Regular Paper

KDM : Kyoto Software Design Mentor

YOSHIHIRO MATSUMOTO†

The software design mentor called Kyoto software design mentor (KDM) furnishes a discipline for transforming software requirements to program systems using a unified semantic model. The proposed model makes the semantic elements in the unified domain mappable not only to the logical elements included in the requirements, but also to the logical elements included in the target-program system. In the KDM, the target-program system is assumed to apply an agent-based concurrent architecture. A set consisting of: (1) semantic domain, (2) abstract grammar used to describe semantic models, (3) unified model, and (4) logic of the target-program system is called "design set". Through the KDM steps, requirements are transformed to the design set, and then the design set is transformed to the target-program system.

1. Introduction

The Kyoto software design mentor (KDM) is a discipline taught to the students in the Information Systems Engineering Laboratory, Department of Information Science at Kyoto University. It aims to transfer to the students the real software factory technologies and experiences which the author acquired in the software factory described in Ref. 12)~19) and 20).

The key features of the KDM are summarized as follows :

(1) Most of the existing design methods, such as structured design Ref. 29), focus their major attentions on the logic-centered transformation which means the transformation in which the software requirements are transformed to the program systems using the insight into the logic implied in the requirements (e. g. data structures, functional structures, logical relationships between inputs, functions and outputs). The logic-centered transformation was not always successful in the aforementioned software factory because of the following reasons :
   * Existing methods did not furnish the formal definition of what traces are, nor indicate the concrete way how to define traces between terms in the requirements specifications and those in the target-program systems.
   * The lack of traceability made us find difficulties in revising target-program systems in case of requrements changes, and in updating requirement specifications in case of program changes.

(2) The KDM is based on the semantics-centered transformation. It is based on our observation that the semantic model which satisfies the requirements logic can be homomorphically transformed to the semantic model which satisfies the logic of the target-program system. Therefore both semantic models are unifiable. The traceability between the requirements and target-program system can be maintained through the unification homomorphisms.

(3) The unification of semantic models which can satisfy not only the logic of requirements but also the logic of the implementation is the central issue in the KDM. The target-program systems can be produced from the unified semantic model.

(4) The KDM enforces designers first to identify semantic models. It is not necessary to define the requirements logic precisely. While the model creation is in progress, the member, who is responsible for the semantic models, confirm that the seman-

† Department of Information Science, Faculty of Engineering, Kyoto University

tic models satisfy the requirements logic.

(5) The semantic models, if identified, are described from various aspects using abstract grammars, each suitable for describing each aspect. As a result, several different descriptions will be produced, which are called semantic sub-models in KDM. In order to create the target-program system, these submodels should be integrated. We need a single unified model, with which all these submodels can be integrated. In the KDM, the unified model is called the "conceptual model". There is no necessity for the conceptual model to be universal for all kinds of applications, but only for the class of target applications.

(6) After the conceptual model is constructed, the logic of the implementation (i. e. logical structure, configuration of program-system, module connections, and execution mechanism) is designed so that the logic can be satisfied by the conceptual model. The conceptual model may be revised so as to adjust itself to satisfy the designed logic, as well as to satisfy the logic of the requirements.

(7) A set consisting of the descriptions of (a) semantic domain, (b) abstract grammar used to describe semantic models, (c) conceptual model, and (d) logic of the target-program system is called "design set".

(8) The KDM enforces designers to formulate design sets assuming that major logical elements in the target-program system are concurrent objects called "agents", procedural subsystems, and data abstractions.

(9) The KDM can adapt itself to the varieties of the existing object oriented/based design methods (which are substantially based on the semantics-centered transformation) in the following manner :

* Object models, functional models, or state diagrams in Ref. 23), 25) can be assumed as the semantic submodels in the KDM.
* The object diagrams in Ref. 2), and the OODLE diagrams in Ref. 25) can be assumed as the conceptual model in the KDM.
* Procedural data abstraction, class/meta hierarchy, and message-passings can be assumed as the logic of the target-program system.

(10) Existing object oriented/based design methods do not support concurrency. The KDM agents enable to incorporate concurrency.

The paper addresses the KDM through the following sections :

Section 2 : The fundamental issues, such as semantic domain, model, semantic submodel and conceptual model are discussed.

Section 3 : Principles applied in KDM, such as the notion of a design set, are described.

Section 4 : Major logical components to configurate the target-program systems are introduced.

Section 5 : An example is presented in order to show how the KDM can be used to solve real problems.

2. Fundamental Issues

The meaning of the words "logical expression", "semantics", and "semantic model" are intuitively defined in the following example :

⟨Example⟩

[logical expression]

A logical expression L is given as follows :

L : $H(x, y) \wedge G(x, z)$

where logical elements are H, x, y, G and z.

[semantic domain]

A semantic domain S, which is a set of classified values, is given as follows :

S :

NATION —type of data
  consists of (only a part is shown) :
    SWEDEN —value of the type NATION
    NORWAY —value of the type NATION
    FINLAND —value of the type NATION

IS-ADJACENT-TO —type of relationship
  consists of (only a part is shown) :
    IS-WEST-ADJACENT-TO
      —value of the type IS-ADJACENT-TO
    IS-EAST-ADJACENT-TO
      —value of the type IS-ADJACENT-

TO.

[mapping]

The mappings between the logical elements in L and the semantic values in S, which result the evaluation of the logical expression L true, are as follows:

x −> SWEDEN, y −> NORWAY, z −> FINLAND,

H −> IS-WEST-ADJACENT-TO, G −> IS-EAST-ADJACENT-TO.

[semantic model]

The semantic domain, described above, which makes the given logical expression true by the shown mapping is called a semantic model of the logical expression.

A set consisting of L, S and the shown mapping is called "institution" by J. A. Goguen[5].

The semantic domain is an ordered collection of classified values. An example of domain is the "Scott's domain", developed by D. Scott[24], used for proving partial correctness of recursive programs.

Sets of semantic elements and their relationships between semantic elements are described using abstract syntactic notations or abstract grammars. For example, we have various grammars to draw E-R (entity-relationship) diagrams. In computer-aided software engineering (CASE) environments, we have STL (semantic transfer language)[26], CDIF (CASE data interchange format)[3], IRD/schema (information retrieval dictionary schema)[9], and PCTE/SDS (portable common tool environment schema definition set)[22]. These languages and systems are used to describe domains for data, and are used to manage the semantic integrity of the data which is exchanged between software tools.

