JAIST Repository
https://dspace.jaist.ac.jp/

Title	A scenario-based object-oriented modeling method with algebraic specification techniques
Author(s)	Nakajima, Shin; Futatsugi, Kokichi
Citation	Research report (School of Information Science, Japan Advanced Institute of Science and Technology), IS-RR-2001-016: 1-41
Issue Date	2001-07-31
Type	Technical Report
Text version	publisher
URL	http://hdl.handle.net/10119/8389
Rights	
Description	リサーチレポート（北陸先端科学技術大学院大学情報科学研究科）

JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

A Scenario-based Object-Oriented Modeling Method with Algebraic Specification Techniques

Shin NAKAJIMA
NEC Corporation
and
Kokichi FUTATSUGI
Japan Advanced Institute of Science and Technology

July.31.2001
IS-RR-2001-016

A Scenario-based Object-Oriented Modeling Method with Algebraic Specification Techniques

Shin NAKAJIMA*
NEC Corporation
and
Kokichi FUTATSUGI†
Japan Advanced Institute of Science and Technology

Abstract

A new scenario-based object-oriented modeling method is proposed. It incorporates a semi-formal intermediate design notation GILO (Generic Interaction Language for Objects) that provides essential ingredients to represent scenario. GILO, being influenced by algebraic specification techniques, has well-defined syntax and semantics, and the modeling method is more rational than existing scenario-based modeling methods. Further, GILO descriptions can be translated into modules written in CafeOBJ, an algebraic logic language. CafeOBJ has clear semantics based on hidden order-sorted rewriting logic and descriptions can be executable. In order to show how the proposed modeling method is applied, a concrete case study on a standard design benchmark problem (the SAKE Warehouse problem) is presented. The resultant design artifact, which confirms to scenario-based object-oriented modeling method, is executable and validated in a rapid-prototyping manner with CafeOBJ.
Keywords: method integration, rapid prototyping, algebraic logic language, object-oriented modeling, collective behavior

1 Introduction

Object-oriented modeling method has been widely accepted in the industry as an established technology for improving the quality of software from early

*nakajima@ccm.cl.nec.co.jp
†kokichi@jaist.ac.jp

stages of the development. Since Booch's seminal paper [4], many methods have been proposed on object-oriented design [23]. It is followed by efforts to produce a unified method [5] and then UML [27][39], that can be used as a standard modeling notation. The variety of methods fall into two broad categories [42][48]: data-driven and responsibility-driven modeling methods. The latter is sometimes called scenario-based one [7], and examples include Jacobson's OOSE [24], CRC [2], RDD [49], and Fusion [9]. Scenario is a representation of partial functional requirements of the system in an abstract manner, and is a modeling tool to help validate the functional behavior of the design artifact. Since the concept of scenario becomes important in object-oriented modeling methods, UML also provides such a concept as one of the core modeling notations.

Scenario in the current modeling methods, however, is somewhat intuitive and is not so rigorous that the quality of design artifacts solely relies on design review by experienced human engineers. By elaborating the concept of scenario to be rigorous enough to be amenable to mechanical checking, one can expect to greatly reduce the cost of design review. It needs a thorough understanding of the scenario concept and to find a way to "method integration," in which one employs the scenario-based modeling method to analyze the problem at hand, and then uses a formal specification language to have rigorous design artifacts.

We propose a new scenario-based object-oriented modeling method with GILO (Generic Interaction Language for Objects) [35]. GILO is a semi-formal intermediate design notation that provides essential ingredients to represent scenario. It is an outcome of a critical analysis of existing scenario-based methods and authors' experience in development of several object-oriented systems [3][46][47]. Further, we employ the algebraic specification technique to be the formal basis of GILO, which realizes the method integration. Our method is dependent on concurrent rewriting logic that is suitable for modeling state changes in the algebraic logic tradition [31]. Particularly, we use CafeOBJ [10][11][15], a new algebraic logic language of the OBJ family [14][18]. CafeOBJ has clear semantics based on hidden order-sorted rewriting logic that subsumes order-sorted equational logic and a subset of concurrent rewriting logic. CafeOBJ has executable semantics, and thus the language can serve as the basis for a rapid prototyping tool in the early stages of software development.

