New Trends in Software Methodologies, Tools and Techniques
H. Fujita and I. Zualkernan (Eds.)
IOS Press, 2008

doi:10.3233/978-1-58603-916-5-126

The Ultimate Systems Development Method Based on Finite State Machine

Zenya KOONO[a,1] and Hui CHEN[b]
[a]*Representative, Creation Project*
[b]*Information Science Center, Kokushikan University*

Abstract: This design method stands upon three bases. The first is to use a finite state machine (FSM) model of around three states, where the design is very easy. The second is to use an event driven OS, which enables direct execution from the specification level to the final implementation model. The third is the hierarchical architecture of the above-mentioned FSM's, which minimizes the software size. This method features high productivity and high quality. Although it is a development for embedded systems, it may be applicable to any systems as an ultimate design method.

Keywords: Embedded system, Systems design, Hierarchical decomposition of concept, Software size, Documents, FSM, Event driven OS, SDL, Automatic design.

Introduction

The purpose of this paper is to propose an ultimate development method for primarily embedded systems, which may also be used for systems in general.

An "Embedded System" is the name for a type of product, which was formerly a hardware product, but is currently controlled by software. An example of is a "digital camera". Formerly all cameras were mechanical hardware including their control, but now most cameras are "digital cameras" controlled by software. As this trend grows rapidly, it has become one of the main topics of discussion in the software field. As for the implementation technologies, however, there are various views and much confusion.

Recently, experts acknowledge that "applications of Model Driven Architecture (MDA)" are a standard process for embedded systems. The author's view is close to this, but more fundamental and more extensive. Both are based on a "finite state machine (FSM)" model, with which most software people have not yet become familiar. As this paper reports procedures for defining this method, it may be a good opportunity for readers to integrate this with what they already know of MDA.

Section 2 explains the specialties of embedded systems and, outlines the concepts of this process. Section 3 explains the procedures involved. A design in repeating hierarchical decomposing is the prerequisite. One of the key ideas, the design procedure for an elementary FSM of around 3 states, is reported in 3.2. In order to show the best practice and understand how to be automated in future, the detailed

[1] Corresponding author: Zenya Koono, Representative, Creation Project, Honfujisawa 2-13-5, Fujisawa, Kanagawa 251-0875, Japan; E-mail: koono@vesta.ocn.ne.jp

procedures are disclosed. The second key idea, the design of the hierarchical architecture, is described in 3.3. The third key idea of an event driven OS is introduced in 3.4. All these reported here are best practices, and have been accumulated through repeated developments using FSM's, and have been applied in the education of around 100 teams of university students. Section 4 describes the main results. Section 5 is discussions on this method.

2. Design Philosophy of this Development Method

2.1 Specialties of Embedded System

There have been many discussions on the specialties of embedded[2] systems. There one has yet to be discussed. *Embedded system suppliers are working in a free but competing market.* [1](Cf. usual software vendors have not been working in such free and competing markets.) Managements are eagerly pouring excellent people and R&D investment into each field for achieving the higher technology needs to win this competition. Also, all hardware engineers have improved their technologies in each field, and have attained great expertise therein.

There is another substantial difference between the embedded software and the usual software. Most software has a fixed input-to-output relation; therefore it may be said to be "combinatorial logic". An embedded system is usually for the control of hardware with various states; therefore *embedded system software is "sequential logic".*

Any hardware logic design textbook says that the "sequential logic" part must be implemented using memory for storing states, and the rest may be implemented by "combinatorial logic". It is the same also in software. Embedded system software needs *modeling by Finite State Machine (FSM).* Usual MDA uses an automatic generation for implementation in current OS, but this system uses *an event driven OS, and thus it becomes possible to execute from the specifications to the final code directly.*

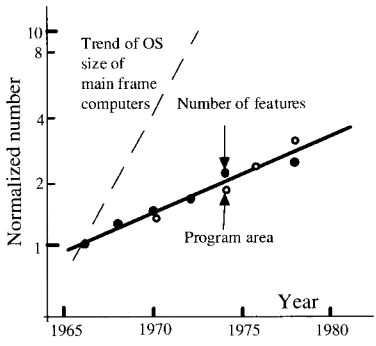

Figure 1. Increasing trend of software

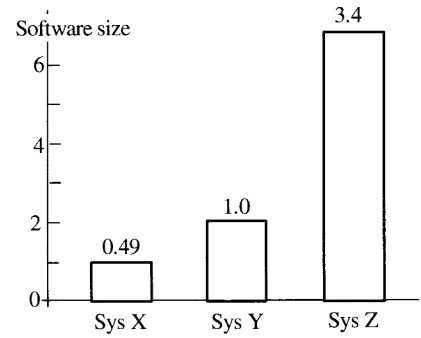

Figure 2. Software size

Let us examine a technical trend. Figure 1[2] shows the increasing trend of the software size and the number of features from a paper on the world's first large

embedded system No.1 ESS[2][3]. The horizontal axis shows years and the vertical axis shows the software size and the number of features both on a normalized scale. Both graphs show the same trend of approx. 10% increase/year. The finder, Koono, found that this appears both in hardware and software, and the additions of mother bodies of major functional enhancement (Data Base and Data Communication) in the OS caused the rapid growth trend shown by the dotted line. This finding stimulated NTT, and they also found that their communications software showed the same trend. Thus it was accepted as a general trend of these technologies. In the middle of the 1990's, (namely to increase $1.1^{30} = 17.5$ times), the software size of most major switching systems had reached several mega lines, following this trend. This increasing trend still appears now.

The figure shows following:

If the size of the mother body is large, it not only invites a large initial development cost but also a large updating (so-called maintenance) cost.

Software size is a crucial factor in software. Figure 2[4] shows the sizes of the central part of the control software of the three systems of a kind of embedded system. The differences are as follows:

* Software sizes, normalized by Y, are Z=3.4 : Y=1.0 : X=0.49 respectively.

Probable causes of each:

*System Z project had been developed, following the client's request, by software people not accustomed to this field.

*System Y project was conducted by an experienced leader in this field.

*In system X project, "the large software" had annoyed people previously, and they adopted "mutually independent FSM's" for a substantial reduction of software size.

Most characteristics of human work distribute in lognormal distribution, where it ranges from 1/3 times to 3 times of the mean value respectively[5]. The above samples correspond to the largest, the mean, and the smallest respectively. *The software size is a reflection of the organization's technical ability* accumulated in order to compete successfully.

2.2 Design Concept of this Development Method

Major design concepts of this method[1,6] are listed below:

The strongest motivation is to compete successfully. The organization has to adapt the organization and take strategies to succeed. The usual organization provides a profit center for each business. In an embedded system, in order to win, the "systems engineer" group is the center and exerts the greatest power on the following "software" group and the "hardware" group exerts pressure on the production group, but it takes the all responsibility of the business. Therefore, "systems engineer" is enthusiastic in R&D, and invests a lot effort both in design automation and manufacturing automation.

---

[2] In 1965, the world first computer controlled telephone-switching-system (called No. 1 ESS) started services replacing a huge mechanic-electric telephone switching system. It was the world first large embedded system, which Bell Telephone Laboratories, AT&T (a giant in communications) developed. In the same period, IBM (a giant in computer) began to ship System/360, and American Airline (a huge computer user) started a world first on-line seat reservation service. These were world level topics at that time.