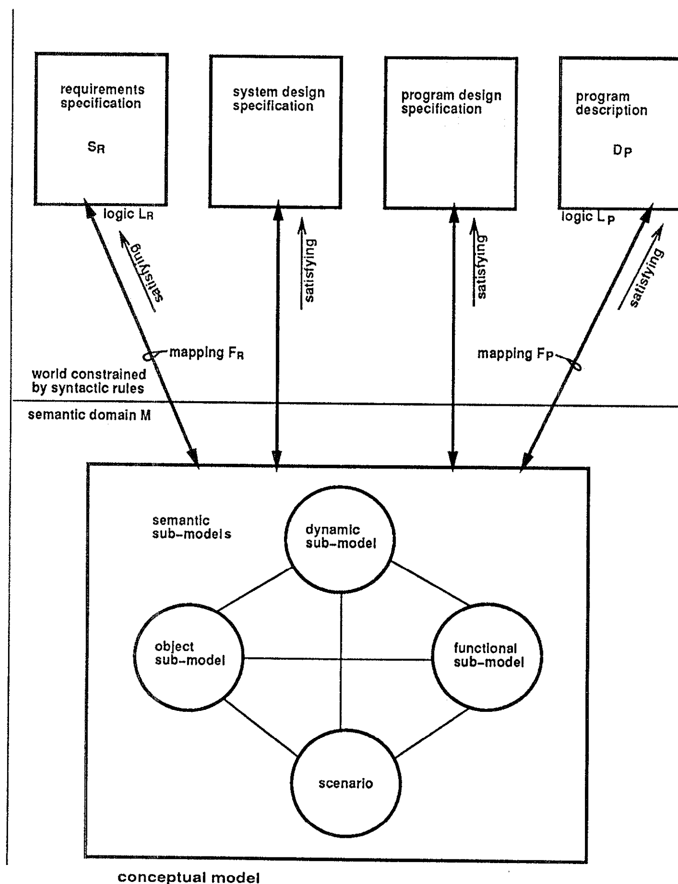


Fig. 1 The conceptual model and the related software descriptions.

The precise semantic of software products have often been described using mathematics such as specification algebra[4] and order-sorted algebra[6]. By H. Ehrig, specifications are considered as objects of a category together with suitable morphisms called specification morphisms. J. A. Goguen proposed the order-sorted algebra for describing semantics of parameterized programming, which provides a subsort partial ordering among sorts and interprets it semantically as subset inclusion.

Formal engineering approaches to construct large correct specifications[1),8),10)] also use algebras for describing precise semantics of specified items.

The description of the specification-part of an Ada-package is represented by a category of vocabularies in Ref. 21). If package A uses package B, the category of vocabularies for B is homomorphically inclusive in the category of vocabularies for A. Specifications for larger Ada packages are constructed step by step using functors for each inclusion morphism of vocabulary-categories.

The logic of requirements, design, program and test produced by the particular project can be satisfied by one unified semantic model. In other words, all logical properties included in the descriptions of requirements, design, program, and test specifications produced in the same project can be satisfied by the single unified semantic model (the conceptual model) when the mappings from the unified semantic domain to each logical description are made.

Figure 1 shows the relationships between four selected product descriptions (requirements specification, system design specification, program design specification and program description) and a conceptual model. The conceptual model integrates four semantic submodels: object submodel, dynamic submodel, functional submodel and scenario, where relationships between semantic elements included in each submodel are described using abstract grammars.

Each semantic element included in the conceptual model should be mappable to each logical element included in the requirements specification and satisfies its logic. Simultaneously, the coceptual model should satisfy the logic of the target-program system as well as the logic of the requirements.

3. Principles

Let us assume that we have existing documents produced by the project that was already successfully finished. The following documents are available:

semantic model	M
requrements specification	S_R
the logic of S_R	L_R
program description	D_P
the logic of D_P	L_P
mapping of M to S_R to make L_R true	F_R
mapping of M to D_P to make L_P true	F_P

Assume that a new project is organized. In this project, the semantic model M' is identified through the requirements analysis. And it is found that the M' belongs to the same class as the one that the M belongs to. If the semantic model M' of the requirements specification S_R', which we will have for the new project, is homomorphically inclusive to the semantic model M of the former specification S_R, the program description D_P' can be produced automatically. Using this principle, the application generators for various types of applications such as office automation, industrial direct digital control, supervisory control, and elctric power generation control have been developed and commercially used.

These program generators generally provided the languages for describing the semantic models and the logic of the program descriptions, and they generated target-program systems using M, L_P, and the library which accommodated reusable program modules.

An example program gererator called the COPOS (Computerized Optimum Plant Operation System)[27] developed in the software factory referred to in Section 1, has been practically used for generating the target-program systems of thermal/nuclear power generation plant control systems using the following principle:

The COPOS semantic model is described using a hierarchical state system in which the following conditions, called "state hierarchy", must hold.

(1) L_i is a logical expression consisting of logical operations between atomic variables each of which has a truth value called "heuristic value"[12]. A heuristic

value represents a result of heuristic decision about an input value. For example, the fact that the heuristic value of x becomes true means that the human decision about the input x is: "The event, which should be taken measures, has happened about x".

(2) In S_i, the logical expression L_i must always be satisfied by the mapping from the model M.

(3) In $S_{i,j}$, a sub-state of S_i, the logical expression $L_i \wedge L_j$ must always be satisfied by the mapping from the model M.

(4) In the sub-state $S_{i,j,\ldots,k}$, the logical expression $L_i \wedge L_j \wedge \ldots \wedge L_k$ must always be satisfied by the mapping from the model M.

(5) An action is an atomic execution of a program module. An activity is a automatic concatenation of the same action which continues until the stopping condition becomes true.

(6) Actions and activities are allowed to take place only for the specified state. If the state should change, the action or activity currently taking place in that state must be stopped or aborted.

The logical expression associated with each state, mentioned in (1), (2), and (3), are computed at each designated period. The length of the computation period, which is generally very short, is defined for each state. If the specified logical expression should become false in the computation, all activities being allowed for execution in that state and its sub-states are stopped or aborted. If any new state is met and identified, the new action or activity which are allowed to take place in the new state can be started.

The action or activity which are allowed to take place in a certain state, can only be started when the designated starting conditions (preconditions) are satisfied. Activities can continue until stopping conditions become true. When an action or activity successfully stops, its postcondition must be satisfied.

Each action or activity is defined using a set of descriptions consisting of the elements shown as follows:

(1) Name of the action (or activity).

(2) Name of the state in which the action (or activity) is allowed to act.

(3) Definition of the state (the definition consists of a logical expression called the state-condition, the semantic domain, and the mapping to make the expression true).

(4) Sampling time-interval at which the state-condition is periodically computed.

(5) Starting condition (the set consisting of a logical expression, semantic domain, and the mapping for enabling evaluation of the expression). —If the conjunction between the designated state-condition and starting condition becomes true, the action (or activity) can be started. The starting condition, which is generally complicated, is separated from the state-condition in order to make state-conditions, which should be computed very often, simpler.

(6) Stopping condition (a logical expression which is used to stop activities when it becomes true).

(7) Postcondition (the condition that must be satisfied when the action or activity stops).

(8) Action or activity (the designation of the subordinated program modules which should be executed to accomplish the required action or activity).

