ICOT Technical Report: TR-0835

TR-0835

Net-Oriented Analysis and Design

by
S. Honiden & N. Uchihira (Toshiba)

March, 1993

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191～5

Institute for New Generation Computer Technology

Net-Oriented Analysis and Design

Shinichi Honiden
Naoshi Uchihira

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

Keywords:

Petri nets, State Transition Diagram, Data Flow Diagram, Algebraic Specification, Temporal Logic, Object-Oriented Analysis and Design, CASE

Abstract

Net-Oriented Analysis and Design (NOAD) is defined as three items:
(1) Various nets are utilized as an effective modeling method.
(2) Inter-relationships among various nets are determined.
(3) Verification or analysis methods for nets are provided and they are implemented based on the mathematical theory, that is Net theory.
Very few methods have been presented to satisfy these three items. For example, the Real-Time SA method covers item (1) only. The Object-Oriented Analysis and Design method (OOA/OOD) covers items (1) and (2). NOAD can be regarded as an extension to OOA/OOD. This paper discusses how effectively various nets have been used in actual software development support methods and tools and evaluates such several methods and tools from the NOAD viewpoint.

1. Introduction

Systematic methodologies and efficient support tools play an essential role for a group of software developers to work together to create a large scale software system. Software engineering is intended to establish the technology to embody an engineering (not a handicraft) approach that will enhance software productivity and reliability. However, so far the results are limited to the lower stream of the software development process such as programming and quality assurance. In order to increase software productivity, it is necessary to establish the technology to support the analysis and design phases.

Recently, Computer Aided Software Engineering (CASE) to support the analysis and design phases has been attracting software developers' attention; as high-performance workstations become popular, a number of such CASE tools have been commercialized on them. One of the major characteristics of these tools is the use of various nets as a modeling technique. Various nets example are data-flow diagram, state transition diagram, and Petri nets.

We define Net-Oriented Analysis and Design (NOAD) as three items:
(1) Various nets are utilized as an effective modeling method.
(2) Inter-relationships among various nets are determined.
(3) Verification or analysis methods for nets are provided and they are implemented based on the mathematical theory which is called Net theory.

Very few methods have been presented to satisfy these three items. For example, the Real-Time SA method covers item (1) only. The Object-Oriented Analysis and Design method (OOA/OOD) covers items (1) and (2).

NOAD can be regarded as one of the methods extending OOA/OOD. That is, it can be defined by the following formula:

NOAD = OOA/OOD + Net theory.

This paper discusses how effectively these various nets have been used in the actual software development support methods and tools and evaluates such methods and tools from NOAD viewpoint.

First, how various nets are utilized is presented using the Object Oriented Analysis method (OOA) as an example. Secondly, how analysis and design activities are performed using various nets in Object-Oriented CASE (OO-CASE) is explained. Thirdly, how the nets are exploited in the software development environment for a concurrent system is discussed. Finally, the problems that should be solved to actually implement the NOAD are described.

2. Object Oriented Analysis (OOA)

The analysis phase is the first phase of a software life cycle. The analysis phase starts with analyzing the problems described in a natural language. This problem description is often ambiguous and incomplete. These ambiguity and incompleteness are eliminated and at the same time the scope of the system (including the boundary between the system and the outside world) is identified. The problem is better described using certain formal models rather than in a natural language. Generally, it is necessary to model the static structure, the event execution sequence, and the data translation.

The OOA uses the object model, the state transition model (a dynamic model), and the functional model (process model) for the static structure, the event execution sequence and the data translation, respectively. The purpose of OOA is to analyze the application domain according to the given problem description and based on these three models, and also to produce a requirements specification that states what, in the application domain, should be built as the system. An OBJECT plays the most important role in this task and therefore becomes the unit (the OBJECT) for modeling. An OBJECT consists of name, attribute (data), and operation (service or method).

Using the OBJECT, the analysis process proceeds according to the following steps, although the details may differ depending on individual OOAs.

- Building the OBJECT model
The OBJECT model is composed of individual OBJECT's and the static relation between them, and is developed based on the Entity-Relationship Diagram (ERD).

- Building the state transition model

Since an OBJECT essentially has a state, the dynamic behavior of the OBJECT can be represented using the State Transition Diagram (STD).

