TR-686

An Integration Environment to Put Formal Specifications into Practical Use in Real-Time Systems

by
S. Honiden, A. Ohsuga & N. Uchihira (Toshiba)

September, 1991

© 1991, ICOT

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

An Integration Environment to Put Formal Specifications into Practical Use in Real-Time Systems

Shinichi Honiden Akihiko Ohsuga Naoshi Uchihira

Systems & Software Engineering Laboratory, Toshiba Corporation,
70 Yanagi-cho, Saiwai-ku, Kawasaki 210, Japan

ABSTRACT

This paper discusses several requirements to put formal specifications into practical use in real-time systems, and an integration environment called MENDELS ZONE to satisfy them. The integration environment includes algebraic specification, temporal logic, real-time Structured Analysis and Object-Oriented Design. It also defines a specification process that assists a designer in translating a specification developed with the real-time Structured Analysis method into an Object Oriented Design specification, and that finally generates Ada tasks. The specification process is defined to consist of several specification steps and several intermediate products. In order to validate the activity for each specification step, each intermediate product is described by two formal specification methods: algebraic specification and temporal logic.

1. Introduction

Of the various software specification methods, formal methods are widely recognized as the methods having the most potential to enhance the quality and the reliability of software. For real-time systems, however, formal methods are not used very much in the industrial environment at present. To put them into practical use, the following five requirements should be fulfilled:

1) It should be possible to combine several formal methods to describe various aspects of the system requested by the user. This is because it is very difficult to do so with only a single formal method.

2) It should be possible to refine abstract specifications into detailed specifications by using formal methods. The refining process can also be regarded as a design methodology to assist formal methods. The process is needed for the following reasons:

a) It enables even inexperienced users to develop desired systems easily by refining formal specifications.

b) Becoming familiar with formal descriptions requires a relatively hard training and much time.

c) If several persons are given the same abstract specification, they should be able to produce similar intermediate specifications as well as compatible final software products.

3) It should be possible to use a simple mechanical method to verify formal descriptions. Also, a means is needed to easily find errors in the specifications.

4) It should be possible to provide a means by which developers can understand the whole, when the size of the desired software system is large, even though formal methods tend to cause designers to focus on details.

5) It should be possible to naturally or inevitably generate a program from a specification described with a formal method. Even if a specification is strictly described, if it does not correspond directly to a program, its practical use is not easy.

A number of studies have been made in the past to satisfy the requirements given above. Regarding methods to effectively combine several formal specifications (requirement 1), a number of proposals have been made (e.g. [Vautherin87], [Kramer87]). However, it cannot be said that these proposals satisfy requirements 2-5. This paper discusses how to satisfy these requirements for a real-time system having concurrent processes. It also describes an integration environment called MENDELS ZONE, which aims at satisfying these requirements.

In order to address requirement 1, a combination of algebraic and temporal logic methods is adopted. The algebraic method describes the characteristics of functions and data which have static aspects, and the temporal logic describes the characteristics of concurrency, synchronization, and exclusive control, which are dynamic aspects. In order to address requirement 2, the real-time Structured Analysis (SA) method [Hatley84, Ward86], widely used for real-time systems, is adopted as the methodology. In real-time SA, the algebraic method is used to describe bubbles in the Data Flow Diagrams (DFDs), and the temporal logic is used to describe the execution control specification. Applying the algebraic method to bubble descriptions in the DFD enables the functional refinement criteria for bubbles to be made clearer, and allows a detailed design process to be defined. As a result, differences between individuals can be made fewer when several persons work together to develop a particular software system. As for the third requirement, although there are various methods for the algebraic approach and temporal logic, our system is limited to an

automatically verifiable subset. Also, the specifications described in the algebraic method are translated into DFD descriptions as much as possible, allowing a visual validation. This is because, depending on the specifications to be verified, visual checks are often more efficient than mechanical, automatic verification. Regarding the fourth requirement, the whole specification can be surveyed by traversing without any restrictions between superior DFDs and inferior DFDs according to the hierarchical relationship. In order to address requirement 5, the concept of the Object Oriented Design (OOD) [Booch86] is adopted. Specifically, a store appearing in a higher-level DFD corresponds to internal state data held in an object. Each bubble around the store is functionally refined into lower-level DFDs, allowing isolation of methods for that object. That is, methods required for data are extracted to construct the object. In addition, the execution order relation among methods is expressed as a state transition diagram. The diagram is generated by a theorem prover, using propositional temporal logic. Concurrent programs are generated from data store definitions, methods, and state transition diagram.