The timing relationships between states, conditions, actions and activities are explained in **Fig. 2**. The state S is defined as the concatenation of the event E which is periodically observed at every T. The one action and two activities, which are allowed to start when the conjunction between the state S and starting condition becomes true, are shown. The activity A stops when the shown stopping condition becomes true. The activity B stops only when the state S becomes false.

Assume that we have the project N. The semantic model M^N, which is produced by the project N, is the design set which satisfies the requirements specification S_R^N. Major program components which cofigurate the program description D_P^N are: the state identifier, the action/activity controller, the database which maintains the design sets, the interpreter which

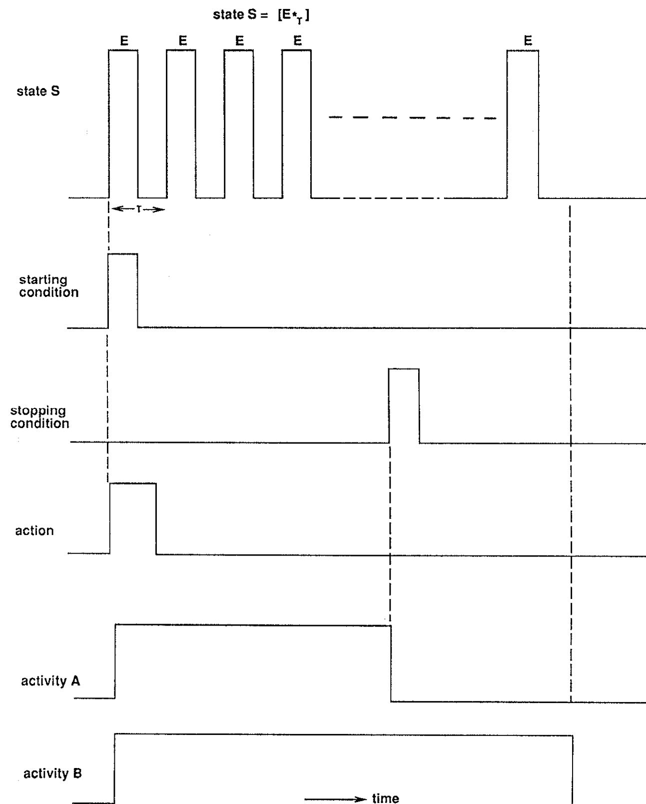


Fig. 2 Timing of the events, conditions, action, and activities.

interprets the design sets, and the subordinated program modules.

The logic L_P^N is roughly explained as follows: The state identifier scans the inputs from external industrial environments at every designated period of time, and then identifies the state in which the industrial environment are stationed. The action/activity controller reads the current state periodically at every designed period of time, and creates or deletes tasks. Each task reads the necessary conditions from the corresponding design set, and controls the execution of subordinated program modules.

If the project N is responsible for a medium-size application program system, nearly one thousand design sets will be developed. The COPOS, in the software factory, reads these sets, descriptions of the L_P^N, and F_P^N, and generates the program description D_P^A. The major program descriptions being generated are device modules (the modules for interfacing external devices) and the scheduler which control actions and activities. Actions and activities are implemented using reusable modules.

The mapping F_P^N is straightforward. The design sets in the M^N are mapped to the database which can be accessed by the scheduler and device modules which are described in the D_P^N.

Based on these experiences, the KDM adopts the notion called the "design set". A design set is a set of data which cosists of the items listed as follows:

(a) the logic (logical expressions included in the logic) and logical elements with which the given problem or requirements is formulated; —This corresponds to the L_R mentioned above.

(b) the unified semantic domain which can satisfy both the logic of the requirements and the logic of the target-program system; —This corresponds to the M

mentioned above.

(c) the abstract grammars which are used to describe semantic submodels and the coceptual model ; —This corresponds to the grammar with which state-transition diagrams, timing charts, sequential control logic diagrams, and action/activity sets are described.

(d) the conceptual model which is an integration of semantic sub-models ; —This corresponds to the model which defines action/activity sets together with the mappings : the F_R and F_P.

(e) the logic and logical elements with which the target-program system is based on.—This corresponds to the L_P mentioned above.

Among these items in the design set, the L_R and the semantic submodels should readily be identified during requirements analysis conducted in the preceding stage. The conceptual model should include the semantic submodels homomorphically, and should satisfy the L_R which will be interrogated through meetings between customers and designers. The L_P should be designed so that each semantic element in the conceptual model can be mapped to each respective logical element in the L_P in satisfaction.

The problem how to design an unified conceptual model is called "model unification".

4. Target Program Systems

Major components which configurate target-program systems are agents, sets of data abstraction (which represent states, conditions and data), and procedural subsystems (procedural executable descriptions). The agents are concurrent objects with the following properties :

(1) An agent capsulates concurrent operations which act on a set of data internal to itself.

(2) "Holon", a system by A. Köstler[11], is considered as a harmony of Holos (whole) and On (individual). A holonic system is a world organized by a set of autonomous elements communicating with one another. Each element in the holonic system, the holoic element, acts to seek for its own profit, while it cooperates with other elements to accomplish the common objectives. Each KDM agent should be designed so that the target-program system can behave like a holonic system.

(3) Each agent, as a client, activates procedural subsystems and data abstractions, creates and manages a sequential or concurrent transaction in response to the request from the external world. Each transaction created and managed by the agent should be atomic, consistent, isolated and durable.

As shown in **Fig. 3**, an agent instance is composed of one scheduler, one storage for accommodating the commonly accessible data, light-weight processes, one real-time clock, input ports, output ports and utilities for managing the agent's resources and execution. Communications between agents are accomplished by concurrent passings of messages. The semantics of "unacknowledged-type message passing" and "mailback-wanted-type message passing"[18),28)] are applied. When two messages are sent, from the agent P through the different output ports of the P, to the input port I of the agent A, the time ordering of the two messages (determined by the P's clock) is preserved in the time ordering in the I when the messages are received.

In the unacknowledged-type message passing, the sender of a message does not wait the acknowledgement (or answer) of that message from the receiver. In the mailback-wanted-type message passing, the names of the agents to which the answer is to be sent (which is called continuing object), are included in the message being sent. The sender does not wait the answers from the receiver after it has sent messages, but, if necessary, it receives new messages sent from the continuing object.

The scheduler starts action at the moment when the agent is created. It can create messages and send them to other agents. It can receive solicited inputs (answers from the agents to which the scheduler has sent a request in advance) through input ports, and activates necessary light-weight processes. It can also receive unsolicited inputs through input ports and activates the necessary light-weight processes. The light-weight processes can access the common data in concurrence. The serialization is accomplished by the service provided by the utilities provided inside the agent. Concurrent transactions can be allowed, where a transaction means a locus of control to connect light-weight processes accommodated in the different agents.

