluster of Extended Finite State Machines. The design proceeds by converting an abstract level message sequence chart to multi-process message sequence charts, to a state diagram for one trace, and finally to a state diagram including all traces. Each state transition route is finally converted to source code. A design of an embedded system is thus partitioned to many small step designs each accompanied with the interfacing documents. Through careful, rigorous and repetitive documentation and check, the overall error rate is decreased. Experiments based on applying this method in developing skills of software design and education are reported.

1. INTRODUCTION

With the advances of the computer technology and software systems becoming the backbone of the modern society, an undetected software error might cause a big disaster. An error in switching software might cause a continental wide telephone traffic jam, and an error in anti-skid brake software might cause a car accident. The reliability of embedded system's software is required to be higher than that of the reliable hardware. It is the purpose of this paper to report a design method [6] for embedded software systems aiming at achieving such reliability.

It is generally accepted that software reliability depends upon the residual error rate of the software at the time of ship-out, and the rate depends on the software development work process. In Japan, most of the software developers improve their work process by Total Quality Control (TQC). For example, a software company could improve the ship-out error rate from 0.045 to less than 0.001 errors/kilo-steps by improving their work process through corporate wide TQC activities [10]. The characteristic point of their work process is their abundant document system. In an embedded software system development, their work practice lists up around 20 kinds of documents including state transition diagrams and state-event matrices. Another software company's standard is to prepare more than 100 pages of A4 document for 1 kilo-steps source code.

This correlation between software error rate and documents is very clear in Japanese software industry [9]. As the document system has been established through many years of work process improvement, only empirical effects are visible. To the best of our knowledge, there is no quantitative model that can explain the effectiveness of software documentation. This paper describes a model [7–9], explains design procedures, and reports on the design procedure together with some experimental results as applied to design of embedded systems.

2. PRINCIPLES OF QUALITY DESIGN

2.1. Design and its error

Figure 1 [2] shows an example of design progress. This is used to explain the procedure without necessarily including the technical details. The initial concept, *clock* in this case, is detailed by data flow diagrams in the left, transformed to flow charts in the middle and then converted to code in the right. Each data flow design consists of many steps of hierarchical detailing, each requiring expertise of a human designer. Hence, design of software consists of chain of human mental operations.

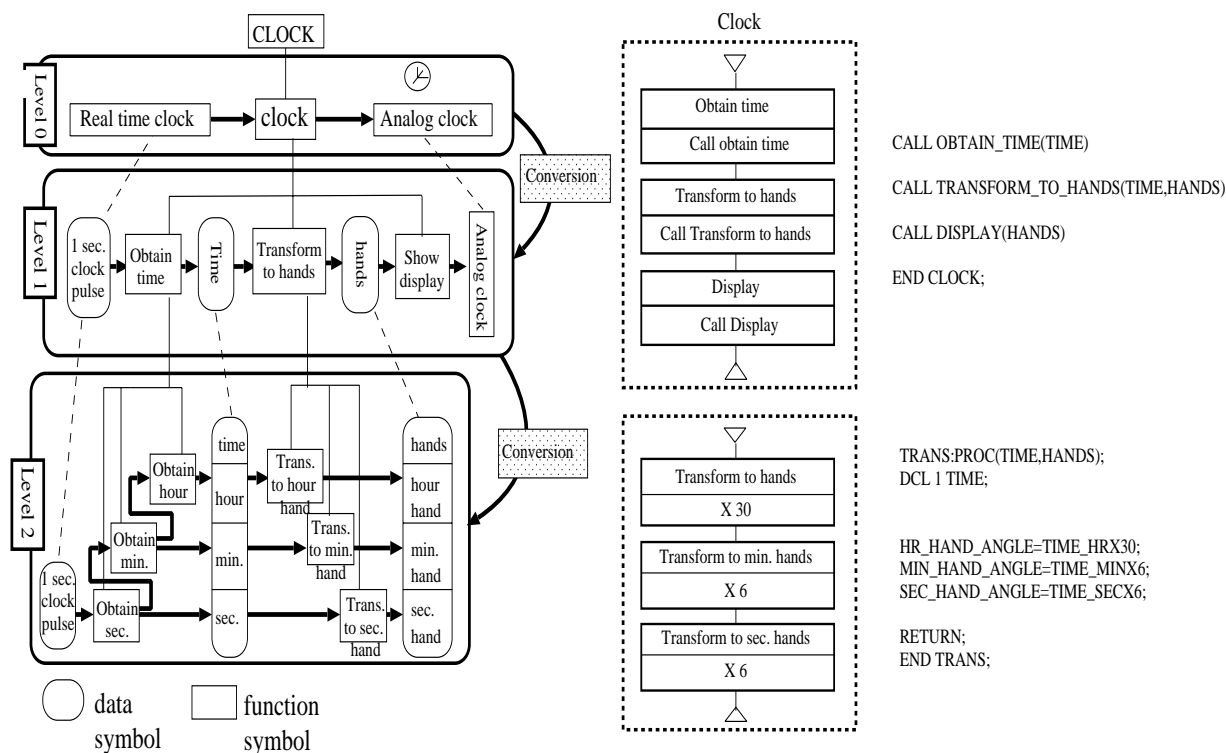


Figure 1. A simple software design case

It is a usual practice in Human Reliability Engineering [1] to decompose a large task to a chain of elementary operations, and evaluate the systems error rate by multiplying the number of operations and the Human Error Probability (HEP) [13]. A well known example is Fault Tree Analysis. This is applied to software design [7–9].