- Building the data translation model

The detailed behavior of individual OBJECTs and the message communication between OBJECTs are represented using the Data-Flow Diagram (DFD). The data flow diagram, which is used in the structured analysis (SA) technique [Demarco78], is a diagrammatic description method that enables easy drawing and intuitive understanding of the contents. The SA technique, proposed by Tom DeMarco in 1978, consists of the data-flow diagram, the mini specification, and the data dictionary (DD). The description of a system starts with drawing a special data flow diagram called the context diagram. Next, each process on the context diagram is decomposed into sub-processes to develop the second level data flow diagrams. The data-flow diagrams consist of processes (bubbles), flows of data, and data storages. The SA technique repeats this functional decomposition on data-flow diagrams to design a system. Data-flow diagrams are adopted in many CASE tools because of their high readability and understandability.

Several OOAs have been proposed including Mellor's OOA [Shlear88], Coad's OOA [Coad91], and Rumbaugh's OMT [Rumbaugh91]. The first version of Mellor's OOA was released in 1979 and since then, has been used in the analysis process in the development of a number of large scale real-time systems. In the course of its practical use, several improvements were introduced, and the current version finally came out in 1986. This OOA uses three models; the information model, the state model, and the process model. First, OBJECTs are identified and the static relationship between them is established using the information model. Next, the state model and the process model are used to recognize the dynamic aspects of each OBJECT and to describe the specification.

The information model is based upon the ER (entity relationship) model, and regards each entity and relationship as an OBJECT. Candidates for OBJECTs are tangible items, roles, incidents, interactions, and specifications. Extracted OBJECTs are reviewed for their validity as OBJECTs and at the same time the attributes are defined for each OBJECT. Then, the attributes are reviewed to build a class hierarchy (corresponding to the "is-a" relationship). Meanwhile, the relationship (corresponding to the R in the ER model) between OBJECTs is also defined using an OBJECT (this OBJECT is called an associative OBJECT). The OBJECTs and their relationships thus defined are called the information model.

After the information model is defined, changes in the states of individual OBJECTs are represented with the state transition diagrams (the state model) as the second stage of the analysis. Then, the detailed actions in each state of

the state transition diagram are represented with the DFD (the process model). Each data store in the DFD corresponds to one of the OBJECTs.

Conventional methods (e.g., the real-time SA [Ward86]) also use the ERD, STD, and DFD notations. The use of nets in the Mellor's OOA is different from that of the real-time SA in that modeling is performed from three different viewpoints (static relationship with other OBJECTs, behavior within the OBJECT, and functions of the OBJECT) for the same "OBJECT". Another difference is that the three models have close relations with each other. For example, the first step is to elicit attributes and attribute values for the entities that correspond to the objects in the ERD. Next, the attribute that determines the primary nature of the object is selected to make its changes correspond to the state changes. These state changes can be represented with the state transition diagram. That is, changes in the attributes of an OBJECT in the ERD are represented with the state transition diagram, which also represents the internal behaviors of the OBJECTs. Actions that trigger the state changes of an object correspond to the arrivals of external messages (by which the required operations within the OBJECT are selected and executed). It is possible to represent the detailed actions in each state or the detailed actions prompting the state changes, with the DFD, as with the Mellor's OOA.

It is also known that the Petri nets model is useful for inter-OBJECT or inside-OBJECT descriptions in the analysis of concurrent/parallel systems.

In summary, various nets have been introduced for OOA. However, they are presently used only as tools for the modeling; net theories for various nets are not yet utilized.

3. How to Use Nets with Object-Oriented CASE

In this Section, how analysis and design activities are performed using various nets in Object-Oriented CASE (OO-CASE) is explained.

To illustrate the OO-CASE, Case for Object-Oriented Analysis and Design (COOAD) which is an object-oriented analysis and design tool being developed at our laboratory is used as an example. COOAD is comprised of the following four phases.

<Phase 1> : Create an OBJECT configuration diagram.
<Phase 2> : Develop the OBJECT architecture.
<Phase 3> : Develop the OBJECT design.
<Phase 4> : Generate code in the object-oriented language C++.

The input to COOAD is a requirements statement. In <Phase 1>, which corresponds to the object-oriented analysis phase, an OBJECT configuration diagram is created. In this diagram, the "is-a" and "has-a" hierarchies are defined. Defining a class consists of naming the class, its attributes, enumerating the attribute values, and specifying the operation names.

<Phase 2> allocates OBJECTs in and out of the system boundary and then refines each OBJECT. When refining an OBJECT, specific relationships with other OBJECTs are defined; that is, the OBJECT architecture is constructed.

<Phase 3> refines each OBJECT in the OBJECT architecture defined in <Phase 2> based on their relationships. Since the message flow among OBJECTs is also defined, an OBJECT in this phase can be regarded as an instance. Unlike ordinary object-oriented designs in which OBJECTs are refined at the class level, COOAD refines OBJECTs at the instance level. The results of this refinement are, naturally, reflected in the information of the classes represented in the OBJECT configuration diagram in <Phase 1>.