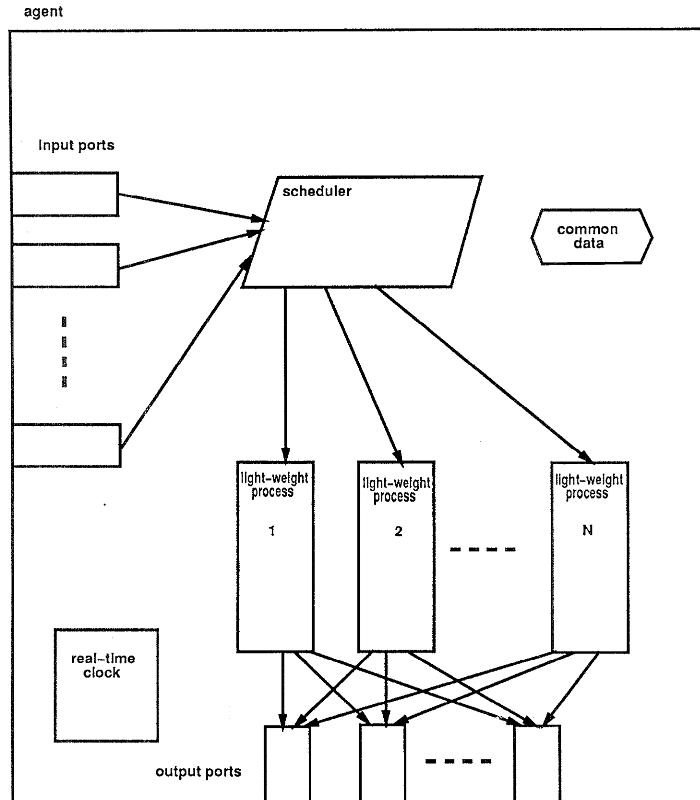


Fig. 3 Pattern representing the feature of an agent.

The atomicity, cosistency, isolation and durability in every transaction should be maintained.

The Ada package called "BANK" shown in Appendix, a toy model of banking service, simulates the signature of an agent. The package "BANK", includes the following components :

(1) input ports : RECEPTIONIST_INPUT_PORT, REQUEST_SERVICE_INPUT_PORT.

(2) output ports : RECEPTIONIST_OUTPUT_PORT, REQUEST_SERVICE_OUTPUT_PORT.

(3) scheduler : RECEPTIONIST.

(4) processes : TELLER_1, TELLER_2, TELLER_3.

(5) common data : CHG_OPER.

(6) utility : MONITOR.
(Task "MONITOR" serializes the accesses to package "CHG_OPER".)

(7) real-time clock.

Task "CUSTOMER_1", "CUSTOMER_2", and "CUSTOMER_3" simulate clients who request banking functions : "DEPOSIT", "WITHDRAW", "CHECK_ACCOUNT", and "PAY_BILL".

5. Example Use of the KDM

The model unification, which indicates the step for designing an unified conceptual model coping with its dependence on L_R, L_P and semantic submodels, is discussed in this section. The unification means that the semantic models which satisfiy L_R, L_P and all semantic submodels are unified. The L_R, L_P and submodels are homorphically included in the unified model through the unification.

The following list presents major items which configurate our conceptual model of the example mentioned later in this section.

(1) Event :

An event is defined as a truth value obtained

as a result of computation of a logical expression in which all atomic units are what we call heuristic values. A heuristic value is a result of heuristic decision about an input value, which indicates if the input value matches the preset condition or not. For example, the fact that the heuristic value of t_j is true means the decision that the temperature value denoted by the symbol t_j matches the preset truth value. The input measurement and computation for heuristic decisions are activated periodically.

(2) State:

A state is defined as a concatenated set of an event. For example, state S is defined as:

$S=[E*_T]$ —"$E*$" means a concatenation of the event E,

where E is an event observed repeatedly at period T.

(3) State structure:

The state S_i is said superior to the substate S_{i+1} if the target object in the substase S_{i+1} is allowed to exist only while it is also in the state S_i. If the target object gets out of the state S_i, all actions and activities performed under the state S_{i+1} must be stopped or aborted.

(4) Condition:

A condition is an event observed at a designated timing. Conditions includes preconditions for triggering actions or activities, intermediate-conditions which are checked during execution of actions or activities, stopping conditions for stopping activities, and postconditions which are checked when actions or activities stop.

Our toy example aims to control a car-engine speed. We have the external devices shown as follows:

(1) Engine switch includes: two not-self-holding type push button switches labeled "IN" and "OFF".

(2) Engine mode switch includes: four not-self-holding type push button switches labeled "PARK", "REVERSE", "NEUTRAL", and "DRIVE".

(3) Speed mode switch includes: four not-self-holding type push button switches labeled "IDLING", "FIRST", "SECOND", and "THIRD".

(4) Speed setter is an analog type positioner.

(5) Engine/speed control actuator is an actuator to accept the engine-in/off and speed-control signals (results of proportional/integral/differential (PID) computation for the difference between current-speed and set-speed), and to generate outputs to each car device.

The KDM recommends designers to follow the following steps:

(1) Identify external devices, and design a ***device agent*** for interfacing each external device.

(2) Identify the sets of common data (or files), and design a ***data agent*** for each identified data set.

(3) Identify data abstractions, and design a ***computation agent*** for each data abstraction.

(4) Build the conceptual model, using the design set format, by going through the following substeps:

(4a) Define events (E), states ($[E*_T]$). and period T.

(4b) Describe state-diagrams.

(4b) Define every action to be performed for each respective state, the condition to trigger each action/activity, and the condition to stop each activity.

(4 c) For all actions, define computation and the data to be computed, and complete action and activity sets as explained in Section 4.

(5) Design precisely the device, data and computation agents which were identified in the step (1), (2) and (3). Each agent should be designed so that each element included in the conceptual model can be mapped to the corresponding element in each agent straightforwardly.

(6) Describe data flow models, and then design message patterns.

(7) Describe scenario, and then adjust agents so that the dynamic behaviour of the agent system is consistent with the described scenario.