When performing mental operations during design, a human inevitably makes mistakes at a small rate. This is called Human Error Probability (HEP_d). The error build-in rate is given by,

$$E_b = S \times HEP_d \quad (1)$$

where E_b is the error built-in rate; and S is the number of mental operations per number of lines of code.

A novice has an error build-in rate of around 100 errors/kilo-line (or 0.1) even in simple programs, and experienced designer's error build-in rate is lower than 10 errors/kilo-line (or 0.01).

It is an empirical fact that checking after design reduces errors. Let assume that another person checks the initial design. The check is also a kind of human operation, therefore it inevitably errs at some rate. Let's assume that check is first, a re-practice of the design, second, comparison of the result with the original design and third, the comparison is error free. Therefore the errors remain when both the original design and the check produce the same error. The residual error rate is given by,

$$E_r = (S \times HEP_d) \times (S' \times HEP_c) \quad (2)$$

where E_r is the Residual-error-rate; S' is the normalized number of mental operations during check and HEP_c is the Human Error Probability during check. When the design is re-practiced, $(S \times HEP_d)$ and $(S' \times HEP_c)$ have the same value. When they are both 0.1 (100 errors per kilo-line of code), the residual error rate is 0.01 or 10 errors/kilo-line of code. Errors are decreased to 1/10. If check is repeated C times, the residual error rate is given by

$$E_r = (S \times HEP_d) \times (S' \times HEP_c)^C \quad (3)$$

In an actual design, however, the reduction is not so high. If the same person checks again, the check includes the similar kind of mental operations where the initial design errs. So the important point of checks after design is to make checks independent from the initial design. A main purpose of check lists and various independent reviews and inspections is to reduce this dependability.

There is another approach to reduce the errors. Let assume that a design is divided to M sections of design each requires the same number of mental operations, and a single step check is performed after each design section. The residual error rate in this case is given by,

$$E_r = ((S/M) \times HEP_d) \times ((S'/M) \times HEP_c) \times M \quad (4)$$

or

$$E_r = (S \times HEP_d) \times (S' \times HEP_c)/M \quad (5)$$

This also shows that the residual error decreases to 1/M.

Though this calculation assumes that both design and check err at random, this result coincides with many empirical facts. They are as follows:

- Checks on small steps of design progress are easy, accurate and effective.
- A team with high quality performance always has a rich document system.
- The independent check increases the overall performance quality.

The latter is usually developed by providing countermeasures from root cause analysis during TQC activities.

Machine testing is essentially a comparison of the initial product design and the test design. So the situation is similar to the checks described before.

2.2. Design document and check system

A main point of concern is how to give an active role to the documents in the design process, and to reuse them in future design practices. Figure 2 [5] shows relationships in the design process. The left side shows hierarchical decomposition, and the right side shows hierarchical integration with testing. The problem is how to incorporate a document and check system into the design process. The principle is to divide design into small sections and therefore make the designs easy to check.