The OBJECT refinement performed in <Phase 3> corresponds to the operation refinement. New attributes are added while refining the operations. The design of an operation is developed from the viewpoint of handling of messages received from other OBJECTs. Before starting the message design, it is necessary to elicit all the operations comprising the OBJECT. OBJECTs have states and changes in these states can be represented with the state transition diagram. It is important to determine how the diagram should be drawn; in other words, it is important to define what the state change really mean. One of the aims of COOAD is to re-use previously created information as much as possible. When drawing the state transition diagram, we use the information about attribute values which were enumerated in <Phase 1>. This information for the attributes (often called primary identifiers) which characterizes the OBJECT, is provided to the user. The state transition diagram is developed so that each individual attribute value will correspond to a different state in the diagram. An action which changes the state in the state transition diagram corresponds to a operation in OBJECT (See Figure 1).

After the operations are elicited, the relationship between the operations is examined using the data flow diagram. In the data flow diagram, a group of the attributes of an OBJECT is represented as one data store. New bubbles are also generated in the course of refining the diagram. These bubbles represent the OBJECT's operations created at the design stage. The data flow between the bubbles (operations) that belong to different OBJECTs correspond to the message between the OBJECTs. Figure 1 shows the screen of <Phase 3>; in some cases, the bubbles are decomposed in this screen. COOAD will guide the decomposition using several pre-determined rules.

In <Phase 4>, a skeleton of the desired program is generated in C++.

As mentioned above, COOAD provides several intermediate products represented by various nets in analysis and design phases. COOAD also allows various nets to be utilized with each other.

4. MENDELS ZONE

MENDELS ZONE is a software development environment for concurrent programs [Honiden90, Uchihira87, Uchihira90a]. The target language, MENDEL, is a concurrent programming language based on Petri nets. A concurrent program consists of a section for its functions and one for synchronization. The former is written using algebraic specifications and the latter using temporal logic [Honiden92]. Also, in MENDEL, the former corresponds to MENDEL components and the latter corresponds to synchronization mechanism among MENDEL components. In order to solve the problems specific to each formal specification, data flow diagrams are used for the algebraic specification and Petri nets for the temporal logic. This section discusses each of these specifications.

4.1 Generation of MENDEL Components

The algebraic specification method is adopted as a method for describing the formal specification; the formal description of the data flow within a system is developed using abstract data types. The algebraic specification method, which is theoretically based on many-sorted algebra, was proposed around 1975 as a method for specifying abstract data types. The many-sorted algebra consists of a family of the same type of data sets (sorts) and a set of operations on these sorts. The meaning of each operation is given by equational logic. The meanings of the operations are defined based on the concept known as initial algebra.

An algebraic model which consists of sorts and operations has a close relationship to the data flow model of the SA technique. Fusing both models makes it possible to integrate the data specification and the functional specification, thus compensating the lack of data descriptive power in the SA technique. [Honiden91] has proposed a description method that allows designers to describe the specification at any required abstract level, using readable models of the SA technique. It also addresses how to decompose the functions.

The SA technique describes a system with a model consisting of processes and the data flows among them. At this stage, data flow diagrams play a major role. On the other hand, the algebraic specification plays a major role in describing the abstract data types, which is a method of describing a system with a model consisting of the sorts and the operations on them. At this stage,

signature graphs are used to express the relationships between sorts and operations.

Figures 2 and 3 show an example of a data flow diagram and a signature graph. The diagram in Figure 2 represents a system in which five kinds of data, a - e, flow among three processes X, Y, and Z. On the other hand, the diagram in Figure 3 represents a data type having five kinds of sorts, a - e, and four kinds of operations, X1, X2, Y, and Z. These two diagrams are interchangeable if the arrows and the bubbles are exchanged. However, X must be divided into X1 and X2. This division is derived from the fact that an operation in the algebraic specification is defined to return data of one sort as its return value. The viewpoint of decomposition by output data can become a guideline for the functional decomposition in the SA technique.

The differences of these quite similar diagrams are due to the differences in characteristics between the SA technique and the algebraic specification. In short, while the data-flow diagram is effective for designing a system while grasping the entire system, the signature graph is effective in describing the nature of data based on the formal semantics (in this case, the semantics based on the many-sorted algebra). It should be noted that a data flow diagram is not equivalent to a signature diagram. A data flow diagram which can be produced from a signature graph is limited to a simple one without data storages. Furthermore, it is not an intelligible way of finding a hierarchy in a signature graph that can be produced by converting a data flow diagram.