The design sets for the car-engine problem can be organized in the following manner. At the beginning of the sub-step (4a), the state-diagrams, of which a simple example is shown in **Fig. 4**, is drawn using the diagrammatic grammar of Ref. 7) with some extension. The extended properties are obvious from the following explanation:

(1) The state "Engine/speed" shown in Fig. 4 is defined as [Normal$*$] where event "Normal" indicates that all devices which implement

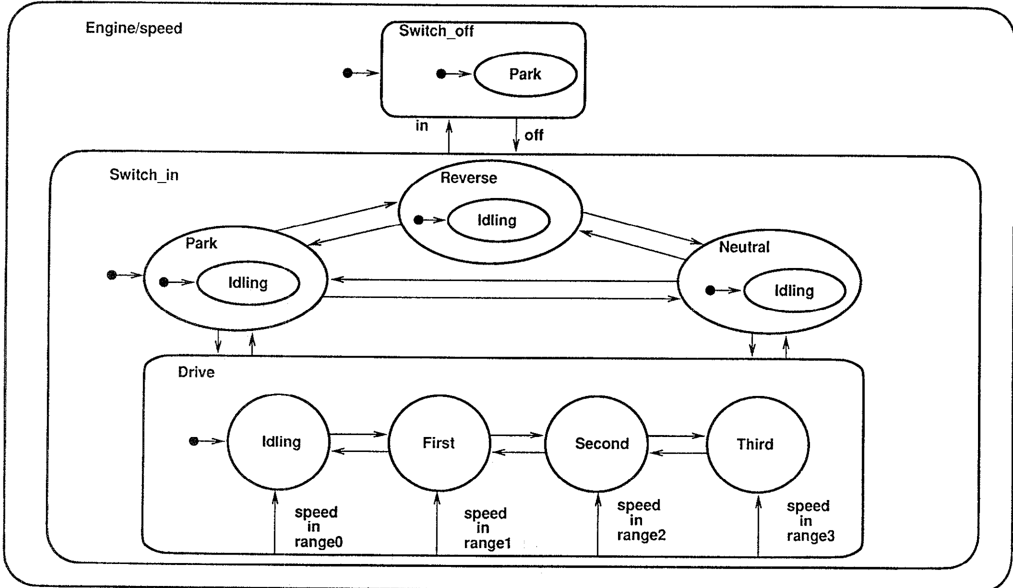


Fig. 4 State diagram of car engine transmission.

the engine/speed control system are in the normal condition. The event "Normal" is implemented by the conjunctions of the heuristic values, each of which represents the heuristic decision that the observed device is in the normal condition.

(2) The state "Switch_off" is defined as [(Normal $\wedge$ OFF) $*$] where the event "OFF" indicates that the push button OFF has been pushed.

(3) Every sub-state is defined in the manner explained in Section 4. For example, the state "Idling" is defined as :

$$[(\text{Normal} \wedge \text{IN} \wedge ((\text{PARK} \vee \text{REVERSE} \vee \text{NEUTRAL}) \vee (\text{DRIVE} \wedge \text{IDLING}))) *]$$

where the terms written in capital letters represent the events that the respective push buttons have been pushed.

(4) When a new state is entered, the sub-state pointed by a blackrooted arrow is automatically selected. Assume that the target system is in the state "Switch_off" and the push button IN has been pushed. The target system automatically goes into the state "Switch-in/Park/Idling".

Figure 5 shows a rough sketch of the target-program system architecture of the car-engine example. For each external device shown in the figure, each specified device agent is assigned. The process "scheduler" in the computation agent reads necessary input-values from every device agent, computes heuristic values, computes events, and identifies the current state. Using the action/activity sets which are stored in the common data set, the scheduler selects and activates necessary light-weight processes for serving control, display, or alarm. The scheduler may also activate other processes such as event-identification or state-identification. The schedular updates periodically the data in the common data storage, and maintains "current events", which will be accessed by other light-weight processes. The outputs from every light-weight process are sent to output devices through the "output_port".

6. Concluding Remarks

A poet usually organizes his own semantic domain while observing the real world objects, and maps each semantic element in that domain to the respective real world object for understanding what happens there. Then, the poet creates the poem which can be satisfied by the same semantic domain. The syntactic logic for composing the poem, such as rhyme, often gives some hard constraints (for example, the Japanese type poems called "haiku" must follow a very strict logic called "5-7-5"). This logic

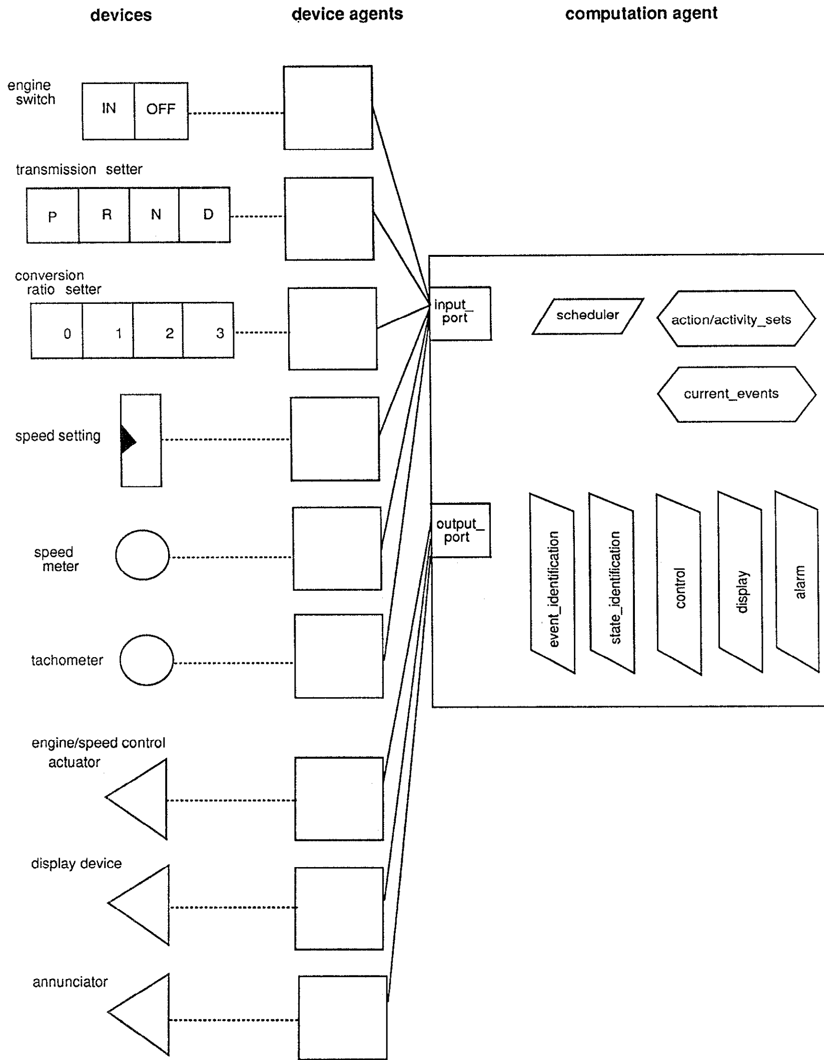


Fig. 5 Agents for the car-engine control example.

corresponds to the logic for implementing the target-program system in case of software design. The logic of the real world phenomena is quite different from the logic of the poem, so is the logic of the requirements different from the logic of the target-program systems. However the semantic domain to satisfy these different logics are completely the same, and all semantic models can be inclusive. The KDM does not enforce logic-centered design. Instead, the KDM enforces first to create a unified semantic model which can satisfy both the logic of the requirements and logic of the target-program system. Through the KDM steps, the target-program system can be produced from the unified semantic model. The KDM applies the agent-based architecture for the target-program system, because the semantics of the agent-based system can be made homorphical-inclusive in the semantics of the real world problem.