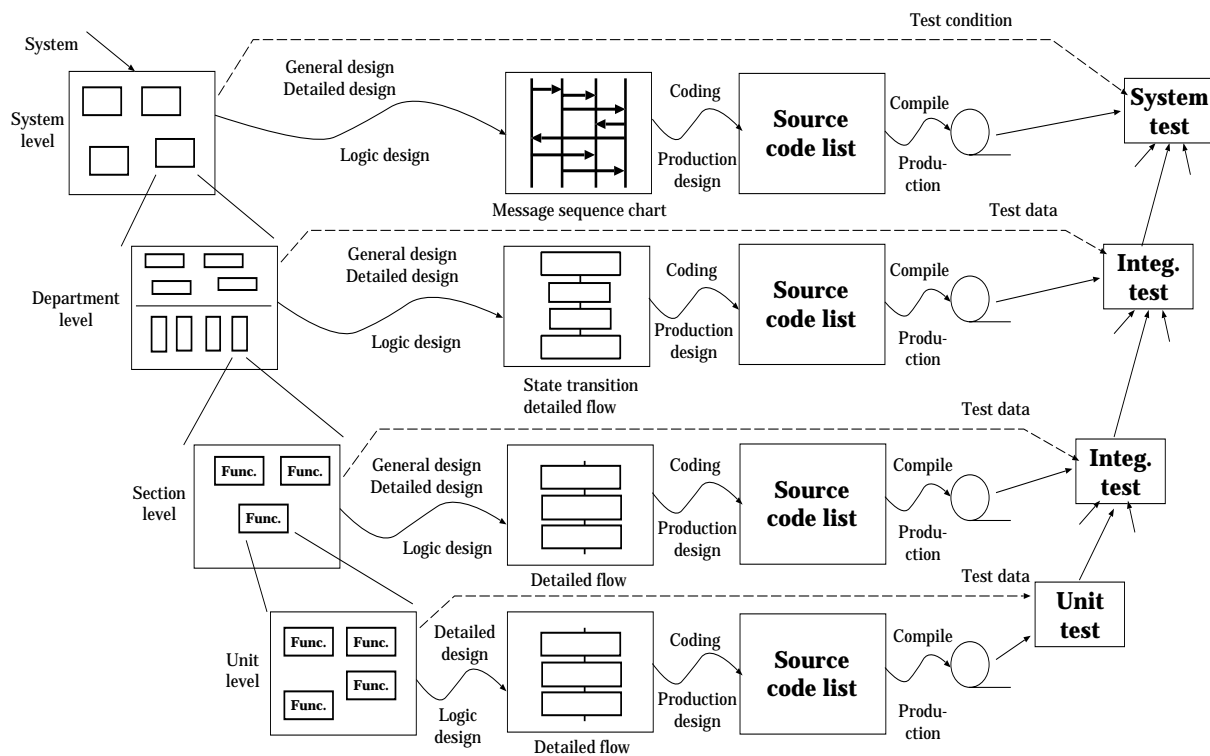


Figure 2. A relation of design process

It is obvious that hierarchical break down of processes must be made at many levels. A simple measure would be making each process mono-functional. The hierarchical decomposition processes accompanies essentially the same documents to enable sequential processes.

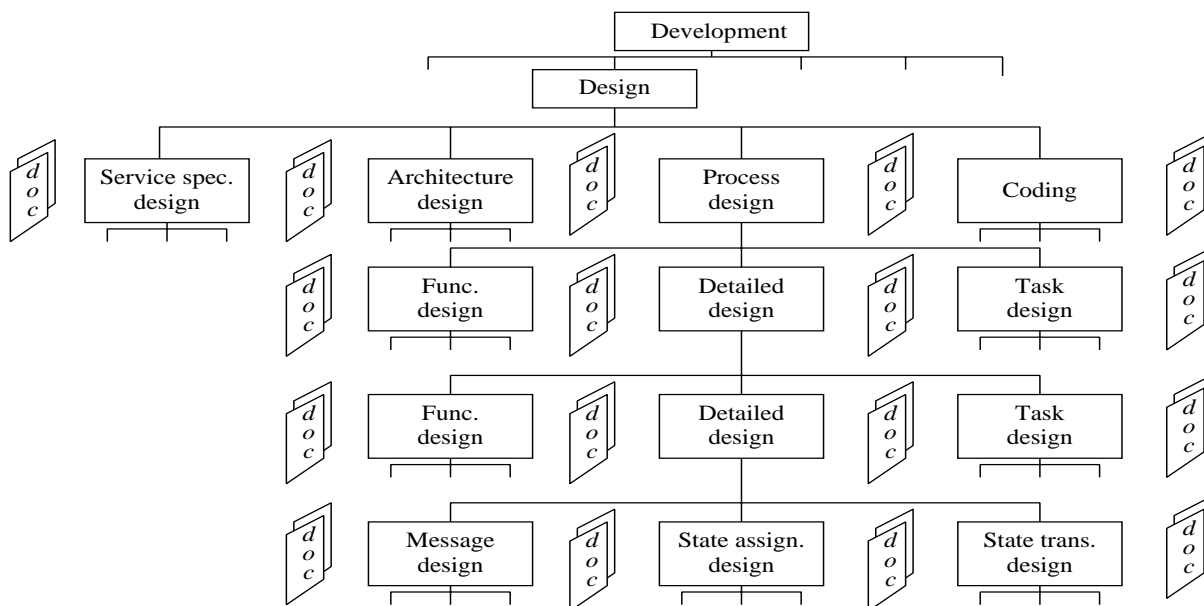


Figure 3. Document structure

Figure 3 shows the hierarchy of design documentation. A design process is divided hierarchically, and at each interface of the design process, a set of design documents must be provided. Besides these major stream documents, there are some other minor but effective documents such as state-event matrices.

Various statistics on checks after design (reviews and inspections) show that the check efficiency is around 1 man-hour per checked-out error for typical design errors. When these errors are checked-out by machine testing, it consumes several man hours per error. It is more economical desk check for errors than to check by machine. A single step check, however, is not perfect, it is usually necessary to check several times.

It is important for designers to prepare documents at each small design step, check rigorously, revise documents and repeat checks. However, design documentation practice usually contradicts with the designers' desire of producing less documents, whenever possible. It is an important merit of using CASE tools if they could offer quality documents and reduce the man power required for document preparation and revision to a minimum level.