2. Architecture for specification process support in MENDELS ZONE

This section gives an outline of MENDELS ZONE and describes the fundamental techniques to support it.

The method proposed in this paper consists of the following five phases [Fig. 1]:

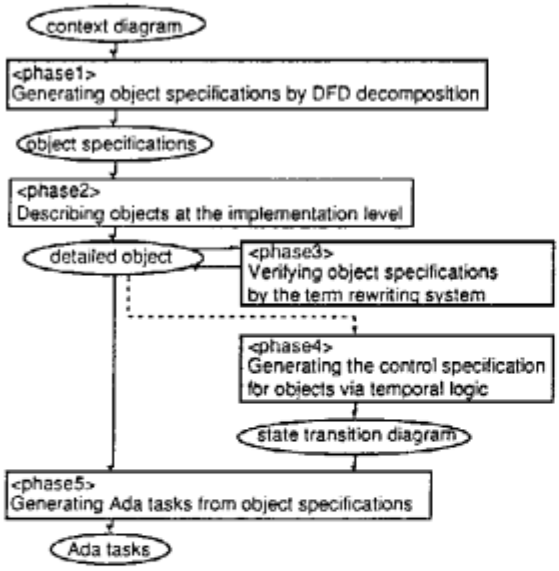


Figure 1 System overview

<Phase 1>

First, the context diagram for the given specification is defined. The diagram corresponds to a single bubble. I/O data that defines the interface between the target system and the outside world is defined via an algebraic specification. Next, the context diagram is decomposed into DFD descriptions. At this point, individual bubble's operations are described using algebraic specifications. Figure 2 gives extended BNF for the syntax of bubble specifications.

```
<object>::=  object:<object name>
             sort:<sort name list>
             opns:<oplist>
             eqns:<eqlist>
<bubble>::=  bubble:<bubble name>
             inSort:<sort name list>
             outSort:<sort name list>
             locSort:<sort name list>
             opns:<oplist>
             eqns:<eqlist>
<object name>::=<name>
<bubble name>::=<name>
<sort name list>::=<name list>
<oplist>::=(<name list>:[<name list>] -> <name>)+
<eqlist>::=(<term>=<term>)+
<name list>::=(<name>,)+<name>
```

Figure 2 Object and bubble syntax

The signature or equation from algebraic specifications can be translated into DFD form and validated visually. In addition, each bubble in a DFD description is decomposed according to the decomposition rules until the termination condition is satisfied. Our method defines the decomposition rules and the termination condition as follows:

(Decomposition Rule 1)

A bubble is decomposed so as to make one operation correspond to one output data item. Generally, as an algebraic specification defines the operation in the form of a function giving no side effect, data returned as a value is limited to one type. Therefore, to describe an algebraic specification, a bubble is decomposed so as to generate only one output data type.

(Decomposition Rule 2)

Decomposition is accomplished based on the description in the right hand part of the equation. The characteristics of a bubble's operation are expressed with an equation. It is assumed that the expression in the right hand part of an equation expresses how the operation works, and at the same time, expresses the decomposition of the operation. For example, equation A(x, y)=D(B(x),C(y)) indicates that A is decomposed into a combination of three operations B, C, D, and means that B processes input data x, C processes input data y, and the processing results are used as input data to D.

(Decomposition Rule 3)

A data store corresponds to an object. Figure 2 also shows the syntax for an object. Data corresponding to an internal state of the system and the operations on the data such as read and update are grouped in the form of objects.

Decomposition on such operations is distinguished from other decompositions.

(Decomposition Rule 4)

A bubble is decomposed so that each bubble accesses only one data store. When the functional decomposition is completed down to the lowest layer, the operations accessing the data store directly are grouped to form an object. According to Decomposition Rule 4, each operation accesses just one data store, so the object to be formed is uniquely determined.

(Termination Condition)

When the right hand part of an equation in a bubble consists of primitive operations or recursive function form, decomposition for the bubble is terminated. A recursive function is not decomposed anymore because it cannot be represented via an ordinary DFD. From our experience, a recursive function usually appears in a lower-level DFD. This means that a recursive function is already fully decomposed.