We therefore proposed a method for describing system specifications by fusing a readable and intelligible model of the SA technique and formal data descriptions by algebraic specification, and showed how to compensate for the defects of the two techniques while maintaining their advantages [Honiden91]. In this method, a bubble's operations are described using algebraic specifications. Figure 4 gives the syntax of bubble specifications in extended BNF.

Also, we proved that using the two models together by relating them to each other would yield a clear guideline for the functional decomposition criterion that had been a problem with both techniques. The signature or equation from algebraic specifications can be translated into the DFD form and visually validated (See Figure 5). 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 rule as follows.

(Decomposition Rule 1)
A bubble is decomposed so as to make one operation correspond to one output data item. Since an algebraic specification defines the operation in the form of a function, returned data 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 done based on the description in the right hand part of an equation. The characteristics of a bubble's operation are expressed by 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 their results are used as input data to D.

(Decomposition Rule 3)
A data store corresponds to an object. The syntax for an object is also shown in Figure 4. 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 an object. 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, operations directly accessing one data store are grouped to form an object. This object is uniquely determined since each operation accesses just one data store.

(Termination Rule)
When the right hand part of an equation in a bubble consists of primitive operations or recursive functions, the decomposition of 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 appears in a lower-level DFD.

Using above rules, we defined a detailed specification process. Figure 6 shows the whole specification process and Appendix A gives an explanation of each step.

Functional decomposition on DFDs is repeated according to the above decomposition rules until the termination condition is satisfied. The MENDEL components are extracted based on object-oriented design concepts; generated MENDEL components are stored in a component library.

4.2 Generation of Synchronization Mechanism among MENDEL Components

This section discusses a component reuse system which retrieves and combines the MENDEL components stored in a component library, to generate a target MENDEL program [Uchihira87, Uchihira90a]. A MENDEL program generated simply by combining the components (such a program is called a body-part) satisfies the functional requirements but does not handle synchronization. Consequently, it may cause a deadlock. To compensate for this deficiency, a synchronization specification (such as a deadlock-free mechanism) is described based on the temporal logic and, to satisfy the requirement, a part (called the synchronization part) that controls the synchronization between components is automatically generated using the theorem prover. Since the validity of each component is guaranteed, a MENDEL program generated by combining the body parts and the synchronization part is also assured of satisfying both of the function and synchronization requirements. The MENDEL program thus generated is eventually compiled into KL1 code to execute on a parallel machine.

4.2.1 Generation of Body Part

Each MENDEL component can be represented with Petri nets. The combination of components is done with the Petri nets editor (See Figure 7). MENDELS ZONE provides the component management browser and the component combination aid subsystem, GARNET, as support facilities for the component retrieval and combination phase (the body part generation phase).

(1) Component Management Browser

It manages and retrieves the external interfaces of individual components and displays the components from several viewpoints.

(2) GARNET

It selects candidate components to be combined with each other by examining the syntactic and semantic consistency according to the data specifications (represented with a semantic network) given for the external interfaces of individual components.

4.2.2 Generation of Synchronization Part

When combining components in the body part generation phase, the data consistency between components was of primary concern; no consideration was given to their synchronization. It may lead to problems such as deadlock. To avoid such problems, the synchronization part is generated by describing the synchronization specification with temporal logic.

(1) Temporal Logic and Tableau Method

Temporal logic stems from classical logic, extended by adding the temporal operators to handle time. Linear time Propositional Temporal Logic (LPTL) has been adopted for the synchronization part. The LPTL allows the following descriptions:

- presence or absence of a deadlock (e.g., action A occurs any number of times.),
- order of actions (e.g., action B occurs following action A.), and
- prohibition of actions (e.g., Once action A has occurred, action B will never occur.).

The tableau method, a theorem prover of the temporal logic, is available for generating the sequence of actions in a concurrent program that satisfies the specifications described with the temporal logic.

(2) How to Generate The Synchronization Part

The structure of the body part is extracted as Petri nets. The synchronization part is formulated as a procedure which generates the transition firing sequence of the Petri nets so that the Petri nets extracted from the body part will satisfy the specifications described with LPTL expressions. [Uchihira90b] shows that there is an algorithm to generate, from given Petri nets and the temporal logic, a firing sequence which satisfies both of them. This algorithm is an extension of the tableau method. The generated synchronization part is represented by a collection of all the firing sequences which satisfy both the Petri nets and temporal logic.

5. Conclusion

In this paper we have discussed how effectively various nets are used in the actual software development support methods or tools.

We evaluate such methods and tools from the NOAD viewpoint. In order to implement the NOAD based on OOA/OOD, OOA/OOD should adopt various nets which have the verification or analysis method based on net theory. Next, OOA/OOD should be re-constructed using the adopted nets as modeling techniques. Finally, the range of the verification or analysis based on net theory should be determined. In order to increase this range, some other formal method is required in addition to net theory. Therefore, the combination of net theory and other formal methods is also applicable.