3. HIGH LEVEL DESIGN

Figure 4 shows a hierarchical structure with system and block in SDL style. In the initial level design, a system is partitioned to several sub-systems. The dividing is generally made based on the abstracted concept, considering minimization of the software size, by making the structural blocks mutually independent.

This is performed by considering a major abstract data flow and dividing it along the flow. Flows A, C, G, and B is the major data flow. Sub-system S_sA corresponds to an input processing, subsystem S_sB corresponds to an internal processing, and sub-system S_sC corresponds to an output processing. Perhaps the best way to divide the flow is to use Myers' S-T-S theorem [11], in which the interface C (input side the most abstracted point)

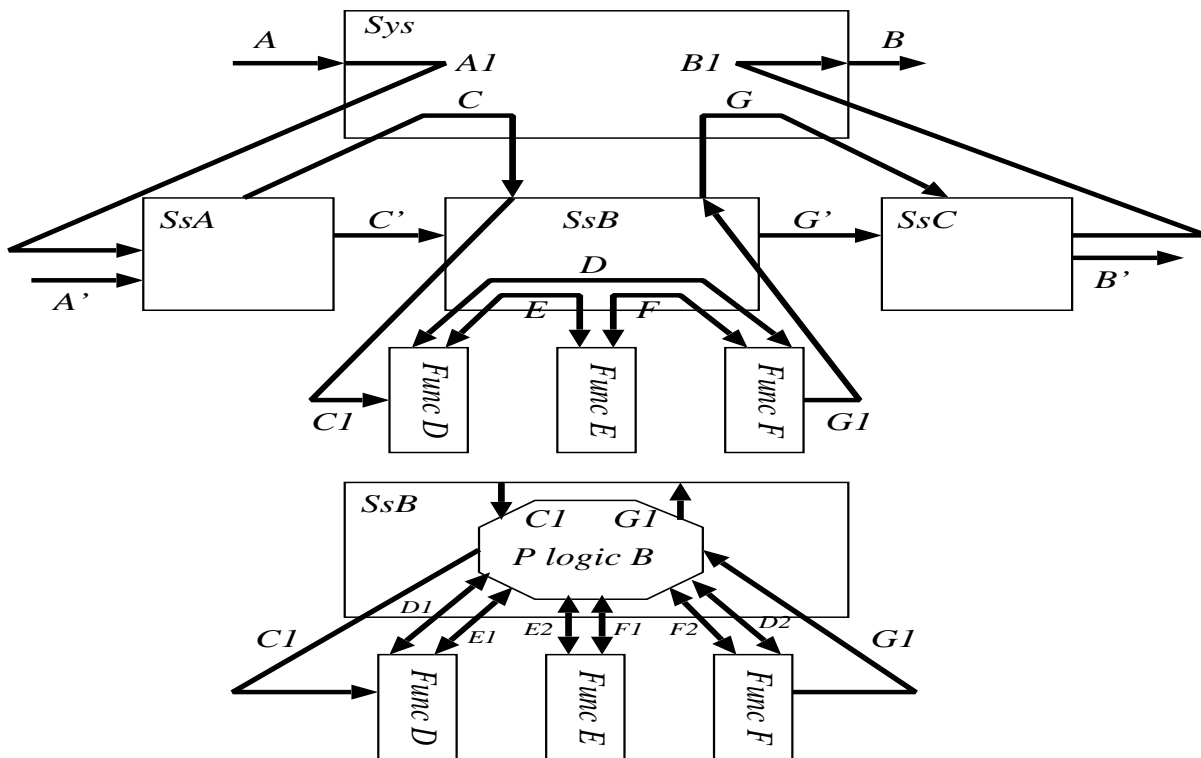


Figure 4. High level design

must be the most distant concept from the input still remaining the input concept, and the interface G (output side the most abstract point) must be the most distant concept from the output still remaining the output concept. These interfaces must be expressed by abstract concepts. Actual input and output may be physical signals and actual interface may be A' or B' . Irregular interfaces in input and output may be isolated to minimize their effect on the central system.

Now that the interface has been decided upon, the sub-system functions should be considered. In practice, each such function is decomposed to several lower level functions. In this example, the input and output of sub-system S_sB are clearly defined in their concept. If the sub-system S_sB is clear and small enough, it can be implemented by a process, as shown in the bottom of Figure 4. The functions D , E and F and the process logic are clearly defined, resulting in higher quality as is the case of modular programming. Otherwise, based on those interfaces, more detailed level data flow is considered. When a simple data flow diagram is drawn (e.g. in ISO 7 layer protocol model, cascading layer-(n), layer-($n+1$), ..., layer-(m)), each function is realized by a process or divided further. Further decompositions must be made by paying consideration to make each fraction be mono-functional with clear interfaces. This kind of decomposition results in better testability with smaller software in overall size, as each component is mutually independent from the rest.