During Phase 1, functional decomposition on DFDs is repeated according to the above decomposition rules until the termination condition is satisfied, the objects are extracted based on the concept of OOD, and the skeleton of Ada tasks is generated as the final output. Figure 3 shows the basic format for an Ada task.

```
task specification::=
        task <task name> [ is
          { <entry declaration> }
      end [ <task name> ] ] ;
task body::=
      task body <task name> is
        [ <declaration> ]
      begin
        loop
          select
            <select alternative>
        { or
            <select alternative> }
          end select;
        end loop;
      end [ <task name> ];
<select alternative>::=
      [ when <condition> => ]
        <accept statement>
<condition>::=
      <node no. variable> = <integer>
<accept statement>::=
      accept <entry name> [(<expression>)][ <formal
          parameter>] [do
        <sequence of statements>
        <update statement for node no. variable>
      end [<entry name>]];
```

Figure 3 Basic format for an Ada task

The procedure for OOD is given below. The specification process for OOD is roughly divided into the following steps:

(1) Extracting objects.
(2) Defining the object attributes.
(3) Defining the object methods.
(4) Defining the messages between the objects.

Typically, Steps (2) and (3) are not done consecutively; instead they are actually done concurrently or alternately. These steps are detailed below.

(1) Objects are defined based on the data stores appearing in a DFD according to Decomposition Rule 3. The desired Ada tasks are generated based on those objects.

(2), (3) The bubbles around a data store are functionally decomposed to extract the operations closely related to the data store (operations correspond to bubbles in the DFD and methods in OOD). That is, the operations which directly access the data store are extracted. At the same time, the data required for operation extraction is also identified (data corresponds to the sorts in the algebraic specification and the attributes in OOD). If an equation directly accesses the data declared in an object (for read, write, or update), it is automatically registered in the corresponding object by the system as a data access operation. In this way, the operations related to the data defined for an object are specified. When the functional decomposition is completed down to the lowest layer, the operations accessing the data store directly are grouped to form an object. According to Decomposition Rule 4, since each operation accesses just one data store, the object to be formed is uniquely determined. However, the operations that do not directly access any data store do not belong to any object, and must be assigned to the object in which there is the operation that transfers the data.

(4) The interface between two objects corresponds to the data transfer between operations "a" and "b" which belong to objects A and B respectively. Information on the data transfer between operations "a" and "b" is obtained from a DFD generated by a functional decomposition, because the operations here correspond to bubbles and the DFD represents the data flow between operations.

<Phase 2>

At Phase 1, a DFD was decomposed to create objects. Operations assigned to an object generated at Phase 1 do not specify physical data structures or information on implementation. This is because those operations describe data from outside the object and can also be regarded as the operations for abstract data types. Accordingly, at Phase 2, objects are described in detail using equations representing the physical data structure. At Phase 2, the physical data structure is determined first for data contained in the object. In this case, the target application environment needs to be concerned. The descriptions from Phase 1 are the external specifications for operations, while those generated during Phase 2 are the internal specifications for operations.

<Phase 3>

The specification, given in detail using equations at Phase 2, is not often validated visually. The reason is that detailed equations may consist of recursive functions and this form can not be translated into an ordinary DFD. Therefore, some other validation or verification method is required. The verification requires semantics to be given

to the descriptions. The term rewriting system is a model in which operational semantics is given to the specifications described using equations and computation is performed using the equations as rewriting rules from the left hand part to the right hand part. The term rewriting system provides a completion procedure to generate new rewriting rules, so that the set of term rewriting rules provided satisfies termination and confluence properties. Two major points exist for verification here. One is to ensure termination and confluence properties by using the completion procedure in the term rewriting system. The other is to closely examine whether the descriptions (equations) for the object satisfy any verification points given by the user. In both cases, if the completion procedure does not terminate, verification is impossible. Therefore, verification is limited to the range where the completion procedure terminates.

<Phase 4>

A verified object can handle multiple-accesses from several operations. The task mechanism in Ada can suspend the execution of one operation during the execution of another operation. However, this mechanism cannot handle the exclusive control among operations (for example, after an execution of operation "a", operation "c" must not be executed until operation "b" is executed). A graph is generated, which can be regarded as a state transition diagram. Hence, the execution sequence among operations is specified in a state transition diagram which represents the control specification. The control flow in real-time SA is represented with the state transition diagram. As a state transition diagram is usually manually prepared, its validity is not ensured. Our method employs the method proposed by Wolper [Wolper83]. Wolper's method is based on the tableau method. In the tableau method, to decide the satisfiability for the given formula, a model graph, which is a state transition diagram, is constructed. The diagram indicates that the logical expression on an arc is true at its starting node. Therefore, the interpretation of the diagram depends on what is assigned to the logical expression. Here, a data access operation is assigned to the logical expression. In this case, the diagram indicates the execution sequence for operations. That is, the access sequence for the operations to an internal data in Ada is generated from the specification described with temporal logic, by using the theorem prover. In a real-time system, timing constraints must be specified in addition to the control flow, which corresponds to an execution sequence for operations. Hence, RT-PTL(Real-time Propositional Temporal Logic) is adopted for the timing constraint [Honiden90].