As for the COOAD method described in Section 3, we are investigating the application of algebraic specifications and its verification methods to the various nets in order to increase the verification range.

We studied the combination of net theories and other formal methods in MENDELS ZONE, but did not reach the phase where net theory was utilized effectively.

Acknowledgements

The authors would like to thank Professor Kenji Onaga of Hiroshima University and Professor Sadatoshi Kumagai of Osaka University who originally presented the concept of Net-Oriented Analysis and Design. 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, for providing continuous support.

References

[Coad91] P.Coad, E.Yourdon, Object-Oriented Analysis: Second Edition, Prentice-Hall.1991

[Demarco78] T.DeMarco, Structured Analysis and System Specification, Yourdon, New York, 1978

[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

[Honiden91] S.Honiden et al., An Integration Environment to Put Formal Specification into Practical Use in Real-Time Systems, Proc. of the 6th International Workshop on Software Specification and Design, 1991

[Honiden92] S.Honiden et al., An Integration Method of Real-Time SA and Object Oriented Design Using Algebraic and Temporal Specifications, Trans. IPSJ, Vol.33, No.2, 1992 (in Japanese)

[Rumbaugh91] J.Rumbaugh et al., Object-Oriented Modeling and Design, Prentice-Hall, 1991

[Shlaer88] S.Shlaer, S.J.Mellor, Object-Oriented System Analysis: Modeling the World in Data, Prentice-Hall,1988

[Uchihira87] N.Uchihira et al., Concurrent Program Synthesis with Reusable Components Using Temporal Logic, Proc. 11th COMPSAC, 1987

[Uchihira90a] N.Uchihira et al., Synthesis of Concurrent Programs: Automated Reasoning Complements Software Reuse, Proc. 23th HICSS, 1990

[Uchihira90b] N.Uchihira et al., Verification and Synthesis of Concurrent Programs Using Petri Nets and Temporal Logic, Trans. IEICE, Vol.E73, No.12, 1990

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

Appendix A. Specification Process

Step 1: I/O data between the target system and its outside world is specified in 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* and *outSort* entries 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 effect that data item are selected. 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 the introduction of an intermediate data is required, it is assigned an appropriate name and described in the *locSort* entry.
Step 4: A DFD is created; 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; any missing parts are added to the diagram. If there are none, 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 the original bubble's operation.
Step 8: If the corrections made in Step 6 cause several output data items to be generated, return to Step 3 and carry out the functional decomposition again. If every bubble (operation) has one output item (in the *opns* entry) and no more corrections are required, go to Step 9.
Step 9: 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 as an equation.
Step 10: If the equation in the *eqns* entry for the operation having the name of the bubble itself is expressed as a recursive function, the bubble is not decomposed any more. If the equation is expressed as a 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 no more bubbles to be decomposed, go to Step 13. Otherwise, go to Step 5.
Step 13: Every operation is joined to the appropriate objects.

<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 4: Object and bubble syntax

Figure 1: MMI example of COOAD

Figure 2: An example of a data flow diagram

Figure 3: An example of a signature graph

Figure 5: An algebraic specification and its DFD translation in MENDELS ZONE

Figure 6: A detailed specification process
(Note: A directed straight line indicates data flow, a directed dashed line indicates conditional control flow.)