Human short term memory structure that can simultaneously process at most 5-7 individual concepts (i.e. magic 7). To avoid the conceptual overflow, the number of functions in the function decomposition must be limited to no more than 5-7.

The high level interface concept is an abstract concept, and as the decomposition is

carried out, interface concept becomes gradually more clear and detailed. The high level abstract interface concept must be maintained in the detailed interfaces to keep the functional integrity of each component processes or blocks. (In Structured Analysis, by comparison, only hierarchical structuring of data is allowed.) There are several difficulties in doing so:

- A sequential process has both input and output, so that both way interfaces are very clear in some cases. In such a case, a component is divided to the one of input side and output side.
- Messages convey two kinds of information, an event (or trigger) and other data associated by the events. At higher level, one of them is conspicuous and as the design is elaborated, the exact form of messages must be drawn and detailed interfaces must be determined.
- Coupling between components is an important problem. At the top in the system level as shown in Figure 4, information from a part to the other may be delivered in two ways. One is to deliver all the information via messages and the other is to share common memory area. The latter sacrifices the dependence of parts. This is the same problem in module design for non-sequential programs.

At the top system level in Figure 4, the substance of system is usually empty without any problem. At the bottom, the lowest level process interfaces external ports in a hierarchical manner. In the middle, there are two ways of coupling, empty case and a process coupling case, where a block has its logic performed by a process as shown at the bottom of Figure 4. When a function is performed by too many lower level processes, it becomes hard to keep track of the design, and harder check and test, consequently. In this case, it is a better policy to provide a central process in each level, and to make lower level functions correspond to such a process.

4. PROCESS LEVEL DESIGN

In this section, design ranging from external behaviors to state diagrams is discussed. Sequence charts have been found useful. Figure 5 shows the steps.

Sequence charts are widely used in industry for protocol-based routing and signaling and are familiar concepts to people in these fields. Sequence charts stay naturally above the state transition diagram layer. Although there are many versions of sequence charts and advanced Message Sequence Chart (MSC) tool is a part of SDL body, here a classical sequence chart is used during the discussion.

In Figure 5, diagrams from A to C show gradual embodiment processes. The design starts from a simple sequence chart for actual external behaviors, as shown by diagram A in the figure. It is converted to a multiple process sequence chart as shown by diagram A1. Each vertical axis corresponding to a sequential process and may be extracted as shown by diagram B, and it then is converted to state diagram shown by diagrams C1 and C2.