The present paper is organized as follows: the second section presents scenario-based object-oriented modeling method, the third section proposes a new modeling method with an emphasis on the intermediate design no-

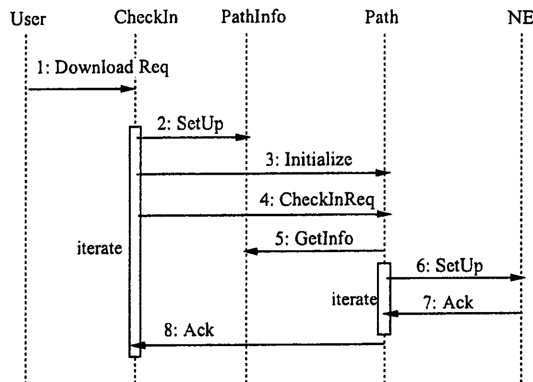


Figure 1: Event Trace Diagram

tation GILO, the fourth section illustrates a way of method integration with CafeOBJ and shows a case in which the proposed modeling method is used for a problem of medium size, the fifth section compares our work with related ones, the sixth section summarizes discussions on the proposed approach and then concludes the paper. The appendix includes a brief introduction of CafeOBJ.

2 Object-Oriented Modeling Methods

Object-oriented modeling method generally has two roles; (1) it provides basic notations to represent various aspects of *objects*, and (2) it provides further notations and guidelines to help derive object definitions. Different modeling methods assume different guidelines but fall into two broad categories [42][48]: data-driven modeling and responsibility-driven modeling methods. The latter is sometimes called scenario-based modeling method [7]. In data-driven modeling method such as OMT [41], structural aspects of objects and relationships between objects are the main concern at early stages of development; functional aspects of objects are left until later. Design validation is therefore concentrated with structural aspects of object diagrams, such as inheritance relationships between classes or multiplicity constraints on object attributes.

In scenario-based modeling method, such as that of OOSE [24], CRC [2] or RDD [49], an object is an entity that is responsible for a particular part of functionality that the whole system provides. Identifying objects requires analysis of functional behavior of the system and involves separating out the functional coupling between objects. Thus, the method puts emphasis on validating the functional behavior of more than one object in early stages of the development. It implies that rapid-prototyping of a specification comprising many objects is inevitable for obtaining high quality design artifact. The approach has been successful in the user interface design and now becomes popular in the world of object-oriented modeling [7]. As summarized in a famous statement [2][22];

> No object is an island,

collective behavior of objects [34] is essential in the design of any significant object-oriented software system. This observation also supports the importance of scenario-based object-oriented modeling methods.

A scenario is actually a prototypical history of an execution and consists of a sequence of messages exchanged among participant objects. Figure 1 is an example representation of scenario used in object-oriented modeling. The horizontal arrow refers to a message sending event, and the numbers in the diagram specify the order of events where an event is a message to some object. The diagram is called a message sequence chart (MSC) or an event trace diagram.

While the usage and abstraction level of the notations differ in each method [24][49], the essence of a scenario is to have two notions:

1. the information about participant objects,

2. the information about the sequence of their interactions.

The scenario concept in general, however, has two drawbacks to be used as a basic modeling tool. First scenario is just an execution history, and is not a complete specification. One must combine a lot of related (sub)scenarios to obtain a consistent specification. Second, scenario in the current modeling methods is intuitive. As Figure 1 shows, the representation is simple and easy to understand, but needs intuition to reason about the meaning. In a word, the reasoning is dependent on engineer's experience. Therefore, rationalized notation for the scenario-based object-oriented design specification is called for. It is desirable at the same time to be rigorous enough to be amenable to mechanical checking.