The following are some strategies, of which this paper discusses the major points:

1. Minimize the cost, which is (largest productivity) with (minimum software size).
   In hardware, they use (the least numbers) of (the cheapest components). Similarly to this, they require (the least software size) in (the easiest way manufacturing).
   *In this method, *software is basically FSM, and an event driven OS,* named Midas[23] is used for controlling FSM software.
   *In FSM, *the easiest to design is an elementary FSM of around three states*.
   *In order to make the software size the minimum, *the system is all composed of mutually independent elementary FSM*, similarly to "combinatorial" software.
   *In order to develop the software after the "systems engineer" group, various software design automations should be taken.
2. High quality throughout the products' life[5].
   * *Leave accurate and correct documents to cover all the necessary areas.*
   * *Design in each small step and do strict, rigorous and careful checks.*
   * Rational ways of testing.
3. Strategy for advance
   *Evolve to include all the design
   * Enhanced R&D, including evolving design automation.
   * Feedback so as not to repeat the same errors.

Those aspects not discussed here will be mentioned later in other sections.

3. Core Technologies of This Method

3.1 Repeating the Hierarchical Decomposition of the Concept

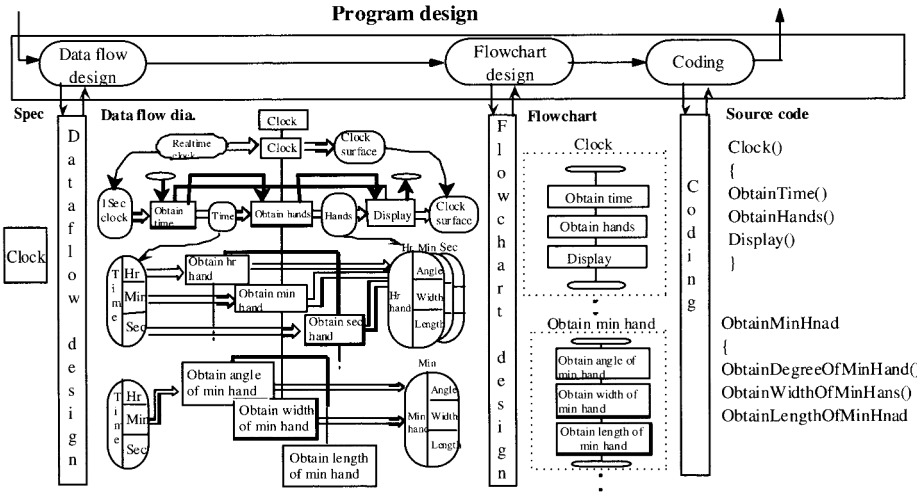

Figure 3. Design records of clock program

Design in this system follows the principle of repeating the hierarchical decomposition of the fundamental concepts for intentional activities. It is a functional model of human cerebrum in its best mode as published in SOMET06[8,9,10].

Figure 3[10,11] shows the design record of a "clock" program. It is the combined use of graphic and natural language (hereafter, this includes scientific and engineering terms). The rectangular box with a semi-circle at the top and bottom shows the data, and a square box shows the function. The data flow begins with data, and repeating function and data, ends with data. The topmost section of the data flows on the left edge is the specification, "clock." By adding data at both sides, the second section is *an elementary data flow* of the "clock," which is *a parent concept*.

It is decomposed to form the hierarchically detailed data flows below, consisting of the *three elementary data flows of "obtain clock", "obtain hands" and "display"* in a serial manner[3] as *the children concepts*. (Here, a flowchart starts from a compressed barrel symbol on "obtain time." It is also shown in the center of the figure, and the source code lines, corresponding to it, are shown on the right side of the figure. These are specialties inherent in software, where a control flow is needed.) Next, among the children, that for "obtain hands" is shown in the next level data flow, which is decomposed to three data flows in a parallel manner.

By thus repeating hierarchical concept decompositions, each concept becomes simple enough. In the figure, an example of "obtain angle of minutes" is shown. Then it is expressed by the source code of a programming language as an implementation mean. *These repeating hierarchical decompositions are the essential operation in design.* The authors developed an automatic software design system as reported in SOMET06[8,9].

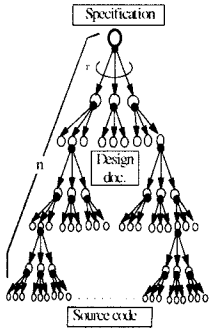

Figure 4. Hierarchically expanding network

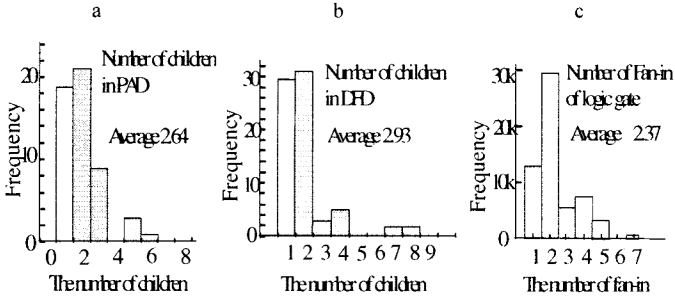

Figure 5. Expansion rate

A hierarchical network, shown in Figure 4[7,11], models the entire operation. Eventually all the expansion rates in Figure 3 are all 3. But the average or the optimum seems to be a little less than 3, probably e=2.7182... Figure 5[11,5] shows several field

[3] Myers' STS division [12]. When an elementary data flow may be regarded as a flow of information conversions, the optimum intersection is to divide it to three mutually independent parts, by two *abstract* data, named the most *abstract* points (MAP). The input side MAP is the most distant concept from the input still, still keeping the color of the input, while the output side MAP is the most distant concept from the output, still keeping the color of the output. The input side is called as Source, the center is Transform, and the output side is Sink. As they are independent, this gives the minimum software size partitioning. Although Myers applied this only at the highest first level, it is extended to be used anywhere when possible.

data of software design (a and b) and logic design (c). As the authors reported in SOMET07[11,5], the various characteristics stem from the human characteristics. The average expansion rate is another one of them, and it is the optimum point for a human when the person involved feels it to be the best.

These show that repeating hierarchical decompositions is a function of human cerebrum for intentional activity, and following are the best practices of design[11,5].

1. *Decompose a concept into a small step of design as expressed by natural language, using data flow (and flowchart or equivalent if needed).*
2. *For the checks, leave design documents, recording each of the detailing steps.*
3. *After each small step of design, do desk checks carefully, rigorously and repeatedly on what was made during the preceding design and recorded in the documents.*

The principle in this design, for making the software size small, is as follows:

1. Decompose a function, in a serial manner, so as to make the parts mutually independent. The software size becomes minimal.
2. Common parts: All parallel data flows are similar to each other and thus not mutually independent. But, if common parts are found, they may be decomposed to mutually independent common parts and the calling sequences.
3. Quantitative evaluation and research: In each design decision, choose the minimum software size (the logically simplest) case. If no good solutions appear immediately, let it be studied as a research problem.

The size reduction might invite some troubles. If only the software size reduction is obligated, bad methods that might undermine comprehensibility might be taken. As most vendors count the software size as a measure of the sale, the size reduction might appear to them to lead to a drop in sales. It is important to share the same understanding that it aims at decreasing people's efforts, just as a muscle works less when habitually doing more creative work.