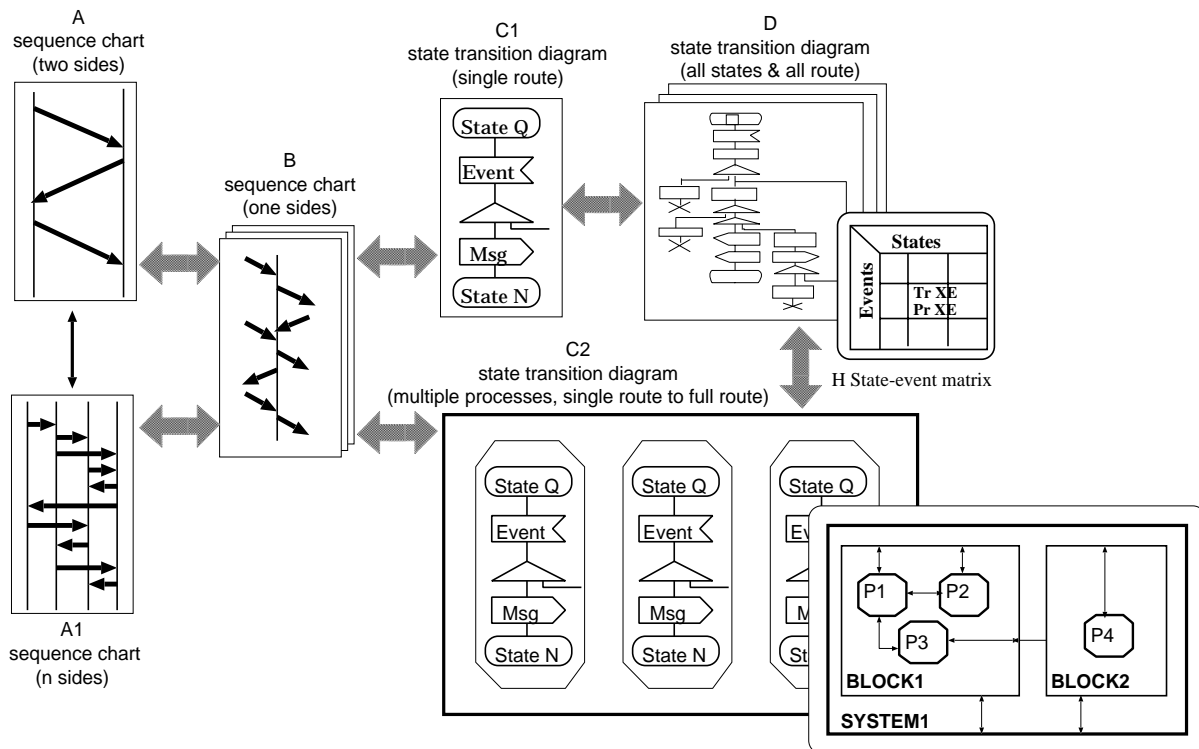


Figure 5. Sequence to process

The starting point is to write a sequence chart between two or three sides with the most abstracted or the most far signals. If any disagreement in conceptual level among signals exists it injures the integrity of the process. Therefore Careful check on both sides signals is needed.

The initial series of sequence charts should cover a typical service and entire operation, such as a call. A sequence chart can describe only one route of possible external behaviors. However, other typical or even unusual operations should also be written down. Summing up these sequence charts, it is frequently found that these charts may be further sectioned horizontally to several fractions such as initial, middle and final groups. Detailing to more detailed signaling sequences may be performed on each of them.

In most SDL design, a system is realized with several EFSM's. In this case, a simple sequence chart between two sides may be transformed to a sequence chart intersected by one or several processes as shown by diagram A1. If the external signals are already defined, the integrity of the interface processes should be sacrificed to ensure each internal process has conceptual integrity. Signal names and each process concept should be examined closely.

Before fixing the final design of this stage, the concept of intermediate processes must be determined. In this process checks are made that the structure of the interface concept is not deformed, and that minor signals for communicating locally are not missing. The difficulty increases as the number of intersecting processes increases. A virtual mono-functional process should be taken to decrease the complexity. The vertical axis corresponding to the virtual process is vertically subdivided several vertical axes, and the necessary messages must be added.

5. PROGRAM LEVEL DESIGN

Design steps in this section are shown in Figure 6. The design starts from elementary state transition diagrams as shown in Figure 6. A state transition includes decisions (branches) and message outputs, but lacks tasks to perform possible state transition. Exact definitions of each state must be given as shown in Figure 6/B. If the system has been broken down to mono functional EFSM's, each state relates to a few resources. The figure shows pictorial state definitions like classical SDL, but current SDL does not include such application specific feature. Designers must prepare a clear document for each state describing all resources with their states.

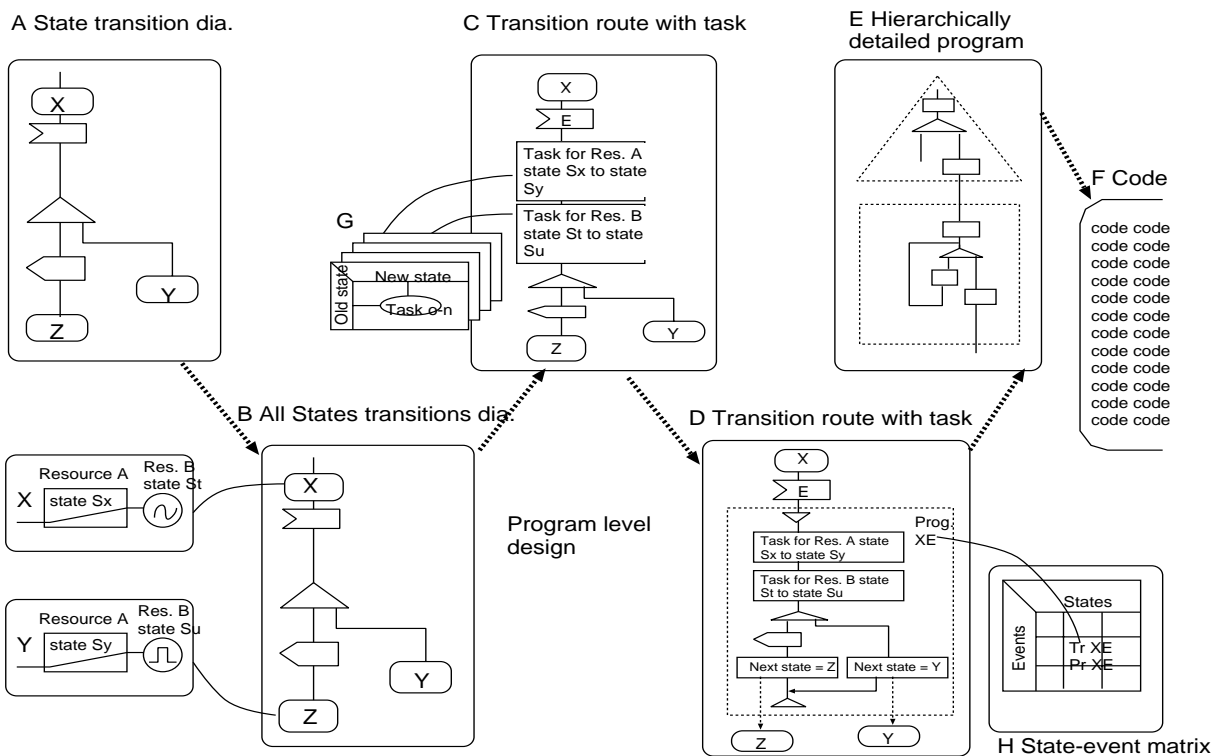


Figure 6. Program level design

Using pictures or tables describing related resources with their states, tasks, etc. is practical. The difference of states from the initial state (before a transition) and the new state (after the transition) indicates task(s) to be inserted during the state transition. Figure 6/C shows the transition with tasks inserted. They are shown by tasks table G in the same figure.