Agents described using Ada, shown in the example, still do not provide the features of the holonic system. These are our future research items.

Acknowledgement This research has been supported by a Scientific Grant from the Ministry of Education and Culture of Japan.

My sincere thanks go to Dr. Veiko Seppänen

who has reviewed this manuscript, and to Mrs. Chunglin Sui who assisted the author for programming many agents in Ada.

References

1) Björner, D. and Jones, C. B. (eds.) : *The Vienna Development Method : The Meta-Language*, Springer-Verlag (1978).
2) Booch, G. : *Object Oriented Design*, The Benjamin/Cummings Publishing (1991).
3) Electronic Industries Association (eds.) : CDIF (CASE Data Interchange Format) EIA IS/81, IS/82, and IS/83 (1991).
4) Ehrig, H. and Mahr, B. : *Fundamentals of Algebraic Specification 1*, Springer-Verlag, Berlin, Heidelberg (1985), and Ehrig, H. and Mahr, B. : *Fundamentals of Algebraic Specification 2*, Springer-Verlag, Berlin, Heidelberg (1990).
5) Goguen, J. A. and Burstall, R. M. : Introducing institutions, *Lecture Notes in Computer Science*, Vol. 164, Springer-Verlag, Berlin, Heidelberg (1984), also in Proceedings of Workshop on Logics of Programs (1983).
6) Goguen, J. A. : Principles of Parameterized Programming, *Software Reusability* Vol. 1, Biggerstaff, T. J. and Perlis, A. J. (eds.), pp. 159-225, ACM Press Frontier Series (1989).
7) Harel, D., et al. : STATEMATE : A Working Environment for the Development of Complex Reactive Systems, *Proceedings of the 10th International Conf. on Software Engineering*, pp. 396-406 (1988).
8) Heyes, I. (ed.) : *Specification Case Studies*, International Series in Computer Science, Printice-Hall (1987).
9) ISO/IEC JTC1/SC21 (eds.) : Information Resource Dictionary System (IRDS), Service Interface, Working Draft, Revision 11, SC21 N4895 (1990).
10) Jonkers, H. B. M. : Introduction to COLD-K, in *Algebraic Methods : Theory, Tools and Applications*, Wirsing, M. and Bergstra, J. A. (eds.), Springer-Verlag, Berlin, Heidelberg, pp. 139-205 (1989).
11) Köstler, A. : *The Ghost in the Machine*, Hutchinson, London (1967).
12) Matsumoto, Y. : A Method of Software Requirements Definition in Process Control, *Proceedings of COMPSAC 77, IEEE Computer Society*, pp. 128-132 (1977).
13) Matsumoto, Y., et al. : SPS : A Software Production System for Mini-computer and Micro-computers, *Proceedings of COMPSAC 78, IEEE Computer Society*, pp. 396-401 (1978).
14) Matsumoto, Y., et al. : SWB System : A Software Factory, *Proceedings of the Symposium on Software Engineering Environments*, 16-20 June 1980, Lahnstein, Germany (sponsored by GMD). (also included in *Software Engineering Environments,* Hunke, H. (ed.), GMD (1981).
15) Matsumoto, Y. : Software Education in an Industry, *Proc. IEEE COMPSAC'82,* pp. 92-94, Nov. 10-12, Chicago (1982).
16) Matsumoto, Y. : Organizational Effort for Reusing Existing Software, *Proc. COMPCON'84 Fall,* p. 131, Washington, D. C. (1984).
17) Matsumoto, Y. : Some Experiences in Promoting Reusable Software : Presentation in Higher Abstract Levels, *IEEE Trans. Softw. Eng.,* Vol. SE-10, No. 5, pp. 502-513 (1984).
18) Matsumoto, Y. : Requirements engineering and software development : A study toward another life-cycle model, *Proceedings of Brown Boveri Symposium on Computer Systems for Process Control*, September 2-3 1985, in, Computer System for Process Control. Güth, R. (ed.), pp. 241-263, Plenum Press, New York (1986).
19) Matsumoto, Y. : A Software Factory : An Overall Approach to Software Production, Software Reusability, Freeman, P. (ed.), pp. 155-178, Computer Society Press of the IEEE (1987).
20) Matsumoto, Y. : Approaching Productivity and Quality in Software Production—How to Manage a Software Factory, *Proceedings of DRP'87 International Conf.,* May 18-20, Diebold Research Program-Europe, pp. 103-122 (1987).
21) Matsumoto, Y. : Software Design Process as Category Morphism, *J. Inf. Process*, Vol. 14, No. 3, pp. 272-283 (1991).
22) ECMA/TC33, eds. : Portable Common Tool Environment (PCTE), Standard ECMA-149, Abstract Specification (1990).
23) Rumbaugh, L., et al. : *Object-oriented Modeling and Design,* Prentice-Hall, Englewood Cliffs (1991).
24) Scott, D. S. : Outline of Mathematical Theory of Computation, *Proceedings of the 4th Annal Princeton Conf. on Information Sciences and Systems*, pp. 169-176 (1970).
25) Shlaer, S. and Mellor, S. J. : *Object Lifecycles : Modeling the World in States*, Prentice-Hall, Englewood Cliffs (1992).
26) IEEE-P1175, eds. : A Standard Reference Model for Computing System Tool Interconnections, Draft 7 (1990).
27) Tanaka, S., et al. : New Comcept Software

System for Power Generation Plant Computer Control "COPOS", *Proceedings of PICA 73*, The IEEE Power Engineering Society, pp. 142-149 (1973).

28) Yonezawa, A. and Matsumoto, Y.: Object-oriented Concurrent Programming and Industrial Software Production, in *Formal Methods and Software Development*, Vol. 2, Ehrig, H. et al. (eds.) pp. 395-409, Springer-Verlag, Berlin, Heidelberg (1985).

29) Yourdon, E. and Constantine, L.: *Structured Design*, Prentice-Hall, Englewood Cliffs (1979).

Appendix An example of the KDM agent

A toy model representing the functions of a bank is simulated by the KDM agent described using Ada.