3.2 Sequential Logic Part one, an Elementary FSM

3.2.1 Basic Concepts of FSM

In this section, "sequential logic" for an elementary FSM is discussed. After an introduction, design of a vending machine as an example progresses to an elementary FSM in a step-by-step manner in the detail of both design and desk checks, and then an expansion to multi FSM is made.

The well-known 'traffic right' case in Figure 6 shows the basic concept. In Figure 6.a, it lights 'green', then 'amber', and 'red' in each interval, and then it returns to 'green' to repeat the same cycle. Each of the three states is a *stable state*, they *operate exclusively* forming *a closed cycle*, and thus they constitute *an elementary FSM, corresponding to a concept.*

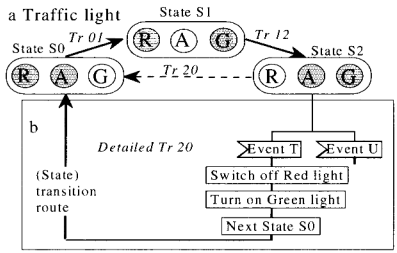

Table 1. Event (command) structure

Event		Event	Auxiliary info.
	Ex. 1	Coin inserted	Amount
	Ex. 2	Specify item	Item no.
Command Message		Command	Auxiliary info.
	Ex. 1	Change	Amount
	Ex. 2	Eject item	Item no.

In Figure 6.a, each arrow shows a transition from one state to another state. They are called *"(state) transition route"*, and is named here $Tr_{\text{old state new state}}$. The input, causing a state transition, is called an *"event", which is a spike-like signal.* A state transition starts at the arrival of an event, and it runs through to the end, and then the state terminates.

In this system, when an input program detects an external event, it prepares a standardized packet, including the event as shown in Table 1. Usually an event (also a command) is accompanied by some *auxiliary information.* One example in the vending machine, an event by the insertion of a coin accompanies the auxiliary information of "25 cents".

Figure 6.b[4] shows a detailed transition route of Tr_{20}, using SDL[4] symbols. A compressed barrel symbol (State S2) shows the state. A wedge symbol under it shows an event. In SDL, a route is written in a flowchart manner, and it may include one or more functions (e.g. processing) and also one or more branches. It is assumed that any transition route is executed without any stops or delays. During Tr_{20}, the hardware changes from state S2 to state S0. The event T triggers the transition route Tr_{20}, and performs the functions "switch off red light" and "turn on green light" that are derived from the old state and the new state. There is a next state definition "next state S0" before the termination. Note that *functions in a route may be systematically obtained from the differences of the initial state and the terminating state*[15]. They are a "combinatorial logic" part, and the previously mentioned three states with each transition route are the "sequential logic" part.

[4] SDL comes from CCITT recommendation "**S**pecification and **D**escription **L**anguage". Vaughan's paper[13] on Bell Telephone Laboratories' research model showed to specify the switching sequence using FSM model. Japanese engineers including Koono confirmed the technique with each companies Laboratory model. Based on its usefulness, NTT and manufacturers' group used it for the switching system's specification description. Thus, it was standardize CCITT recommendation SDL in 1976[14]. CCITT is an organization under United Nations for standardization in the field of communications. For system specification, ISO pushed LOTOS, and CCITT pushed SDL. As SDL has been used by many in the industry, it evolved to a formal definition with both graphic and textual representation in late 1980's. It is technology transferred to UML, which evolved to UML Version 2.

3.2.2 Vending Machine[6]

Let us examine the design procedures of a well-known vending machine to fix an elementary FSM. Figure 7.a shows the simple outline view, on an IPO (Input Process and Output) form. IPO form [16] is best suited to write data flows. It shows the external inputs in I (Input) column and external outputs in O (Output) column. Instead of processing in the center, the outline view is written in P (Processing) column in Figure 7.

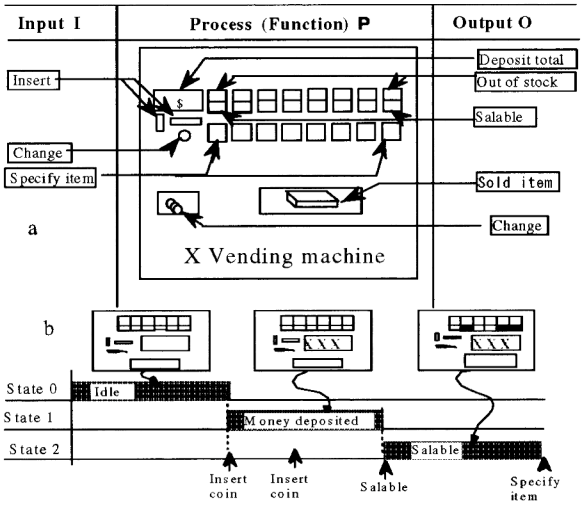

Figure 7. Vending machine

It must be written to meet the following requirements.

1. All of both input devices that generate events, and, all output devices that are actuated by commands, must be shown by the outline view. Others that relate to the hardware status controlled by software must be shown.
2. *Systems engineers should examine both the outline view with and the related hardware documents so as to be able to remember the hardware parts*. This is to be a key to remind the reader of all the related hardware.

The next problem, fixing the top most FSM, is due to the fact that there are various viewpoints among software people. As it defines the overall system,

* *The focus should be on the system itself*

* *To show the most abstracted behavior of the system,*

which "money is inserted to it, and it outputs the specified item",

* *With each state defining respective hardware resources' states,*

by the name "Vending machine (system) X", which

* *Corresponds to the concept of the FSM.*

There are two viewpoints. One is of the vending machine itself and another is the item to be sold. As it denotes the whole system, the vending machine is better. (In some cases such as a belt conveyor case, both the system and the item being transferred are needed.)

In fixing states of an FSM, it should be remembered that these three states are the top most hardware and software interface. Notice what is remarkable and characteristic. Figure 7.b shows a viewpoint defining the three states, with a time chart. The small figures shows what the designer noticed. At first, it is the initial state. The next state is a state when "deposit total" is displayed. The last state is a state when one or more "salable" item is displayed. These three states are stable, exclusive of with each other, and they form a closed cycle.

As this is a simple and education system, some symbolic figures are also shown. Strictly speaking, a "state definition diagram" should be written for defining the hardware state for each state if needed. All resources relating to each state are listed up first, and each state definition diagram must include each state of all the resources. Note that, when the initial and the terminating states are defined, the state transition

route program must perform to realize the differences[15]. If they may be simplified, it is preferable to use some symbolic graphic representation. Note that even if it is simple, it is not a comic but an *official specification* of hardware and software.

The next is to define all the inputs. *Examine if FSM's name (concept), the states and the events conform to each other.* An event (or a command to hardware) is transmitted in a standardized format shown in Table 1. Auxiliary information is sometimes the mainstream data in the "combinatorial" logic part.

These specifications define events (commands), as *official contract or agreement documents, and never a dictionary.* Catalogue figure/diagram shows an overview of the structure preferably graphically (e.g. Table 1), and there are hierarchical tables for defining each event (command or data) and each piece of auxiliary information. The definition starts from a definition of a data name, and goes down to lower level fields in a hierarchical manner. As many people refer to these, each description must be accurate, strict, clear and short. *It should be written as if it is an article of a law*[5]. The description also includes the data length, data type, initial value and so on, as well as the occupying data space area.