Thus completed state transition, however, might have several exits, which contradicts with a good programming practice. The transition is converted to a single I/O program by inserting a next state definition at the end of each path as shown in Figure 6/D. With a mechanism explained later, the program shown by solid line performs the same state transition shown by dotted arrows.

Thus separated programs receive hierarchical detailing as shown in Figure 6/E and they are converted to codes as shown in Figure 6/F. These procedures are the same as ordinary

program design.

Important steps for attaining high quality are,

- Design rigorously at each small step of design,
- Provide the exact information needed for each step, and
- Check rigorously at each step.

This discussion shows that a really useful CASE tool should have at least the following features:

1. Possibility of integrating and handling sequential nature of design at higher level and non-sequential nature at lower level [12].
2. A quality oriented CASE tool requires various other features such as
 - pointers for displaying the previous and the current diagrams, and
 - pointers for displaying and referring to various other tables and diagrams.

6. CONTROL STRUCTURE

Figure 7 shows the principle of the control structure for state transition programs. As conventional system software has been designed for non-real time and non-sequential programs, so it is desirable to provide a matching control structure for a real time embedded system.

The left side of Figure 7 shows the principle of state transition control, which is an essential part of an operating system. The central part is a state transition table, which is a transformation of the state transition table in Figure 5/D. As all the state transitions have reduced to single I/O programs, this two dimensional table contains inputs to all of them. State is obtained from the state store by indexing process number derived from a message. Event is obtained from the message.

The right side of Figure 7 show the detailed control structure. Clock triggered input programs and other input programs pick up external inputs, assemble a message and enqueue to the lower level input queue. The execution control scans the high priority queue then the lower level queue, and on finding any message it dequeues it and performs state transition as described before.

Thus triggered state transition program sends out several messages to other processes. They are enqueued to the high priority queue, for non interrupted operation triggered by an input.

The state transition programs are not allowed to contain any delay function. Delay function brings about various program bugs, brings difficulties in designing the control structure. Instead of delay function, timer process is provided and state transition programs issue the timer process timing request. The timer program updates each timer provided for each timing request. On reaching the final count of the timing, it assembles a message to the client process alerting the timing.

All messages follow a standard format, which result in a common idle queue of the large enough size. It eliminates calculation and adjustment of each queue for the busiest

period. In the message format, destination process and event are also standard, making the state transition control simple.

This type of OS is a tiny program, but very effective for any embedded system. For a high speed or low overhead operation, it is implemented by assembler language. With some considerations, it can accommodate already existing system software.