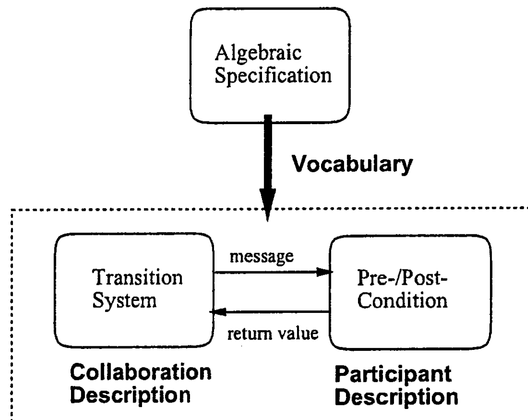


Figure 2: GILO Model

3 Scenario-based Modeling with GILO

3.1 GILO : Intermediate Design Notation

We propose a scenario-based modeling method that uses an intermediate design notation GILO (Generic Interaction Language for Objects) [34][35]. As described in Section 2, a scenario needs to make two aspects of the specification components explicit: the behavior of participant objects and behavior of global interaction between them. Further, the interaction should not be an instance of execution history, but a complete specification to describe dynamic or reactive aspect of the specificand completely.

GILO employs two basic abstraction viewpoints as specification component: class for the behavior of participant object and collaboration for the interaction specification. Since the two aspects are quite divergent computational entities, it is difficult to encode both characteristics in one specification model. We use a multiparadigm specification approach [53] and define relationships between the components precisely. Figure 2 illustrates the basic idea of GILO, which provides three kinds of specification components: (1) Collaboration, (2) Class, and (3) Common Vocabulary.

3.1.1 Collaboration

Since collaboration is responsible for the way participant objects exchange messages, dynamic reactiveness is essential. For collaboration, we have adopted a transition system, which can be considered an abstract computational model of the Message Sequence Chart (MSC) shown, for example, in Figure 1. The model also allows conditional branching and iteration, both of which a MSC cannot express, and thus is more general. Conversely a MSC is a record of an execution history generated by the transition system. Since a message is sent to some of the participant objects in the course of state transition, the transition system can be considered to explicitly describe how the global interaction of the objects proceeds. Collaboration also includes declarations of participant objects.

A few words on the relationship between Collaboration of GILO and UML Collaboration [39] are needed here. UML Collaboration uses diagram notations to illustrate structural relationships among the participant objects in order to put emphasis on the role of each participant in the overall structure. UML Collaboration can also be accompanied with dynamic aspects describing a sequence of messages. The description is an execution trace that is essentially identical to a MSC. In GILO, collaboration is a transition system that manages message-flows among the participant objects in a centralized viewpoint. That collaboration is a reactive entity is the most important point.

3.1.2 Class

In regard to the definition of participant objects, methods of the classes for the objects provide behavioral aspects. Since an object has internal states and method invocation often results in changes in those states, it is natural to model the method behavior in terms of state-oriented specification or a combination of the pre- and post-conditions. In addition, a method may return some value as its result. We allow to use the introduction of return variables for this purpose. That is, the method can update a predefined variable *ret* to return some value to its caller.

Although we call "class", it is more like "role" in the oocam method [40]. It is because our primary concern here is to show how the entity (either class or role) behaves in relation to other participant entities, and not to show how the entities are related from a viewpoint of their structural aspects such as inheritance or link relationship. UML Collaboration [39] also uses a notion

of role to be a basis for describing global flow of controls.

3.1.3 Common Vocabulary

In describing the definition of collaboration or class, we often use auxiliary symbols to which we assign some particular meanings. The third component provides a set of common vocabularies that are employed to interpret each symbol in collaboration or class. We have adopted an abstract datatype technique for this component of GILO specification.

In object-oriented modeling method such as Catalysis, class and common vocabulary is not distinct. Both are user-defined entities introduced as types. Type in Catalysis is the same as class in UML and is usually introduced in class diagrams. We, however, see the distinction is important because class is an entity participating in collaborations while common vocabulary is a set of entities that contribute to provide class with application semantics.