After a specification is delivered to the software group, the aforementioned information and all the symbolic names are written next to the natural language name in a parenthesis. Outside of the software group, as all peoples speak natural language, *the use of symbolic code names is inhibited.*

3.2.3 State Transition Route

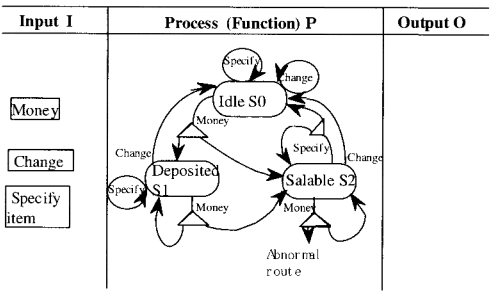

Figure 8. Transition route with decision

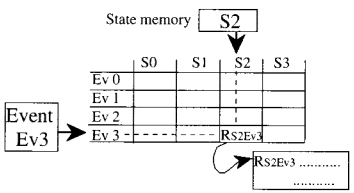

Figure 9. State-event matrix

The next first stage is to draw a simple state transition diagram as shown in Figure 8 [6], referring to the state event matrix in Figure 9 [6]. The state event matrix is a two-dimension table listing all states and events. At a cross-point of state (e.g. S2) and event (e.g. Ev3), a transition route name is recorded. A name here is as (e.g. Tr_{S2EV3}). The purpose of writing a simple transition route diagram is to focus attention on the transition routes only. In practice, write the event name near the origin, and also, for simplicity, write each transition route name along each route as they are deleted.

[5] In order to keep every group's quality level high, it is recommended to charge the loss resulting in the following processes to the group that made an error. The judge should be made based on related documents' descriptions. Each group is assigned the budget for the loss, and managements review each group's percentage of the total/budget. All groups are obliged to decrease their own works as well as what resulted in related groups. Historically, hardware groups had been improved through these strict rules and the successive improvement activities. The same approach should be taken also in software.

First, look at the outline view in Figure 7, and notice on the hardware input in column I to find an event. Referring to the event specification, and with the state pointing to the state event matrix, name the transition route name and write it in the cross-point of the state event matrix, and draw the simple route without any functions as shown in Figure 8.

A transition route might terminate the plural termination states. In such a case, a branch is inserted. An event without any function is written as a circle, starting from a state and retuning to it. When all cross-points of the state event matrix are filled, is the end. If each transition route is provided, the system endeavors to work on any operations that are applied.

Figure 9 also shows a principle of an event driven OS. The OS has each state memory (now S2), corresponding to each FSM. Now, an event (Ev3) arrives. Using the state event matrix, the transition route R_{s2ev3} is found, and the OS executes a program for R_{s2ev3} as shown below. As it includes the next state definition before the end, the OS updates the state in the state memory before it terminates. It is very simple and it can directly execute the transition route program, from specification level to the final program.

In this method, *neither interruptions nor delays should be included in a transition route* for preventing possible errors. If needed, an additional internal state may be provided for the pause, and provide an event or the equivalent for resuming the operation. For timing, a timer process should be provided and the timing information is sent to this. After the timing has passed, it sends back the time-out event.

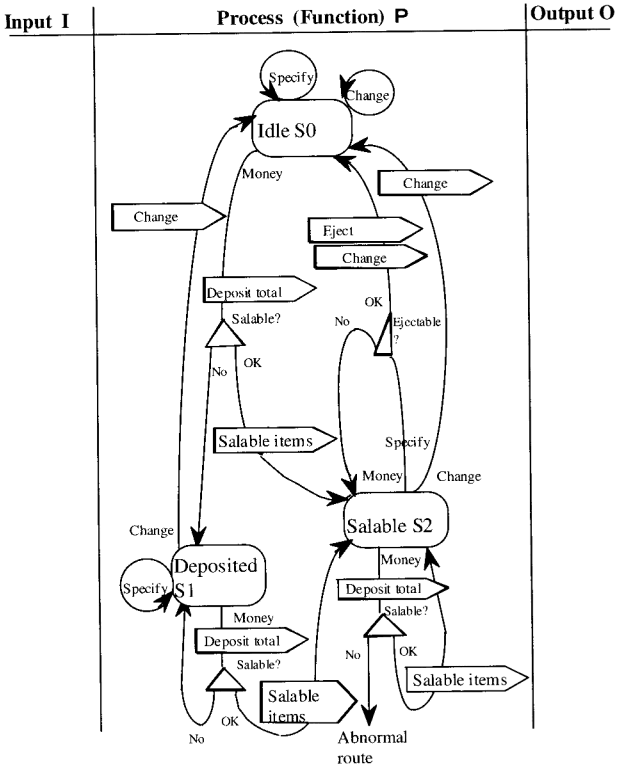

Figure 10. Output inserted transition diagram

The second stage is to insert all the outputs on each transition route as shown on Figure 10. Differences of states create necessary hardware command(s). A command is essentially one way except for immediate response (e.g. as in-operable, OK/NOK), and executed in a very short time.

Keep the completed simple transition diagram as shown in Figure 8, make a copy for this second stage. Do not

rewrite for preventing errors, but instead enlarge and re-use the copied diagram. On this copy, all the outputs should be inserted. The resultant diagram of the second stage must be written strictly in *ONE A4 sheet size*. (If it is written in multiple pages, the error rate surely increases. As a person errs unknowingly, one thing to do is to take various countermeasures so as not to invite errors.)

A design is a detailing from an input document to the output document (or from a part of a document to the detailed part). Both documents of the input and the output must be left, and show the change clearly. *Errors arise during this design between documents. The next desk check is to check strictly and rigorously what was made in the preceding design[17,18].* If some relates to hardware, do not stay within the systems but go one step inside of the hardware to examine the related parts (e.g. hardware sensors and their characteristics.)

In Figure 10[6], outputs in picket symbols are inserted along each transition route of the original simple transition diagram. This principle has been already explained using the example of the traffic light in Figure 6.b.

1. Prepare diagrams showing each hardware state.
2. Do the following procedure for all transition routes, keeping the diagram to *be one A4 sheet*.
3. Pick up a transition route. Based on both hardware state diagrams, differences of hardware states are listed, the necessary output commands derived, and each one insert into the transition route. The result is as shown in Figure 10. Do strict checks, and keep the detailed and make a copy for the next stage use.

If the commands are too many on any route, provide another program function for sending them out, and the transition route just calls the program function. This is made for keeping the diagram simple and able to be able to be understood at a glance. After the insertion of all the outputs, the desk check is to confirm that all the transition routes complete all the (hardware) states.