<Phase 5>

A task is generated in Ada from the data defined in the object, its associated operations, and the state transition diagram created at Phase 4. A data store in a DFD becomes the internal data hidden by the task, and each operation for the data store becomes an ACCEPT statement in the task. That is, it takes the form of conditional ACCEPT statements waiting for several entry calls from outside. A numerical digit written in a conditional expression following a WHEN statement indicates a node number in a state transition diagram. An update statement for the node number according to the state transition diagram is added to the end of this ACCEPT statement. Also, the equations are translated into functions in Ada.

3. Example

This section describes the specification process for the lift control problem [Problem87] by using the procedure mentioned in Section 2. Phase 1 includes from Step 1 to Step 13, and Phase 2, Phase 3, Phase 4, and Phase 5 correspond to Step 14, Step 15, Step 16, and Step 17, respectively. Figure 4 and Appendix A show a detailed specification process.

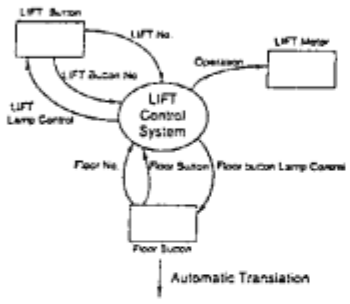
Figure 4 A detailed specification process

The description process is described below.

Step 1: Data I/O between LIFTControlSystem and its outside world is used to specify a context diagram [Fig. 5].

Step 2: The bubble for LIFTControlSystem is described [Fig. 5].

Step 3: The input data items associated with each output data item in LIFTControlSystem are listed. Each I/O pair is assigned a name, and the name is entered in the *opns* entry [Fig. 6]. For input data items not directly associated



bubble: LIFTControlSystem
inSort: LIFTNo.,
LIFTButtonNo.,
FloorNo.,
FloorButton
outSort: LIFTLampControl,
FloorButtonLampControl,
Operation
locSort:
opns:
eqns:

Figure 5 Bubble description for context diagram

bubble: LIFTControlSystem
inSort: LIFTNo.,
LIFTButtonNo.,
FloorNo.,
FloorButton
outSort: LIFTLampControl,
FloorButtonLampControl,
Operation
locSort: LIFTstate
opns: LIFTButtonControl: LIFTNo.,LIFTButtonNo.,LIFTstate
-> LIFTButtonLampControl
FloorButtonControl: FloorNo.,FloorButton,LIFTstate
-> FloorButtonLampControl
LIFTMoveIndicate: LIFTNo.,LIFTButtonNo.,
-> Operation
FloorMoveIndicate: FloorNo.,FloorButton
-> Operation
eqns:

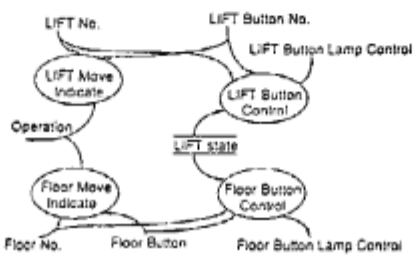
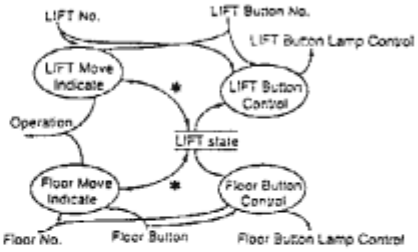


Figure 6 Bubble with definition of operations



bubble: LIFTControlSystem
inSort: LIFTNo.,
LIFTButtonNo.,
FloorNo.,
FloorButton
outSort: LIFTLampControl,
FloorButtonLampControl,
Operation
locSort: LIFTstate
opns: LIFTButtonControl: LIFTNo.,LIFTButtonNo.,LIFTstate
-> LIFTButtonLampControl
FloorButtonControl: FloorNo.,FloorButton,LIFTstate
-> FloorButtonLampControl
LIFTMoveIndicate: LIFTNo.,LIFTButtonNo.,LIFTstate
-> Operation,LIFTstate
FloorMoveIndicate: FloorNo.,FloorButton,LIFTstate
-> Operation,LIFTstate
eqns