This idea is inspired by the Larch family of specification languages [20], in which the shared language component defines common vocabulary and the interface language uses the vocabulary to describe the behavioral specifications of procedures or functions. The common vocabulary of GILO corresponds to the shared language of Larch.

3.2 GILO by Examples

Next presents the three kinds of GILO components by using concrete examples.

3.2.1 Collaboration Description

Collaboration has two components: (1) declaration of participant objects, and (2) state-machine to represent dynamic behavior. Figure 3 shows an example, Collaboration SimpleExample. This has three participant objects; *user* is an object of Class User, *dir* is an object of Class Directory, and *nodes* is an object of Class NodeList that is a list of Node object. A *dir* object maintains the directory information that establishes a mapping between the access key and the nodes. The example collaboration describes a flow that (1) obtaining *nodes* object by looking up the *dir*, (2) sending a turn-on message to each node object in the *nodes*, which forms an iterative loop, and (3) sending a notification to *user* when the process completes.

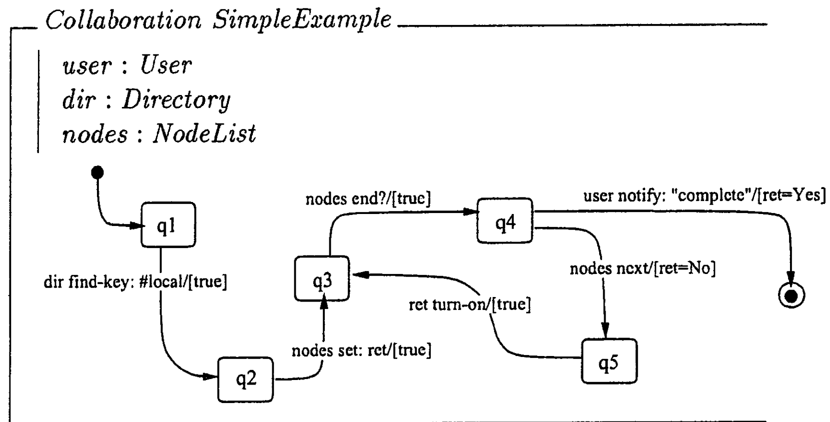


Figure 3: Collaboration Example

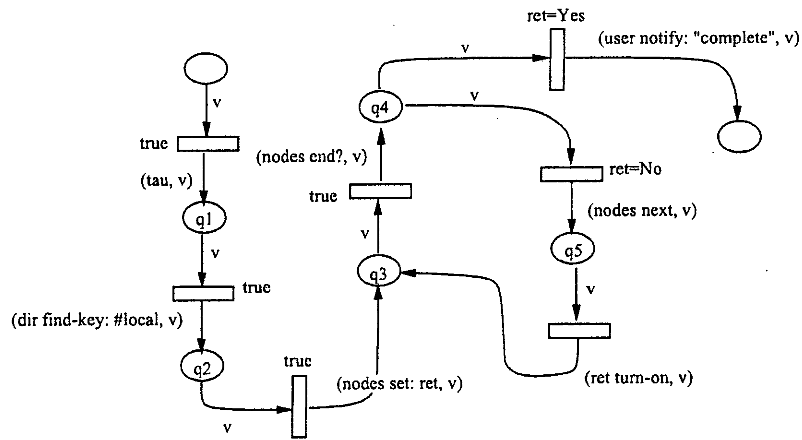


Figure 4: Equivalent CPN Description

A diagram-based labeled transition system is used to represent dynamic behavior of collaboration. Figure 3 is an example, which has 7 states that correspond to a default initial state, five states from q1 to q5, and a final state denoted by $\odot$. And each transition arc has a label denoting a message to one of participant objects and a condition to fire the transition. For example, the expression attached to the arc from q4 to q5 specifies that a *next* message is sent to the object denoted by the variable *nodes* when a return value of the previous message (*ret*) equals to *No*:

nodes next/[ret=No].