3.2.4 Completing Design of a FSM

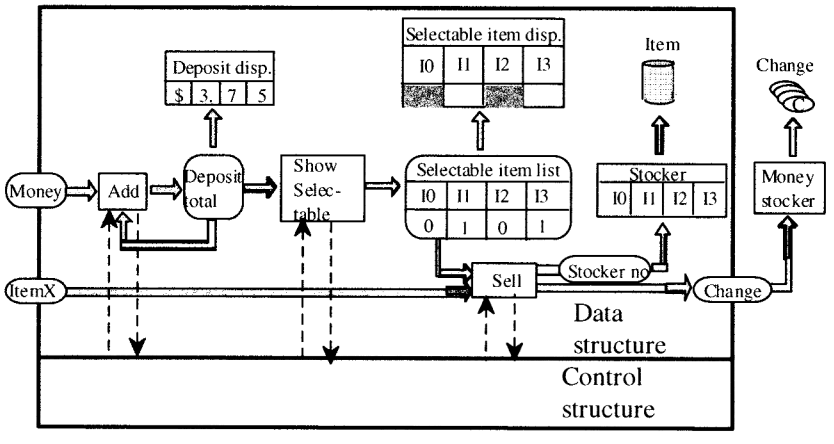

Figure 11. Data flow of a vending machine

The last works for completing the design of an FSM is listed below.

1. Insertion of necessary functions (including next state definition).

After inserting the output hardware drive requirements, the requirements for internal data processing must be designed and the necessary functions inserted. The functions inserted are super-scribed on each state transition route in the previous Figure 10. As a human cannot think outside of +/- around 10 boxes of the object being considered, use hierarchical functions to decrease the number of functions on each route, and keep *the entire simple diagram always on one A4 sheet.* Do similar strict checks as before, considering hierarchical structures.

The design of a system level data flow best clarifies the internal processing requirements. The upper part of Figure 11, named *data structure*, shows a data flow of the vending machine, from the primary input on the left side, to "deposit total", to "selectable item list", and to the final output on the right. The lower part of Figure 11 is named *control structure* of the vending machine. The both-way arrows show some transition programs calls of those functions (e.g. "Add" and "Show selectable") in the data structure.

In some applications (e.g. digital camera), the data structure constitutes majority of the software. The systems engineer (chief designer) calls an expert camera product planner and software group people, responsible for this work, and draws a hierarchical data flow of the camera. After thus decomposing the various sections, the relations between the various features of the camera, internal functions (data flows) and major FSM's become clear. The architecture is settled, and data specifications are prepared. During these, cares should be taken to minimize the size. The greater part of the *control structure* is implemented by FSM's designed by systems people.

2. Move the completed state transition diagram to diagrams on SDL CASE tool. It should be accomplished by just copying, and then careful and strict checks should be made. Figure 12[6] shows a state transition diagram of S0 for the vending machine concerned. *Arrange and write the diagram uniformly, clearly, well balanced and beautifully on the sheet. Those who can do these well will be a good systems engineers.* A completed SDL chart may be said to be an article of the constitution.

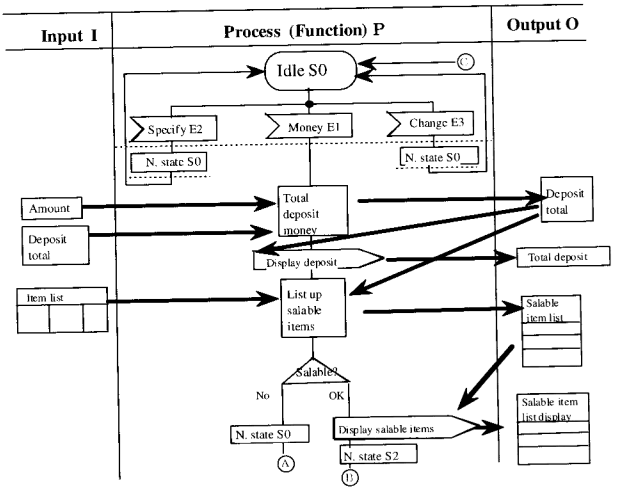

Figure 12. State transition diagram in SDL expression

In Figure 12, the SDL diagrams are in P column of the IPO. As is seen, when data in both I and O columns are written, and each data flow is shown, the operation of a transition route may be easily understood. It is suitable for the education of former

hardware people and also for machine design and checking by the design automation. As the target (e.g. around 3 state FSM) is standardized, the design and check procedures may be similarly standardized. By gradually introducing each automatic operation, the design automation system increases the degree of the automatic operation.

3. Define each transition route function corresponding to each transition route, starting from an event and ending at each following state definition.
4. The hierarchical and structured design may be made *in natural language* using a Structured chart CASE tool[6], preferably with a data flow CASE tool.
5. Fix all program specification for all the functions after enough desk checks.
6. Experimental simulation run of the system, using thus defined skeleton functions with stabs. It is made in order to prove the quality of systems engineers' works.

3.3 Sequential Logic part 2, Multiple FSM's

This method claims to organize a system by a hierarchy of mutually independent FSM's of around three states. Figure 13[6] shows the example of a vending machine organized in this manner. In human brain, Figure 13 is projected like Figure 3.

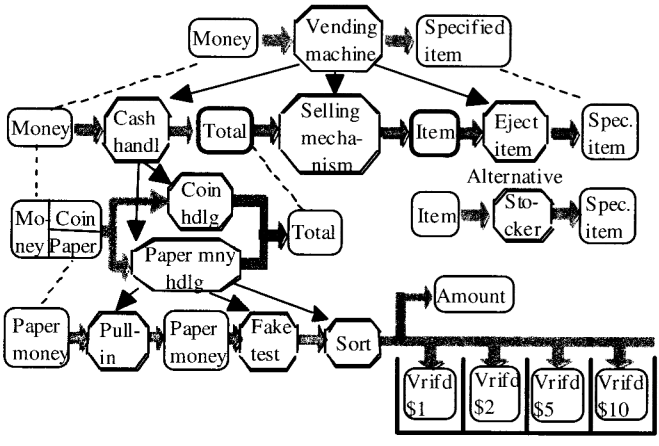

Figure 13. Vending machine with hierarchical FSM group

If the reader examines the figure, each FSM may be designed around three states, and as the level goes down, each level performs at quite different concept level. Furthermore in each level, most flows are partitioned in a serial manner to make each FSM independent, and thus the software size will be greatly decreased, as planned. Therefore, anyone may develop such a system easily.

---

[6] A structured chart is a chart fitted to show a program and to convert to source code in the final stage. Therefore, it enables a small step of hierarchical detailing and a designer can check during a design easily. It is a strategic weapon that makes Japanese high quality software. PAD (**P**rogram **A**nalysis **D**iagram) (Japanese version URL: http://www.hitachi-system.co.jp/topital) and HCP (**H**ierarchical **C**om**P**act Chart) (Japanese version URL: http://www.denso-create.jp/service/products/nnheadway/index.html and http://www.oki.com/jp/Home/JIS/New/OKI-News/1996/11/z9646.html) are two major streams.

Let us examine the design, starting from the top level of Figure 13. The top level concept "vending machine" is decomposed to "cash handling", " Selling mechanism" and "Eject item", as is in a typical Myers' STS decomposition, and they constitute mutually independent FSM's.

Automaton theory says that, if the input side FSM has x states, the internal FSM has y states and the output side FSM has z states, the system may have x·y·z states. If x = y = z = 10, and thus x·y·z = 1000. As the cost is proportional to the number of states, the three FSM case is 30 states, while the entire system is 1000 states, and thus the ratio is 33: 1. Although this is a hypothetical comparison, the substantial reduction of the size is understandable.