Figure 7 Modified bubble description

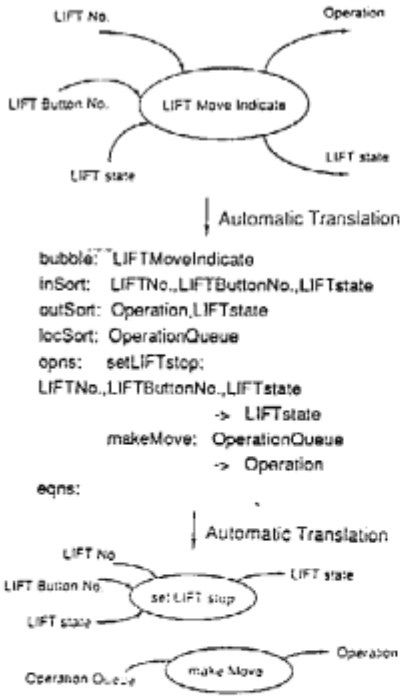


Figure 8 Decomposition process for bubble with two output data

bubble: setLIFTstop
inSort: LIFTNo.,LIFTButtonNo.,LIFTstate
outSort: LIFTstate
locSort: LIFTNo.state
opns: setLIFT: LIFTNo.state,LIFTNo.,LIFTstate
-> LIFTstate
writeStopFloor: LIFTButtonNo.,LIFTNo.state
-> LIFTNo.state
getLIFT: LIFTNo.,LIFTstate
-> LIFTNo.state
eqns: setLIFTstop(L,F,S)
=setLIFT(writeStopFloor(F,getLIFT(L,S)),L,S)

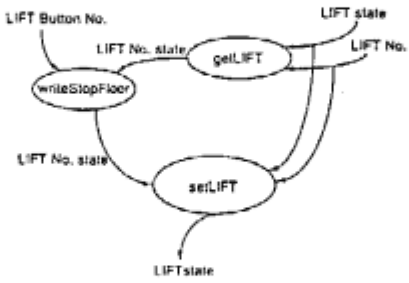


Figure 9 Bubble description with equation for setLIFTstop

object: LIFTstate
sort: LIFTstate,LIFTno.,OperationCommand,OperationStatus,
Direction,LIFTposition, ...
opns: setDirection: LIFTNo.,LIFTstate -> LIFTstate
getDirection: LIFTNo.,LIFTstate -> OperationCommand
setLIFTstop: LIFTNo.,LIFTButtonNo.,LIFTstate -> LIFTstate
setFloorstop: FloorNo.,FloorButton,LIFTstate -> LIFTstate
eqns: setLIFTstop(L,F,S)=setLIFT(writeStopFloor(F,getLIFT(L,S)),L,S)

Figure 10 LIFTstate object description

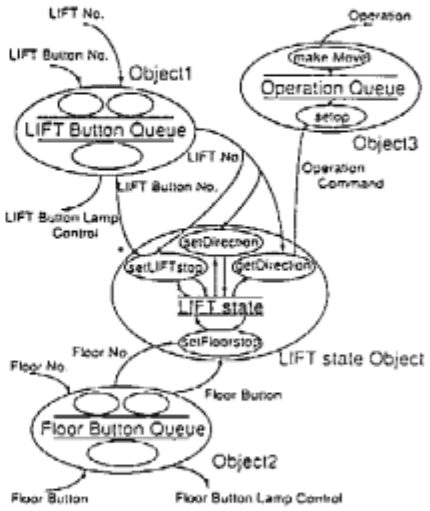


Figure 11 Objects example

with output data items, intermediate data items are created. At this point, LIFTstate for the *locSort* entry is introduced.

Step 4: Each of the four operations in Fig. 6 is associated with each bubble in the DFD, to generate a diagram using an I/O as a data flow [Fig. 6] by the system. In addition, the object for LIFTstate is generated.

Step 5: The diagram in Fig. 6 is examined. The data flow with the asterisk mark shown in Fig. 7 is added.

Step 6: The correction for the diagram is reflected in the bubble description [Fig. 7] by the system. (In this figure, the corrected part is indicated with bold letters.)

Step 7: The decomposed bubbles are specified from the bubble in Fig. 7, which consists of four operations.