Figure 7: MENDEL components represented with Petri nets in MENDELS ZONE

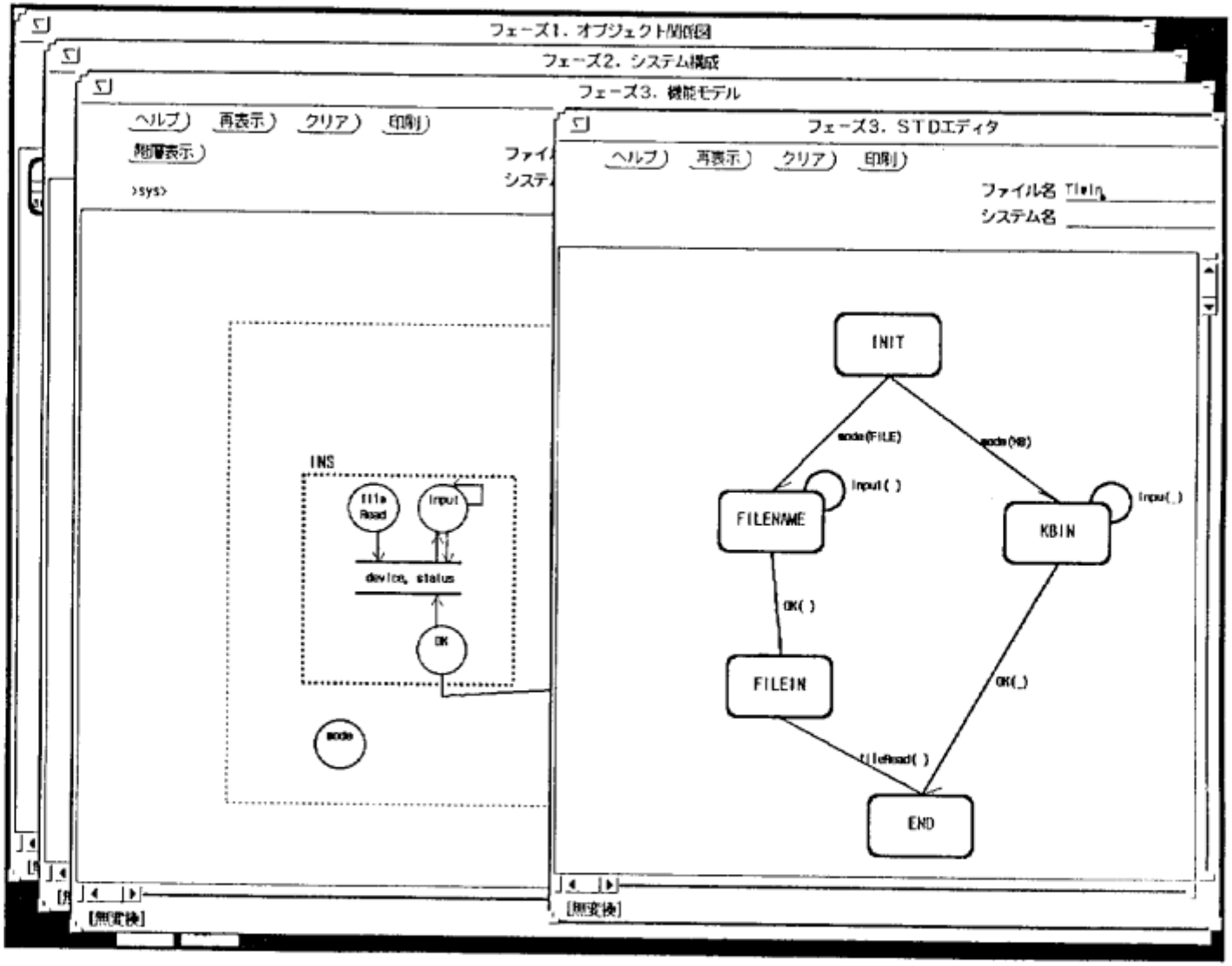

Figure 1: MMI example of COOAD

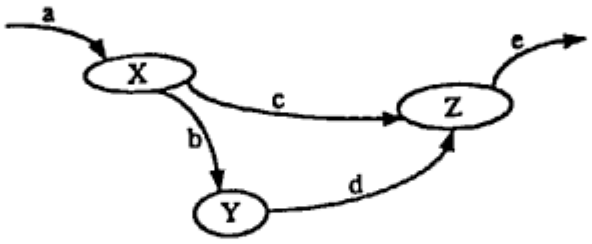

Figure 2: An example of a data flow diagram

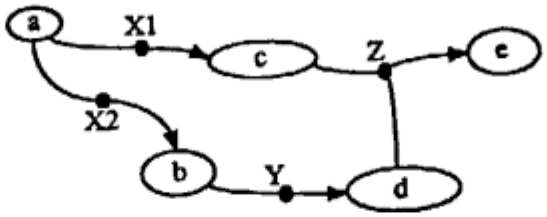

Figure 3: An example of a signature graph

<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 4: Object and bubble syntax