In the next level, "Cash handling" is decomposed to "Paper money handling" and "Coin handling", by the hierarchical decompositions of both the input "Money" to "Coin" and "Paper money", similarly to "Time" to "Hour" and "Minute" in the fourth level of data flows in Figure 3. In the next level, the flow is almost data flow, and STS division divides the flow in a serial manner, resulting in mutually independent FSM's. Therefore, the situation is quite the same as that of Figure 3.

Next, let us examine the case of "Cash" to "Coin" and "Paper money". There are many similar concept groups.

- Cash card, credit card, electronic money, points and token....
- Japanese Yen, Korean Won, Chinese Yuan, American dollar, and so on.

The same situation arises also in "Selling mechanism" and "Eject item". If these expansions were assumed from the start, the modification might be minimized. Therefore, no more explanations will be needed. In hardware, this approach will be known as a strategic business plan based on standardization.

Software productivity had been a hot research topic. Research is the prerequisite. Based on strategic research[19,20], a unified system for PBX (Private Branch eXchange) has been developed for both domestic and export uses[21]. It evolved to Central Office systems for telephone companies[22]. Firstly it was a small system, where one MPU controlled all, and then it evolved to a medium system, where another MPU was dedicated for internal processing. The single architecture enabled full product lines of products. Such wide usability exceeds the normal productivity increase. With the molecular level flexibility, this architecture is very useful.

3.4 Midas OS

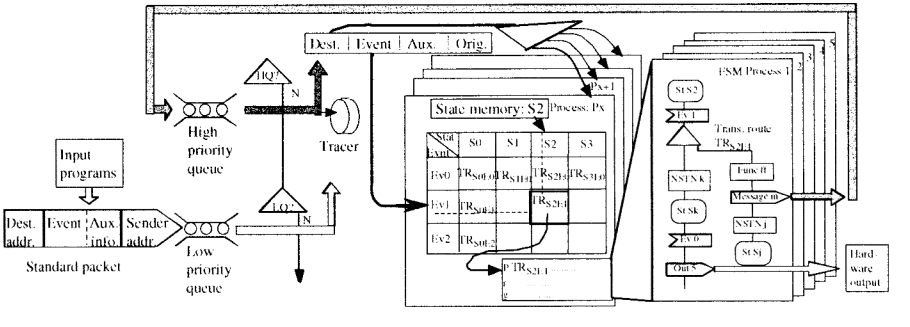

Figure 14. Midas OS

One of the authors Koono developed an event driven type OS[23,24] for highly developed multiple FSM environment in 1975. As it brings many merits to users, it was named "Midas" following the name of a Greek god Midas, who changes everything he touches into to gold. The main function is to control FSM's. It may be an independent OS or it may work under some existing OS. The following is the main structure.

In the right side in Figure 14[23], there are state transition programs for various FSM's. On the leftmost side, there is an input program for detecting an event. The output is standardized to be a packet shown on the bottom left. (A program should be designed so as to give the best-suited form to the following stage.) It has both designated address and sender address (e.g. FSM, program or hardware device etc) like an envelope, with event (command) and related auxiliary information like a letter.

It has at least two level queues of high priority and low priority. The event packet from the input program is always attached to the low priority queue, while all (message) packets for inter FSM communication, are attached to the high priority queue. This prevents confusion of processing by multiple events. By virtue of the standardized packet, they share one large common idle queue of a large size. This simplifies the work of determining the idle queue length.

When the processor is idle, it scans the high priority queue, and if nothing is found anymore, then it scans the low priority queue. If a packet is found during the scan, the OS detaches it from the queue and decodes its designated address as shown at the top right of the figure, to reach the designated FSM. There, the present state of the FSM is obtained from respective state memory. Event is extracted from the packet, and from the cross-point of state event matrix, a transition route program corresponding to the state and the event is obtained, and the program is executed as shown in the right most SDL diagram. If an output (shown by a white picket) exists, it is sent out (directly or through some queue). At the end of each transition route, the next state number definition (in the figure NTNX) sets the state memory of that FSM.

When all packets in the high priority queue are exhausted, the scanning of the low priority queue is made, and thus found new packet (including a new event) begins to be processed. The functions described up to here must be implemented to work speedily.

An endless ring memory is provided, and every packet of processed and related information is recorded. Once the system stops, the ring memory restores the traces of transition programs for some time. Thus, most of both hardware and software failures soon become clear. For these purposes, state transitions are not allowed to interrupt or delay the process, and *all data and information transmission between FSM's are conveyed through event (command) packet upon the request.*

The timer FSM is provided to return the "time-out" event to the client FSM upon the request. For a much faster response need, another operation unit may be provided to work at an interrupt level.

For fast real time applications, the length of the queue must be short. For high reliability systems, full or part of dual/duplex system is possible. As this part is a core system of any operating system has, any combination is possible. By integrating this to an existing operating system, it widens the capability of a full event driven OS. Those simple, systematic, but strict rules enable simple, speedy, transparent, reliable and easy to evaluate (e.g. processing power, various delays and abnormal behaviors) systems. Although these might be seen tight constraints for software people, they are willingly accepted them, saying that "I now understood the control structure of the system".

4. Application Results

As an application of this method, the authors' training[7] of around 100 teams (around 500 students) for more than 10 years in Saitama University is reported [6]. It was a one-month seminar "development of a vending machine", in their "software engineering" lesson for 3rd year students. The process was exactly the same as reported here.

The 3rd year students were in the maturing stage from the preliminary education to be a software people. But their programming ability is still low and most of them built-in around 100 errors/kilo-lines. The work-hours of a team were around 150 hours on average. Around the 70% of the time was used for design, and the remaining 30% was used for tests, which is a result of their low residual defect intensities after desk checks. From their reports, two points, the software size and the quality, are shown.

Table 2 [6] summarized performances of each team in the year 1995. In them, team E developed two systems (parking ticket selling and payment and gate control). Others used 3 FSM's. In the left side columns, their average software size/team was 487 lines of C code. Therefore around 150 lines of C code per FSM is the standard during the ten years. As students add various codes in programs, the net number was around 100 lines.

More detailed study was made on the team' software size, where the leader instructed team members not to write extra codes. Figure 15 [6] shows the distribution of the number of lines of C code of *transition route programs*. The horizontal axis shows the number of lines

Table 2. Quality reports (1995)

Summary of quqlity report (1995)					
Team	S/w size	No. of defects found			Defect Intensity
		T1/2	T3	QA	
A	283	0	0	1	3.53
B	438	4	4	1	20.5
C	486	3	0	2	10.0
D	357	2	2	1	14.0
E	858*	3	0	7	11.7
F	495	1	7	2	20.0
G	442	4	1	1	13.6
H	463	3	0	0	6.5

* For Parking tickets and outgoing control
T1: Test for each function T2: FSM
T3 and QA: System test including running
Defect intensity is for Kilo lines