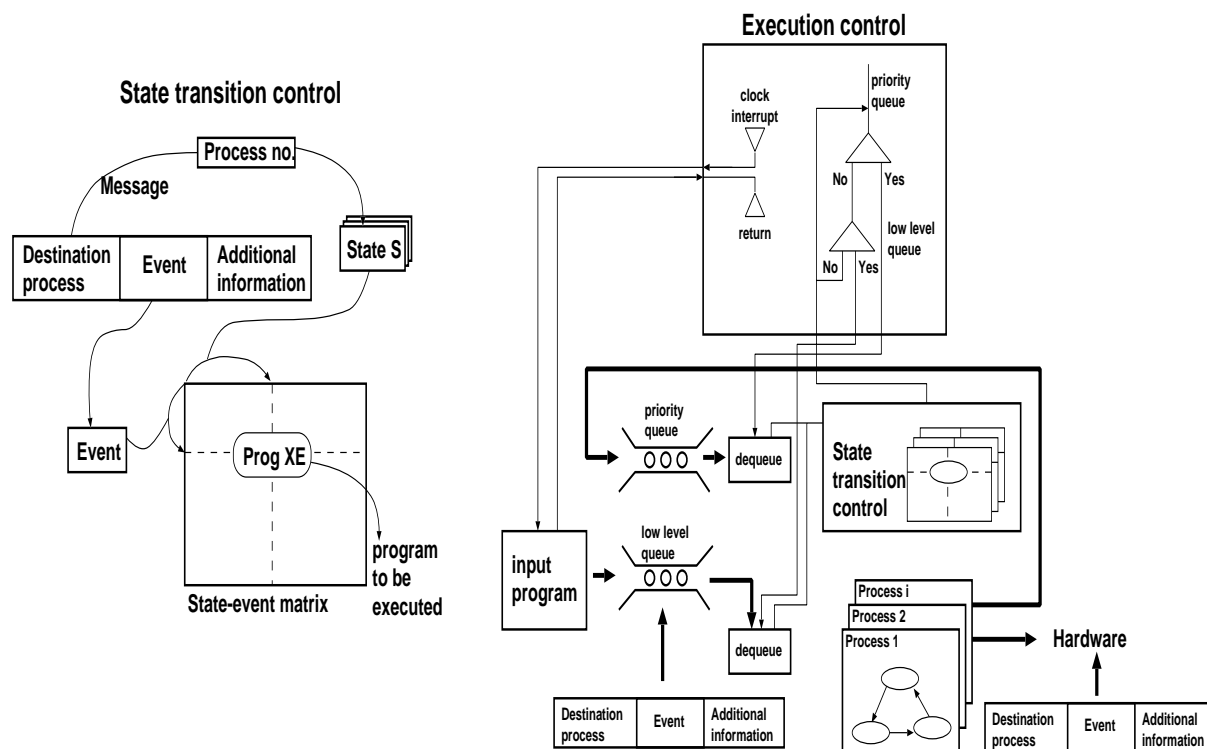


Figure 7. Control structure

7. CONCLUDING REMARKS

In this paper, authors' model for build-in and check-out of software errors, based on the structural design procedure for embedded systems have been explained. The model explains well various industrial practices and empirical experiences.

This procedure has been put into practice by making experiments in developing skills of software design and education. Each team of three to four students is assigned to develop an virtual embedded system consisting of two to four processes. Most of them selected some kind of vending machine, which they are familiar with by the preceding education. Team members discuss their product's external functions, write following documents and review, program design and unit testing, process level testing and finally system level testing. They also report errors found during machine testing as well as root cause analysis and preventive means not to repeat the same kind of errors as is done in TQC. As they are beginners, special emphasis is put on program level design to write the final state transition diagrams and structured flow charts in IPO style. Most high level

errors are checked out by desk checks and reviews, and one to four errors are found during process level machine testing and the final system testing.

Acknowledgment

Authors wish to express their gratitude to Telecommunications Division, Hitachi, Ltd. for supporting this study and their former colleagues for clarifying and summing up various ideas. The authors are also grateful to those who participated in this study and contributed to it.

REFERENCES

1. Y. Hayashi, "Human Reliability Engineering," Kaibundo, 1984 (in Japanese).
2. Z. Koono, K. Ashihara and M. Soga, "Structural Way of Thinking as Applied to Development," in *IEEE Global Telecommunications Conf., GLOBECOM' 87*, Tokyo, Japan, 1987, pp. 1017-1022.
3. Z. Koono, T. Kimura, M. Iwamoto, and M. Soga, "A Stored Program Controlled Environmental Function Tester Based on FMM/SDL Design," *International Switching Symposium*, 1987.
4. Z. Koono, Y. Yonezu, Y. Iwata, and M. Hosoda, "Experiences in Applying SDL," *SDL '89: The Language At Work*, O. Færgemand and M.M. Marques, eds., pp. 395-404, Elsevier Publishing Co., North-Holland, 1989.
5. Z. Koono and M. Soga, "Structural Way of Thinking as Applied to Systems Design," in *Proceedings IEEE Global Communications Conf., GLOBECOM' 90*, December 1990.
6. Z. Koono, and E. Harada, "Structural Way of Thinking As Applied to The Design of Embedded Systems," in *Proceedings IEEE Int. Conference on Communications*, Chicago, USA, June 1992, pp. 1680-1687.
7. Z. Koono, and T. Ohtsubo, "Evaluation of Built-in and Check-out of Software Errors," Technical Paper SE-95-5, IPSJ, 1993 (in Japanese).
8. Z. Koono, and B.H. Far, "Structure of Software Development Process From a Viewpoint of Industrial Software Engineering," Technical Paper SE-98-5, IPSJ, 1994 (in Japanese).
9. Z. Koono, and B.H. Far, "Quantitative Design of A Development Process," *Proceedings 4th European Conference on Software Quality*, 1994.
10. B. Kurahara, K. Uchimaru, and S. Okamoto, "Total Quality Control in an Engineering Group," JUSE Press, 1990 (in Japanese)
11. G.J. Myers, "Composite Structured Design," Van Nostrand Reinhol, 1978.
12. O. Shigo, M. Norifusa, M. Koyamada. T. Kishi, M. Sugai, N. Hagiwara, and H. Hanrin, "Software production System for Communication and Control Software," NEC Technical Journal, vol. 47, no. 1, 1987.
13. A.D. Swain and Guttman, "Handbook of Human Reliability Analysis with Emphasis on Nuclear Power Plants," NUREG/CR -1278, SAND 80-0200, 1983.