Step 8: For the two bubbles; that is, LIFTMoveIndicate and FloorMoveIndicate, for which several output data items result from the correction made in Step 6, the procedure given in Step 3 is used again to accomplish functional decomposition. In Fig. 8, for appropriate bubbles, LIFTMoveIndicate is decomposed into two operations: setLIFTstop and makeMove. Figure 8 shows the decomposed DFD. For each bubble having one output after decomposition, a bubble description is specified.

Step 9: Of the newly defined bubbles, the relation between the input data and output data for setLIFTstop is described in Fig. 9. Here, LIFTNo.state is introduced as a new data item, and several new operations: setLIFT, writeStopFloor, and getLIFT are defined to construct setLIFTstop.

Step 10: Figure 9 shows the DFD based on the equations for setLIFTstop.

Step 11: setLIFTstop is an operation for LIFTstate, so it is entered by the system as a constituting element in an object called LIFTstate. By decomposing each individual bubble around LIFTstate, operations which directly access LIFTstate are extracted.

Step 12: If all operations for the LIFTstate object are extracted as shown in Figure 10, go to Step 13.

Step 13: Figure 11 shows the relation among the LIFTstate object and other objects.

```
setLIFTstop(LN,BN,SS)=setLIFT(writeStopFloor(BN,getLIFT(LN,SS
)),LN,SS)
writeStopFloor(BN,'NIL')='NIL'
writeStopFloor(BN,'LS'(FS,Rem))=if eq(BN,getButtonNo(FS)) then
    'LS'(setButtonState('STOP',FS),Rem)
else
    'LS'(FS,writeStopFloor(BN,Rem))
readStopFloor(LN,BN,SS)=readStopFloor(BN,getLIFT(LN,SS))
readStopFloor(BN,'NIL')='ButtonNoError'
readStopFloor(BN,'LS'(FS,Rem))=if eq(BN,getButtonNo(FS)) then
    getButtonState(FS)
else
    readStopFloor(BN,Rem))
setLIFT(NLS,LN,'NIL')='NIL'
setLIFT(NLS,LN,'SS'(LS,Rem))=if  eq(LN,getLiftNo(LS)) then
    'SS'(setLiftState(NLS,LS),Rem)
else
    'SS'(LS,setLIFT(NLS,LN,Rem))
```

Figure 12 Detailed description for setLIFTstop in the LIFTstate object

Step 14: In this step, detailed descriptions are developed with the physical data structures in LIFTstate taken into account. A part of description is shown in Fig. 12. Before describing the specifications, the structure for LIFT (how many floors does the building have, how many lifts are there in the building, and how is LIFTstate expressed) is determined and reflected.

Step 15: The description developed in Step 14 is verified using the term rewriting system. Figure 13 shows the meanings for verification points and their descriptions.

```
readStopFloor(L,F,setLIFTstop(L,F,S))='stop'
      where   L: LIFTNo.
              F: LIFTButtonNo.
              S: LIFTstate
```

(When a request button is pushed for any floor in any lift, the requested floor is added in the list indicating the floor at which the lift stops)

Figure 13 Verification point example

Step 16: For objects which have been verified using the term rewriting system, it is assumed that the order relation among operations is correctly specified. However, requests from outside LIFTstate can occur at random. Therefore, for example, the following control is required: If setLIFTstop is executed, setDirection must not be executed before executing getDirection. As shown in Fig. 14, a request for such a control is described using the propositional temporal logic. The state transition diagram shown in Fig. 14 is generated using the theorem prover for the propositional temporal logic.

□ (set LIFT stop ⊃ (¬ set Direction U get Direction))
□ (set Floor stop ⊃ (¬ set Direction U get Direction))
□ (set LIFT stop ⊃ ○ ◇ get Direction)
□ (set Floor stop ⊃ ○ ◇ get Direction)

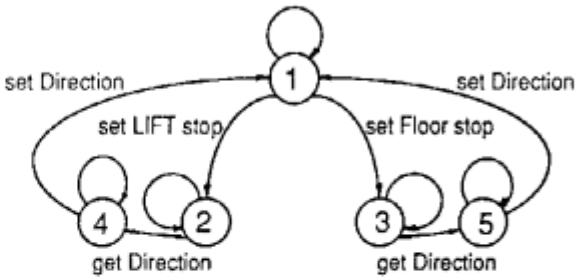


Figure 14 Temporal logic specification and generated state transition diagram

Step 17: As shown in Fig. 15, the LIFTstate object is generated as an Ada task. At this point, from the state transition diagram generated in Step 16, the conditional statements associated with ACCEPT statements are created using numerical digits. Each individual operation