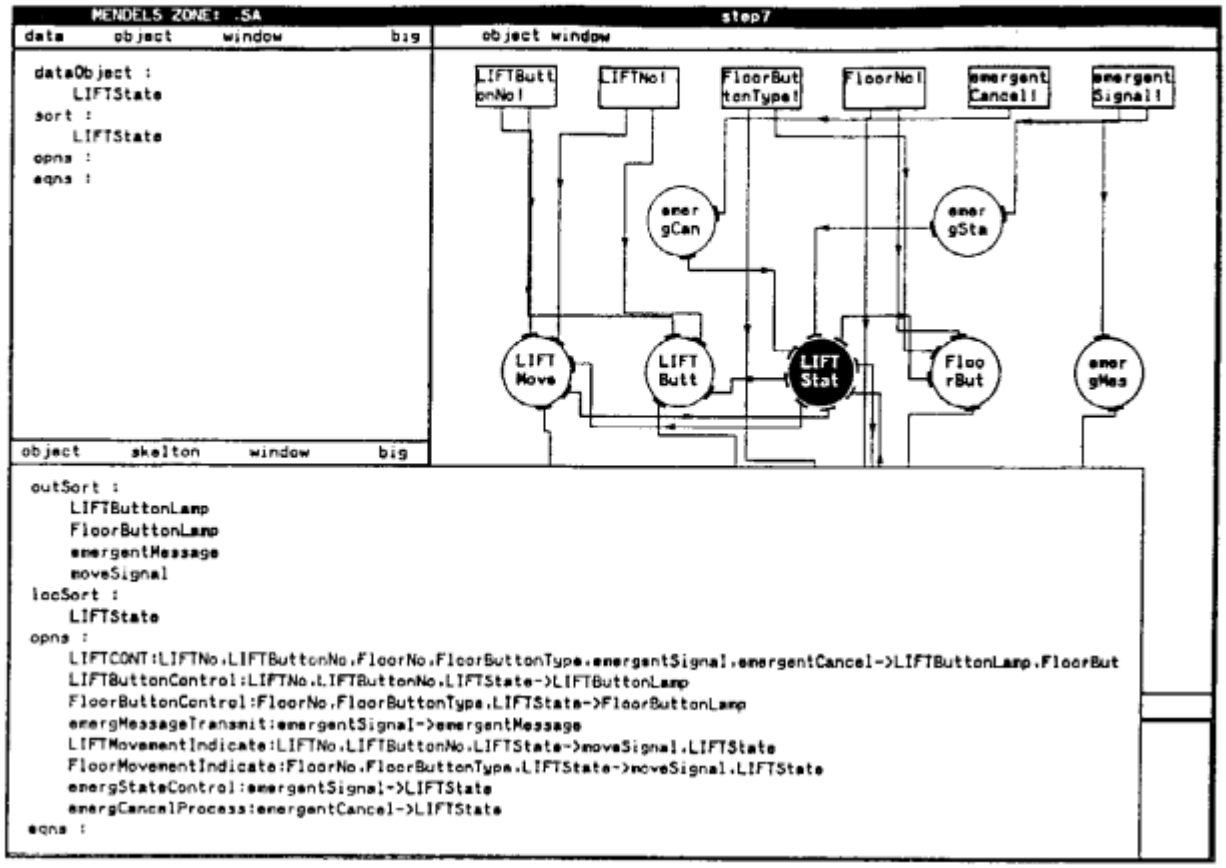

Figure 5: An algebraic specification and tranlated into DFDs in MENDELS ZONE

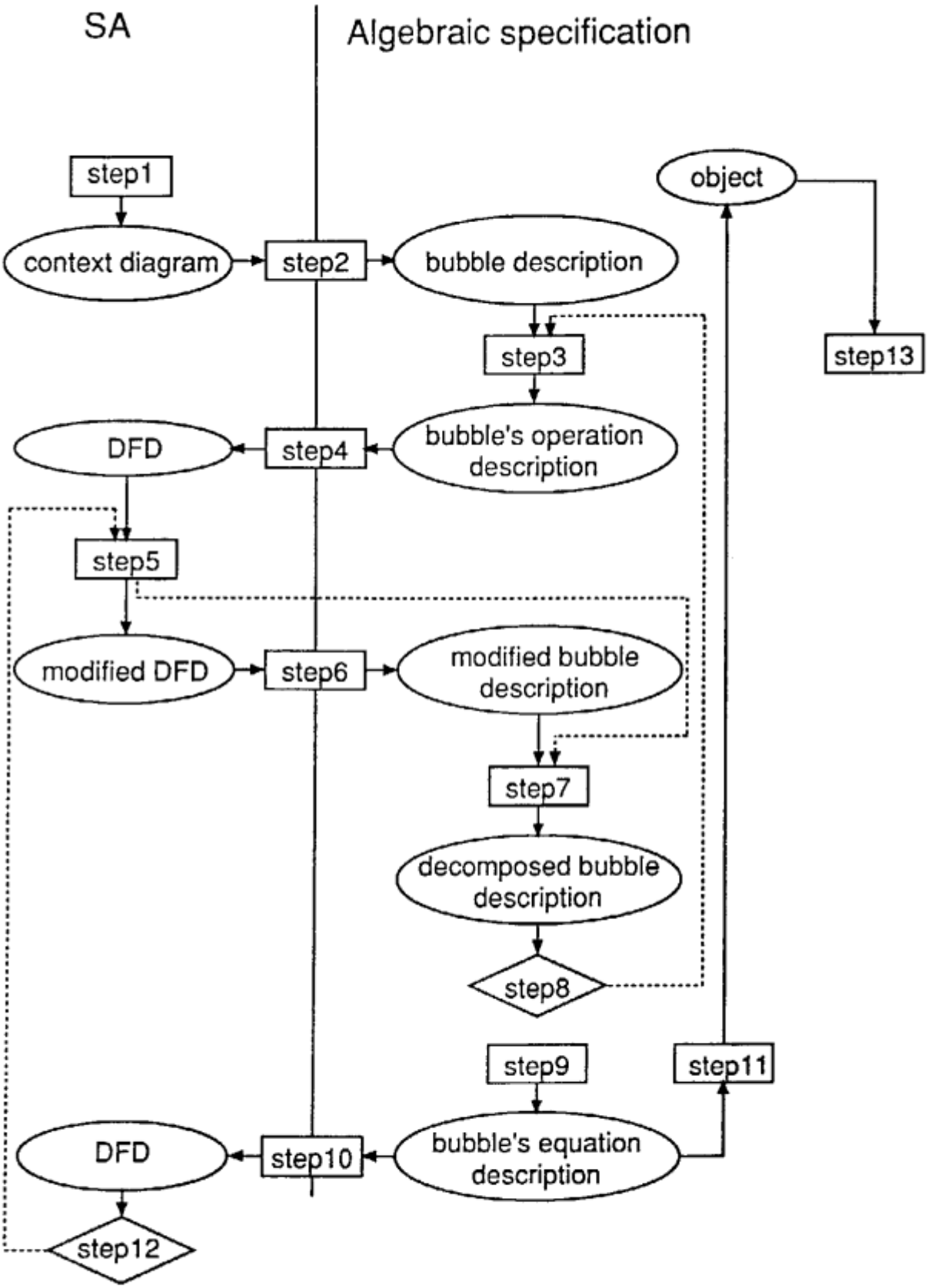

Figure 6: A detailed specification process