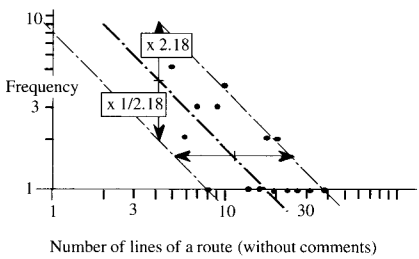

Figure 15. Route length distribution

[7] In the Department of Information and Computer Sciences, Saitama University, third year students learn core lessons of C language, operating system and software engineering, after preliminary program education for two years. The authors educated students in the general way of developing an embedded system, then the principle of FSM, an event driven Midas OS and the vending machine of three states FSM with the program. A student team consists of 4 to 7 students. Teachers appointed each team leader, considering leadership and programming skills, then an OS (Midas OS) specialist by programming skills, and others were shuffled as to all teams have a similar capability. The assignment was to develop a vending machine-like system using plural FSM's in a month, aiming at a top sales product. In order to arrange each team (from 4 to 7 students) have a similar capability, teachers appointed each team leader, considering leadership and programming skills, an OS (Midas OS) specialist by programming skills, and others were shuffled. Role-playing was applied. They were educated that a leader was a president of a company to guide and lead team members, and other members had to contribute to their own team's success. After the development, there is an open presentation, where a president made a general report and others reported each work. The education had been made from 1991 to 2001 at full scale and continued for the next 5 years in a reduced scale.

of each route, and the vertical axis shows the appearance frequency, on both logarithmic scales, and the plot is for a program. The program size ranges from 5 to 40 lines. Therefore, highly frequent routes of the OS must be designed to be "speed first". Due to this simplicity, program level errors may be much decreased. This case had no extra parts for data structure. When using hierarchical structure in a program, it is important to repeat natural language expansions as far as possible, and each program is small enough as shown in Figure 15.

Plots show a belt like zone[8]. A doted bold trend line threads the plots group and two fine sub-trend lines are drawn, which are in parallel to the centerline and equal distance apart. The trend lines show that the number of line of each route shows a negative exponential distribution[9] and the upper is 2.18 times larger and the lower is 1/2.18 times smaller than the center trend line. From these, each transition route shows such a variation. Other details were in the footnote 8 and 9.

The next is quality. In table 1, the testing were T1: unit test for all C function's, T2: FSM test, T3: integration of FSM's and system test, and each number is the number of defects. The rightmost column is for the averaged defect intensity. The number of defects found at each stage is recorded in the table. In the year 1995, an additional strict test was made as "Quality Assurance". It ranged from document check to machine test, as a professional QA does. The total average defect intensity is 12.8 defects/kilo-lines.

Figure 16 [6] shows another year's record. The horizontal axis shows the defect intensity found during testing. The average defect intensity comes to around 12 defects/kilo-lines. As these two show, the average was constant due to the constraints of procedures. As a student's average defect build-in rate is 100 defect/kilo lines, they usually checked out a considerable percentage in machine test. Through the process, more than 80% of defects had been checked-out unknowingly. Thus, the high quality strongly impressed the students. In reports, they wrote "Oh! It worked just integrated!"

Figure 16 shows the relation between the quality and the productivity for another year. The vertical axis shows the productivity, and each plot represents a team. The following facts caused the small variations of the plots: 1. The target is all the similar three states FSM, 2. The process has been strictly constrained, and 3. The team member students are arranged so as to be around the same ability. In an actual field data, such a clear trend would not appear.

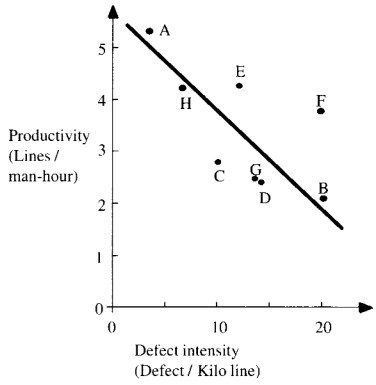

Figure 16. Quality and productivity

The trend line of Figure 16 shows that, if the defect intensity is low, the productivity is high. Both the quality and the productivity is a result of a team's mental ability, it flows naturally. *A poor quality software organization is also a poor software productivity organization. When the effective process is improved, both quality and productivity are improved.*

[8]Statistics says that a lognormal distribution appears when it is thought of as the multiplicative product of many small independent factors. In it, 99.74% of plots are in a range from x1/3 to x3 times of the standard deviation, centering at the mean.

[9] Traffic theory says that the occurrence is random and small; the holding time (the length of a route) shows a negative exponential distribution.

5. Discussions

The authors reported details of the "product" and "process" as well as "characteristics" focusing systems engineers' standpoint. As the software size has been reduced to a minimum, a systems engineer will be free from loads of programs, and can use own time and energy on what the systems design requires. There are many research papers, surveys, investigations, various comparisons, discussions, various trade-offs on this work, as well as the necessary preparations. The accumulation reflects in the works.

Due to the molecular simplicity and rich documents following the strict standardizations, a thus developed FSM or a hierarchical FSM system may become commercial parts. Wide use of this software will undoubtedly change the industry.

The situation is the same also for software group people following the systems group. They need not to worry about "waste", "strain" and "imbalance" anymore, and are encouraged to develop "rational, quantitative and scientific software engineering" as reported in SOMET07[18]. Thus they join the world of qualified engineers. The software group is not only to responsible for the production, cost, quality and delivery of each system, but also for improving the management indices by automatic design[10] as shown in SOMEO06[9] for both software groups and also systems groups through evolving the design system. The design process, described here, is standardized in various aspects, and consideration is taken of the automation[11].

The target is to emergence from labor-intensive to knowledge-intensive work.

An embedded system is a kind of hardware. In a development of a computer, vast machine-hours are used to confirm the correct operation. As an embedded system is one of hardware, it should be computer-tested similarly for all possible cases. It may be impossible in usual systems, but in this architecture, as the possible number of the combinations is much reduced, it will be possible.

Another strategic way is to use "Highly Accelerated and Yield Software Testing (HAYST) [25]". The principle is like "experiment.planning"; in which "simple lattice" is used for preparing cases of experiments. In this testing, "simple lattice" is used for preparing each test case. The usual testing saturates following to the negative exponential way. This method, however, will show a linear growth until its prepared end. Therefore it will be very useful in system phase testing.

The design method mentioned here, is also applicable to Object Oriented Design (OOD), the authors believe[26]. Presently OOD is becoming popular. Although OOD includes various capabilities, the key factor is to access the target item. It means that a software designer understand the behavior involved, and then realizes it as a FSM model. As most people have not been familiar with the concept, there are numerous

---

[10] The new software productivity from the1970's to 1980 was estimated to be from 0.5 to 2 lines/Work-hour[27]. That of early 2000 in Japan is estimated around 10 lines/Work-hour[28]. The growth for 30 years may be around 10 times. This figure is wonderfully small. That of hardware logic design would be around 10^6, which are achieved by high-level definition language (like C language) and hierarchical design tools by reusing decomposition patterns (like a CASE tool). Such heavy delay of software compared with hardware seems to be caused by the lack of competition and the non-quantitative approaches. The specialty of software is a repetitive decomposition of human concept as expressed in (mainly) natural language. As the authors' study [9] achieved a break through on this point, it is important that enough energy is poured into this work.

[11] For design automation, it is important to study the works. In the design of a system, the main document changes several times and finally it reached to a set of SDL chart and others. As is shown by this, around 100 times "write and erase" are needed until the main document is complete. After the completion, however, the frequency of the reference is much low. Therefore, if a CASE tool is designed without consideration of such work characteristics, it will be a failure product. Cut the price of a CASE tool to 1/20 of the present, and sell 1000 times more. If a CASE tool were acknowledged in this manner, more sales would be possible.

variations. But, if the essence is taken, such varieties are unnecessary and a simple hierarchical FSM architecture is enough.