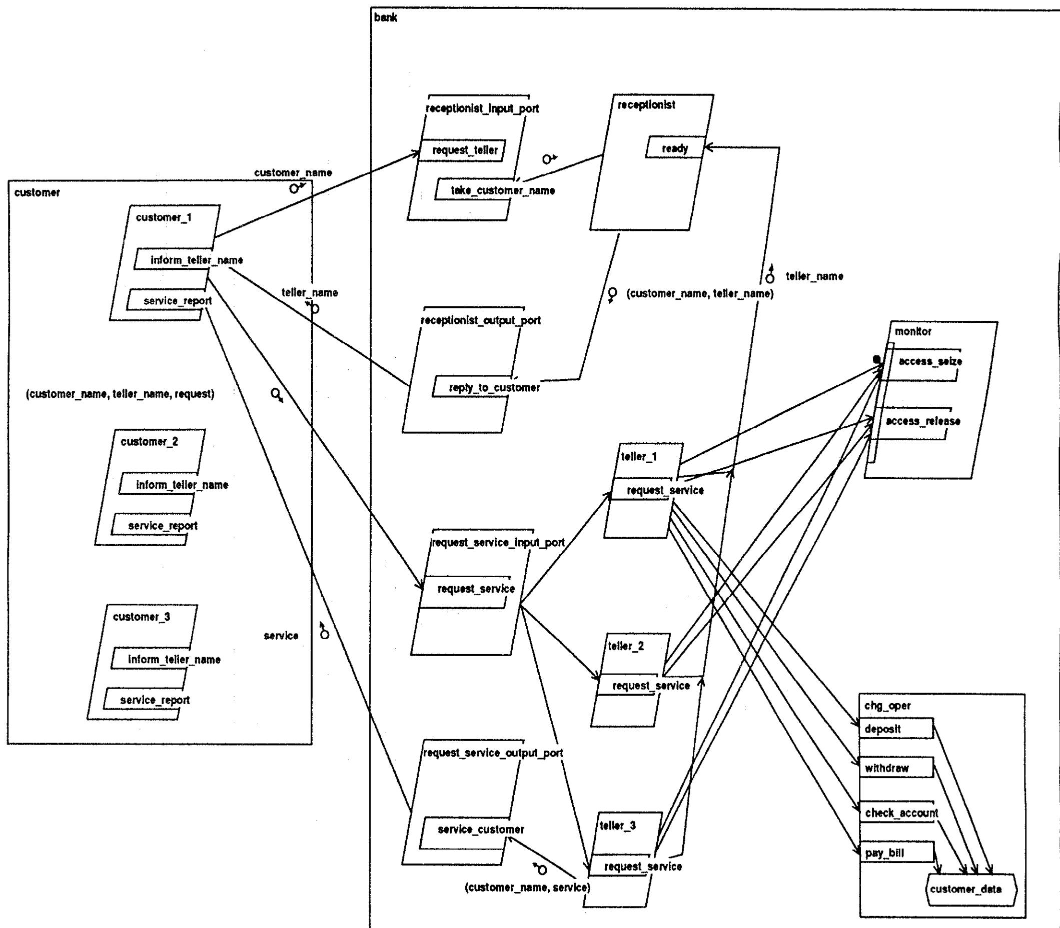


Fig. 6 The target-program system architecture for the BANK.

```
package BANK is
  type T_TYPE is (DEPOSIT, WITHDRAW, CHECK_ACCOUNT, PAY_BILL);
  subtype TE_NAME is INTEGER range 1..3;
  subtype CU_NAME is INTEGER range 1..3;

  task RECEPTIONIST_INPUT_PORT is
      entry REQUEST_TELLER(CUSTOM_NAME: in CU_NAME);
      entry TAKE_CUSTOMER_NAME(CUSTOM_NAME: out CU_NAME);
   end  RECEPTIONIST_INPUT_PORT;
  task REQUEST_SERVICE_INPUT_PORT is
      entry REQUEST_SERVICE(CUSTOMER_NAME: in CU_NAME; TELLER_NAME: in TE_NAME; REQUEST: in out T_TYPE);
   end  REQUEST_SERVICE_INPUT_PORT;
end BANK;

with BANK;
package CUSTOMER is
  use BANK;
  task CUSTOMER_1 is
      entry INFORM_TELLER_NAME(TELLER_NAME: in TE_NAME);
      entry SERVICE_REPORT(SERVICE: in T_TYPE);
   end CUSTOMER_1;
  task CUSTOMER_2 is
      entry INFORM_TELLER_NAME(TELLER_NAME: in TE_NAME);
      entry SERVICE_REPORT(SERVICE: in T_TYPE);
   end CUSTOMER_2;
  task CUSTOMER_3 is
      entry INFORM_TELLER_NAME(TELLER_NAME: in TE_NAME);
      entry SERVICE_REPORT(SERVICE: in T_TYPE);
   end CUSTOMER_3;
end CUSTOMER;

with CUSTOMER;
package body BANK is
   use CUSTOMER;
   CUSTOMER_NUM : INTEGER;
   type CUSTOMER_DATA is
     record
       NAME: STRING(1..10);
       ID_NO: STRING(1..10);
       MONEY: INTEGER;
     end record;
   CUSTOMER_REC: array(1..CUSTOMER_NUM) of  CUSTOMER_DATA;
   TELLER_NAME: TE_NAME;

   task TELLER1 is
      entry REQUEST_SERVICE(CUSTOMER_NAME: in CU_NAME; TRANSACTION: in out T_TYPE);
    end TELLER1;
   task TELLER2 is
      entry REQUEST_SERVICE(CUSTOMER_NAME: in CU_NAME; TRANSACTION: in out T_TYPE);
    end TELLER2;
   task TELLER3 is
      entry REQUEST_SERVICE(CUSTOMER_NAME: in CU_NAME; TRANSACTION: in out T_TYPE);
    end TELLER3;

   task RECEPTIONIST is
      entry READY(TELLER_NAME: in TE_NAME);
    end RECEPTIONIST;

   task RECEPTIONIST_OUTPUT_PORT is
      entry REPLY_TO_CUSTOMER(CUSTOMER_NAME: in CU_NAME; TELLER_NAME: in TE_NAME);
    end RECEPTIONIST_OUTPUT_PORT;

   task REQUEST_SERVICE_OUTPUT_PORT is
      entry SERVICE_CUSTOMER(CUSTOMER_NAME: in CU_NAME; SERVICE: in T_TYPE);
    end REQUEST_SERVICE_OUTPUT_PORT;

   task MONITOR is
       entry ACCESS_SEIZE;
       entry ACCESS_RELEASE;
    end MONITOR;
```

Fig. 7 Ada program for the example of Fig. 6.