expressed as an equation is translated into an Ada function.

```
task LIFTstate is
entry setLIFTstop     (L: in LIFTNo.; B:in LIFTButtonNo.;S: in
                       LIFTstate; S: out LIFTstate);
   entry setFloorstop  (--);
   entry getDirection  (--);
   entry setDirection  (--);
end LIFTstate;
   -
task body LIFTstate is
   N:integer:=1;  --node variable in state transition diagram
   NS: LIFTNo.state;
begin
   loop
      select
         when N=1    =>
           accept setLIFTstop (L: in LIFTNo.; B: in
           LIFTButtonNo.; S: in LIFTstate; S: out LIFTstate)
           do
              NS:=getLIFT(L,S);
              NS:=writeStopFloor(B,NS);
              S:=setLIFT(NS,L,S);
              N:=2;   --update node variable
           end setLIFTstop;
      or
         when N=1    =>
           accept setFloorstop (--) do
              -
              N:=3;
           end setFloorstop;
           -
      end select;
   end loop;
end LIFTstate;
```

Figure 15 Output example in Ada task

4. Related work

We have adopted an integration method for formal approaches. Several integration methods for formal approaches have been presented [Vautherin87] [Kramer87] [Hankley90] [Meiling87] [Folkjar80]. Each method uses VDM and algebraic specification for data description and uses temporal logic, Petri-nets, CCS, or CSP for the description of synchronization and concurrency. However the verification method cannot always be said to be well integrated for these approaches. Although LOTOS [Iso89] supports the above two types of description in terms of language, the verification system has problems to be settled in the future. Our method utilizes the algebraic specification method for data description and temporal logic for the description of synchronization. Since our objective is to implement it as a software system, the descriptive power of the system is limited to the range in which analysis is possible, that is, a mechanically verifiable range. Hence, propositional temporal logic is adopted. The algebraic specification method limits the descriptions to the range which can be verified using the term rewriting system. Also, to improve the readability of the formal specification, DFD is adopted as a visual validation method. The DFD also enables the user to perform visual validation on complex algebraic descriptions. [Docker89] combines equational logic and DFDs. Our method differs from [Docker89] in that the specification process is defined by combining the algebraic specification and DFD. P. Ward discussed the relation between real-time SA and OOD [Ward89]. He suggested that OOD can be supported within the framework of the design methodology for real-time SA. That is, a data store is defined as the internal data in the DFD at the lowest layer, and bubbles around the data store are taken as operations, thus data and its associated operations are regarded as objects. This is similar to our method, but the difference is that in our method operations are extracted by repeating functional decomposition systematically and operations are included in the object uniquely and automatically.

Regarding support tools for formal specifications, there have been a number of approaches based on use of syntax editors, specification libraries, debugger, verification tests generation systems, and direct execution systems. Of the support tools provided by our method, there are two tools that are not provided by the other methods. One is the bidirectional consistency support tool between the algebraic specification and DFD. The other is the navigator for the functional decomposition via DFDs.

5. Conclusion

Our method has the following constraints:
1) Bubbles must have only one output item.
2) The control specification must be restricted to the order relation for the operations in the Ada task.
3) As a constraint for the verification system, verifiable range is limited.

MENDELS ZONE is now being implemented. It has a window displaying a DFD and a window through which an algebraic specification is described. Information in these two windows is consistent; if the user modifies the information in one of the windows, the system automatically modifies the information in the other window. In addition, there is one more window for describing the details of data stores, intermediate data, and I/O data from the outside world. Regarding the description using equations obtained by decomposing the bubbles around a data store, the operations which directly access the data store are automatically entered in that additional window by the system. To apply formal specification methods in practice so that designers really use them, a support environment must be prepared. In our system, the support environment navigates the detailed specification process. Also, using formal specification methods together with the diagrammatic specification method is effective for the designer.

Acknowledgements

This research has been supported in part by the Japanese Fifth Generation Computer Project and its organizing institute ICOT. The authors are grateful to Seiichi Nishijima and Yutaka Ofude of Systems & Software Engineering Laboratory, Toshiba Corporation. The authors are indebted to Dr. Robert Babb, Oregon Graduate Institute, for helpful comments and suggestions.