This requires the evolution of Operating systems. As an FSM is natural for understanding an item, similarly it is natural that every Operating System offer full event driven as an extension of the old fashioned simple start-stop. What should be done is very simple and it will absorb all current systems in a part of the new system.

The world of software had been too conservative. Now, this world will change from being labor-intensive to creating and accumulating knowledge in preparation to the coming world of knowledge.

6. Conclusion

This paper reported an ultimate development method for embedded systems.

1. As an embedded system has been an extension of hardware, the software is different from so-called IT software.
2. The software may be constructed by a hierarchical FSM. The background, both "product and process" and an event driven Operating System as well as the development results have been discussed, putting emphasis on process using graphical means.
3. As the design process consists of an elementary FSM of around three states and the hierarchical structure of such FSM's, these two are each simplest cases. Thus, the process may be said to be the ultimate one.
4. Although this started from an embedded system development, it may be used also for OOD, and both the program design and the system design discussed here, may be automatically designed in the future. Also from this viewpoint, this is ultimate design for these.

The next stage is how to realize these on a large scale in the industry and have the people involved enjoy the merits.

Acknowledgements

One of the authors Koono expresses his deep thanks to management and employees of Hitachi, Ltd. for giving him and opportunities for continuing research/development/feedback. The authors express their gratitude to students of Information and Computer Sciences, Saitama University, who studied "Design of Embedded Systems" together. They are thankful also to members of Software Creation Project for various studies included here. They are thankful also to Mr. Daniel Horgan for his careful checks and elaborate collections of their English.

References

[1] Z. Koono, H. Chen, H. and B.H. Far: Software Systems for Embedded System Business, *IPSJ technical report,* SIG SE 139-4, 2002.10. (in Japanese)

[2] Z. Koono: Processor Systems in High Integration Age, *Joint Conference of Four Electrical Institutes 1979*, No. 27-3, 1979. (in Japanese)

[3] A. E. Joel: Bell System Features and Services, *International Switching Symposium '79*, 1979.
[4] Z. Koono, T. Kondo, M. Igari and M. Soga: Structural Way of Thinking as Applied to Good Design (Part 1. Software size), *IEEE COMSOC Global Telecommunications Conference 1991*, pp.24.3.1-8, 1991.
[5] Z. Koono, H. Chen and H. Abolhassani: An Introduction to the Quantitative, Rational and Scientific Process of Software Development (Part 1), *Software Methodologies, Tools and Techniques 2007*, pp.361-371, H. Fujita and D. Pisanelli (Eds) New Trends in Software Methodologies, Tools and techniques, IOS Press, 2007.
[6] Z. Koono, H. Chen, H. Takano and S. Morimoto: Ten Years Education of Embedded System Developments by Student Teams, *26th Software Quality Symposium* pp. 171-174, JUSE, 2007. 9. (in Japanese)
[7] Z. Koono, H. Chen and B.H. Far: Expert's Knowledge Structure Explains Software Engineering, *Joint Conference on Knowledge-Based Software Engineering (1996)*, 193-197.
[8] H. Chen, B.H. Far and Z. Koono: A Systematic Construction Method of an Expert System Used for Automatic Software Design, *Journal of Japan Society of Artificial Intelligence,* Vol. 12, No. 4, pp.616-626, July, 1997.
[9] Z. Koono, H. Chen and H. Abolhassani: A New Way of Automatic Design of Software (Simulating Human Intentional Activity), *New Trends in Software Methodologies, Tools and Techniques*, Fujita, H. and Mejiri, M., (eds.), p. 361-371, IOS press, 2006.
[10] Z. Koono, K. Ashihara and M. Soga: Structural Way of Thinking as Applied to Development, *IEEE/IEICE Global Telecommunications Conf. (1987)*, 26. 6. 1-6.
[11] Z. Koono and H. Chen: Structure of human Design Knowledge and The Quantitative Evaluation (Part 1/2), *Technical Report of IEICE* KBSE2003-57, pp. 67-72, 2004. (in Japanese)
[12] G.J. Myers: Reliable Software Through Composite Design, Petrocelli/Charter 1975.
[13] H. E. Vaughan: Research Model for Time-Separation Integrated Communication, *B. S. T. J.* Vol. 138, pp. 909-932, July 1959.
[14] CCITT: Specification and Description Language (SDL), *Recommendation* Z.100. (1976).
[15] Z. Koono and B.H. Far: High Quality Design Using SDL Technology, *SDL Forum '95 with MSC in CASE,* Braek, S. and Sarma, A., (eds.), p. 139-150, Elrsvier Science 1995.
[16] IBM: HIPO-Design Aids and Documentation Technique, CG20-1851-1, IBM 1975.
[17] Z. Koono, H. Chen and H. Abolhassani: An Introduction to the Quantitative, Rational and Scientific Process of Software Development (Part 1), *Software Methodologies, Tools and Techniques 2007*, pp.361-371, H. Fujita and D. Pisanelli (Eds) New Trends in Software Methodologies, Tools and techniques, IOS Press, 2007.
[18] Z. Koono, H. Chen and H. Abolhassani: An Introduction to the Quantitative, Rational and Scientific Process of Software Development (Part 2), *Software Methodologies, Tools and Techniques 2007*, pp.372-390, H. Fujita and D. Pisanelli (Eds) New Trends in Software Methodologies, Tools and techniques, IOS Press, 2007.
[19] K. Hiyama, N. Mizuhara, K. Mochizuki and Z. Koono: A software system for electronic switching system using distributed state transition method, *IEEE COMSOC ICC'82,* pp. 5G.3.1-5, 1982.
[20] K. Hiyama and Z. Koono: Uniform software construct for digital switching system, *Hitachi Review,* Vol. 31. No. 5., pp. 263-268, Oct. 1982.
[21] K. Mizuno M. Kusama, M.W. Medin and W.C. Garraty: DX series of wide application digital communication controller, *Hitachi Review,* Vol. 31. No. 5., pp. 275-280, Oct. 1982.
[22] T. Ohtsubo and T. Aizawa: Fully-digital switching system HDX-10, *Hitachi Review,* Vol. 31. No. 5., pp. 269-274, Oct. 1982.
[23] 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*, pp. B. 9. 5. 1-7, 1987.
[24] Z. Koono, S. Matsumoto, R. Ozaki, M. Soga and K. Ozaki K: An Environmental Simulation Tester as Applied to Traffic Characteristics Evaluation, *Proc. of The Twelfth International Tele-traffic Congress,* pp. 1284-1290, 1988.
[25] M. Yoshizawa, K. Akiyama and T. Sengoku: An introduction to Highly Accelerated and Yield Software Testing, 2007, JUSE Press. (in Japanese)
[26] T. Wang and Z. Koono: Using Extended Finite State Machine Model for Object Oriented Design, *Technical report of IPSJ,* SE-107-16, pp. 121-128, 1996. (in Japanese)
[27] B.W. Boehm: Software Engineering Economics, Prentice Hall, 1981.
[28] Software Engineering Center: White Paper of Software Development Data, Software Engineering Center IPA 2005, 2006, 2007.