(Note: A directed straight line indicates data flow,
a directed dashed line indicates conditional control flow.)

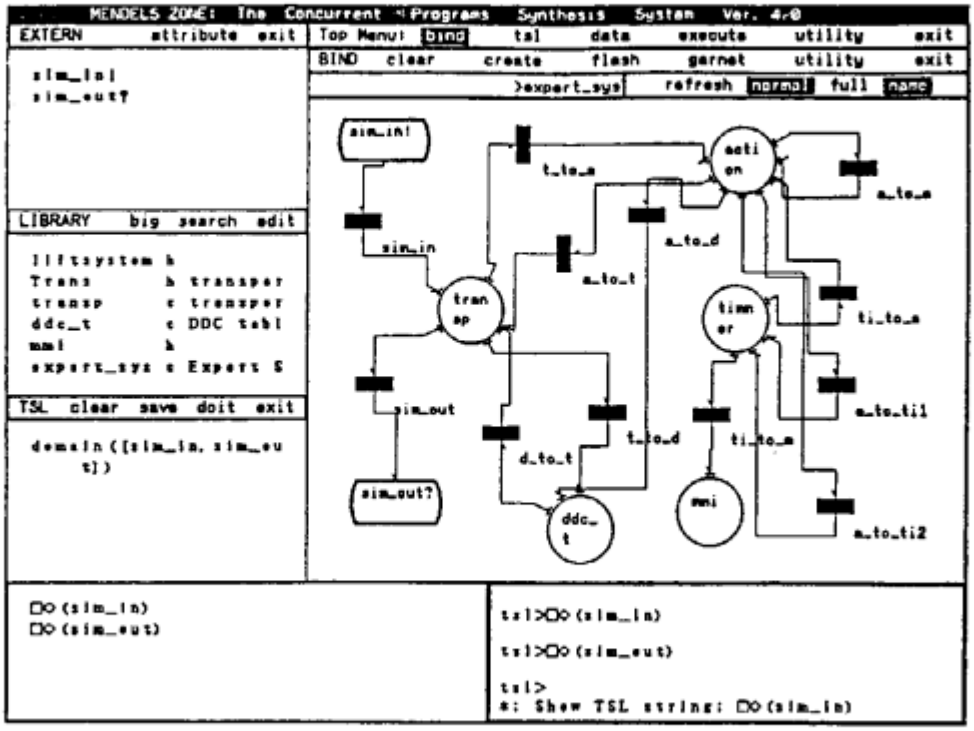

Figure 7: MENDEL components represented with Petri nets in MENDELS ZONE