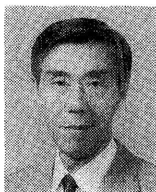
```
package CHG_OPER is
   procedure DEPOSIT(CUSTOMER_NAME: CU_NAME; MONEY: INTEGER);
   procedure WITHDRAW(CUSTOMER_NAME: CU_NAME; MONEY: out INTEGER);
   procedure CHECK_ACCOUNT(CUSTOMER_NAME: CU_NAME; MONEY: out INTEGER);
   procedure PAY_BILL(CUSTOMER_NAME: CU_NAME; MONEY: in  INTEGER);
end CHG_OPER;

task body RECEPTIONIST_INPUT_PORT is
  begin
    loop
      select
        accept  REQUEST_TELLER(CUSTOM_NAME: in CU_NAME);
      or
        accept  TAKE_CUSTOMER_NAME(CUSTOM_NAME: out CU_NAME);
      end select;
     end loop;
  end  RECEPTIONIST_INPUT_PORT;

 task body TELLER1 is
   TELLER_NAME: TE_NAME:=1;
   CUSTOMER_NAME: CU_NAME;
   MONEY: INTEGER;
  begin
    loop
      RECEPTIONIST.READY(TELLER_NAME);
      accept  REQUEST_SERVICE(CUSTOMER_NAME: in CU_NAME; TRANSACTION: in out T_TYPE) do
         MONITOR.ACCESS_SEIZE;
         case TRANSACTION is
           when DEPOSIT => CHG_OPER.DEPOSIT(CUSTOMER_NAME,MONEY);
           when WITHDRAW => CHG_OPER.WITHDRAW(CUSTOMER_NAME,MONEY);
           when CHECK_ACCOUNT =>CHG_OPER.CHECK_ACCOUNT(CUSTOMER_NAME,MONEY);
           when PAY_BILL => CHG_OPER.PAY_BILL(CUSTOMER_NAME,MONEY);
          end case;
         MONITOR.ACCESS_RELEASE;
      end  REQUEST_SERVICE;
      REQUEST_SERVICE_OUTPUT_PORT.SERVICE_CUSTOMER(CUSTOMER_NAME,TRANSACTION);
     end loop;
  end TELLER1;

 task body TELLER2 is
   begin
     null;       -- the same as TELLER1
   end TELLER2;
 task body TELLER3 is
   begin
     null;       -- the same as TELLER1
   end TELLER3;

 task body  RECEPTIONIST_OUTPUT_PORT is
   begin
     loop
       accept REPLY_TO_CUSTOMER(CUSTOMER_NAME: in CU_NAME; TELLER_NAME: in TE_NAME) do
         case CUSTOMER_NAME is
            when 1=> CUSTOMER_1.INFORM_TELLER_NAME(TELLER_NAME);
            when 2=> CUSTOMER_2.INFORM_TELLER_NAME(TELLER_NAME);
            when 3=> CUSTOMER_3.INFORM_TELLER_NAME(TELLER_NAME);
         end case;
       end REPLY_TO_CUSTOMER;
    end loop;
   end RECEPTIONIST_OUTPUT_PORT;

 task body RECEPTIONIST is
    TELLER_NAME: TE_NAME;
    CUSTOMER_NAME: CU_NAME;
   begin
     loop
       accept READY(TELLER_NAME: in TE_NAME);
       RECEPTIONIST_INPUT_PORT.TAKE_CUSTOMER_NAME(CUSTOMER_NAME);
       RECEPTIONIST_OUTPUT_PORT.REPLY_TO_CUSTOMER(CUSTOMER_NAME, TELLER_NAME);
      end loop;
    end RECEPTIONIST;
```

Fig. 7 (Continued)

```
    task body MONITOR is
      type STATE is (IDLE, BUSY);
      PRESENT_STATE: STATE:=IDLE;
      begin
        loop
          select
            when PRESENT_STATE=IDLE =>
                   accept ACCESS_SEIZE  do
                      PRESENT_STATE:=BUSY;
                   end  ACCESS_SEIZE;
           or
             accept ACCESS_RELEASE do
                 PRESENT_STATE:=IDLE;
             end ACCESS_RELEASE;
           end select;
        end loop;
    end MONITOR;

   task body REQUEST_SERVICE_INPUT_PORT is
       REQUEST: T_TYPE;
       begin
         loop
           accept REQUEST_SERVICE(CUSTOMER_NAME: in CU_NAME; TELLER_NAME: in TE_NAME; REQUEST: in out T_TYPE) do
             case TELLER_NAME is
                 when 1 => TELLER1.REQUEST_SERVICE(CUSTOMER_NAME,REQUEST);
                 when 2 => TELLER2.REQUEST_SERVICE(CUSTOMER_NAME,REQUEST);
                 when 3 => TELLER3.REQUEST_SERVICE(CUSTOMER_NAME,REQUEST);
             end case;
           end loop;
       end REQUEST_SERVICE_INPUT_PORT;

     task body REQUEST_SERVICE_OUTPUT_PORT is
        begin
          loop
            accept SERVICE_CUSTOMER(CUSTOMER_NAME: in CU_NAME; SERVICE: in T_TYPE) do
              case CUSTOMER_NAME is
                 when 1=>CUSTOMER_1.SERVICE_REPORT(SERVICE);
                 when 2=>CUSTOMER_2.SERVICE_REPORT(SERVICE);
                 when 3=>CUSTOMER_3.SERVICE_REPORT(SERVICE);
              end case;
             end SERVICE_CUSTOMER;
           end loop;
      end REQUEST_SERVICE_OUTPUT_PORT;

     package body CHG_OPER is
       procedure DEPOSIT(CUSTOMER_NAME: CU_NAME; MONEY: INTEGER) is
        begin
          null;     -- the function to add money to CUSTOMER_DATA.MONEY
       end DEPOSIT;
       procedure WITHDRAW(CUSTOMER_NAME: CU_NAME; MONEY:out INTEGER) is
         begin
           null;    -- the function to subtract money from CUSTOMER_DATA.MONEY
       end WITHDRAW;
       procedure CHECK_ACCOUNT(CUSTOMER_NAME: CU_NAME; MONEY: out INTEGER) is
         begin
           null;    -- the function to read CUSTOMER_DATA
       end CHECK_ACCOUNT;
       procedure PAY_BILL(CUSTOMER_NAME: CU_NAME; MONEY: in INTEGER) is
         begin
           null;    -- the function to print out the pay bill
       end PAY_BILL;
     end CHG_OPER;

end BANK;
```

Fig. 7 (Continued)

(Received October 1, 1992)
(Accepted September 9, 1993)



Yoshihiro Matsumoto was born in 1932, received B.E. and Dr. Eng. degrees both from the University of Tokyo in 1954 and 1974 respectively. Since 1989, he has been a professor at Department of Information Science, Kyoto University. Before he joined Kyoto University, he was in Toshiba Corporation from 1954 to 1988. His specialities are real-time industrial process control, software engineering (he was a founder of the Toshiba Software Factory), data engineering and information engineering. He received: (1) Fellow Award from the IEEE (the Institute of Electrical and Electronics Engineers Inc.) for leadership in the application of computers in industrial control systems in 1982. (2) New Invention Award from the Japanese Association of Electrical Equipment Manufacturers in 1963. (3) All Japan Invention Award from the Japan Institute of Invention and Innovation in 1981. (4) Excellent Researcher Award from the Science and Technology Agency, Japanese Government in 1982. Dr. Matsumoto is an IEEE Follow, a permanent member of IEEJ, a member of JSSST, and IPSJ.