References

[Booch86] C.Booch, Object-Oriented Development, *IEEE Trans. Software Eng.*, Vol.SE-12, No.12,1986

[Docker89] T.W.G.Docker, Flexibility and Rigour in Structured Analysis, *IFIP'89*, 1989

[Folkjar80] P.Folkjar and D.Bjorner, A Formal Model of a generalized CSP-like language, *IFIP'80*, 1980

[Hankley90] W.Hankley and J.Peters, Temporal Specification of Ada Tasks, *HICSS-23*, 1990

[Hatley84] D.Hatley, The Use of Structured Methods in the Development of Large Software Based Avionics Systems, *AIAA/IEEE sixth Digital Avionics Systems Conf.*, 1984

[Honiden90] S.Honiden et al., An Application of Structural Modeling and Automated Reasoning to Real-Time Systems Design, *The Journal of Real-Time Systems*, Vol.1, No.3, 1990

[Iso89] ISO, Information Processing Systems-Open Systems Interconnection-LOTOS-A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, *ISO8807*, 1989

[Kramer87] B.Kramer, SEGRAS-A Formal and Semigraphical Language combining Petri Nets and Abstract Data Types for the Specification of Distributed Systems, *9th ICSE*, 1987

[Meiling87] E.Meiling, A spreadsheet specification in RSL-an illustration of the RAISE specification language, *In ESPRIT'87 Achievements and Impact*, North-Holland, pp. 466-479, 1987.

[Pletat86] U.Pletat, Algebraic specification of abstract data types and CCS : An operational junction, *In Protocol Specification, Testing and Verification*, Elsevier science Publishers, 1986

[Problem87] Problem set, *Proc. of Fourth International Workshop on Software Specification and Design*, CS Press, Los Alamitos, Calif. 1987

[Vautherin87] J.Vautherin, Parallel Systems Specifications with Coloured Petri Nets and Algebraic Specification, *LNCS 266*, 1987

[Ward86] P.Ward, The Transformation Schema : An Extension of the Data Flow Diagram to Represent Control and Timing, *IEEE Trans. Software Eng.*, Vol.12, No.2, 1986

[Ward89] P.Ward, How to Integrate Object Orientation with Structured Analysis and Design, *IEEE Software*, Vol.6, No.2, 1989

[Wolper83] P.Wolper et al., Synthesis of Communicating Processes from Temporal Logic Specification, *ACM TOPLAS*, Vol.6, No.1, 1983

Appendix A. Specification process in Phase 1

Step 1: I/O data between the target system and its outside world is specified to write a context diagram.

Step 2: The bubble expression is specified for the target system. Input and output data for the outside world are described in the *inSort* entry and *outSort* entry respectively.

Step 3: According to Decomposition Rule 1, one output data item for the bubble is selected, and all input data items that are expected to have effects on that data item are picked out. An operation having those input data items in its domain and having the output data in its range is described by giving it a name in the *opns* entry for the bubble. In this case, if an introduction of intermediate data is required, it is assigned an adequate name and described in a *locSort* entry.

Step 4: A DFD is created, in which each operation is used as a bubble subprocess and the I/O for each operation is used as the data flow. In this case, if any data store exists in the diagram, according to Decomposition Rule 3, it is intentionally described as an object, and all the subsequent operations that directly access the data store are defined in the object.

Step 5: The DFD is examined; if there is any part missing from it, it is fixed in the diagram. If not, proceed to Step 7.

Step 6: The corrections made in the diagram in Step 5 are reflected into the bubble description.

Step 7: The decomposed bubbles are created from original bubble's operation.

Step 8: If the correction made in Step 6 causes several output data items to be generated, get back to Step 3 and carry out the functional decomposition again. If every bubble (operation) has one output (in the *opns* entry) and no more correction is required, go to Step 9.

Step 9: Regarding each object, the relation that exists between the input data and output data for each operation is examined. At this point, the function for each operation is defined in the *eqns* entry in the form of equation.

Step 10: If the equation in the *eqns* entry for the operation having the name of the bubble itself is expressed in a recursive function, the bubble is not decomposed any more. If the equation is expressed in non-recursive function, according to Decomposition Rule 2, a DFD is created.

Step 11: If the equation described in the *eqns* entry directly accesses the data declared in a object, the system registers the equation in the object as a data access operation. Thus, the operations on the data defined in the object are extracted.

Step 12: If all operations other than the primitives are already defined in the *eqns* entry and there are not any more bubbles to be decomposed, go to Step 14. In other cases, go to Step 5.

Step 13: Every operation is joined to the appropriate objects.