In order to express an operational interpretation of the transition system, Coloured PetriNet (CPN) [26] is used. Figure 4 is a CPN representation equivalent to the transition system in Figure 3. *Colour* of a token *v* carries an environment to manage status of participant objects.

3.2.2 Class Description

Class also has two components: (1) attributes constituting internal structure of objects, and (2) definitions of method body. Method is invoked when an object receives appropriate messages.

Figure 5 shows an example, Class NodeList, whose object is one of the participant of Collaboration SimpleExample. Class uses a box notation as a concrete syntax, which is similar to the Z notation [43]. A small box named State[1] has a set of attribute declarations. In the example, a NodeList object has an attribute *ptr* of sort *List of OId*. The other three are method definitions. For example, the method *next* has a precondition (**assumes:**) of $\neg$ *(null ptr)* and a postcondition (**ensures:**) specifying an update in the attribute *ptr* with a return value (*ret*). And $\Delta(ptr)$ shows that this method updates only the *ptr* attribute in the course of the method execution. Similarly, the *end?* method shows by using $\Xi(ptr)$ that it does not change value of *ptr*.

3.2.3 Common Vocabulary

Collaboration SimpleExample above uses two constants *yes* and *no*. Class NodeList assumes List-related functions such as *head* or *null*. These auxiliary symbols should be defined somewhere.

[1] A reserved word.

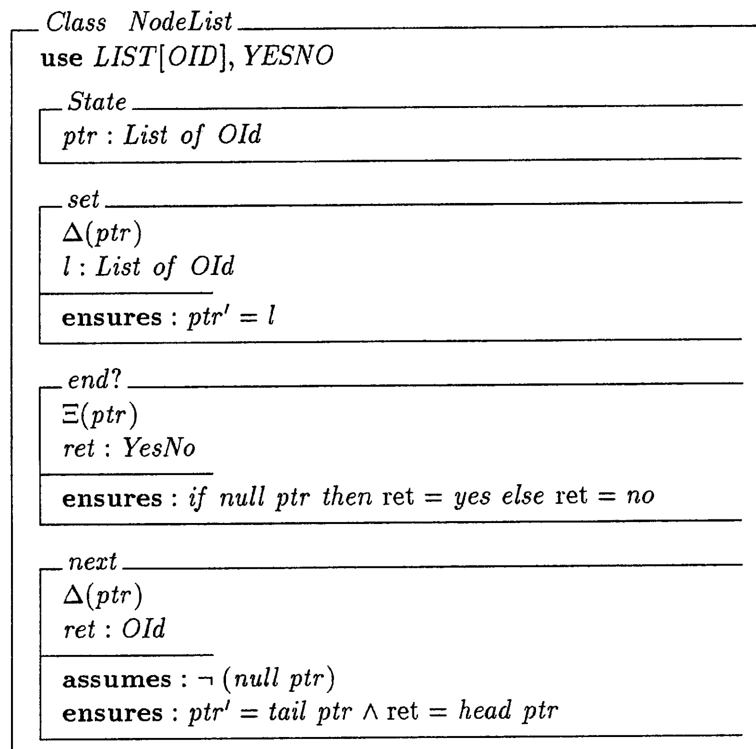


Figure 5: Class Example

Module YESNO

[*YesNo*]

$YesNo ::= yes \mid no$

Module LIST[*X* :: *TRIV*]

protecting *NAT*

[*Elem* < *NeList* < *List*]

$List ::= nil \mid _\ _ : List \times List$
$NeList ::= _\ _ : NeList \times List$
$head\ _ : NeList \rightarrow Elem$
$tail\ _ : NeList \rightarrow List$
$null\ _ : List \rightarrow Bool$
$\mid _ \mid : List \rightarrow Nat$

$E : Elem$
$L : List$

$head\ E\ L = E$
$tail\ E\ L = L$
$null\ nil = true$
$null\ E\ L = false$
$\mid nil \mid = 0$
$\mid E\ L \mid = 1 + \mid L \mid$

Figure 6: Common Vocabulary Example

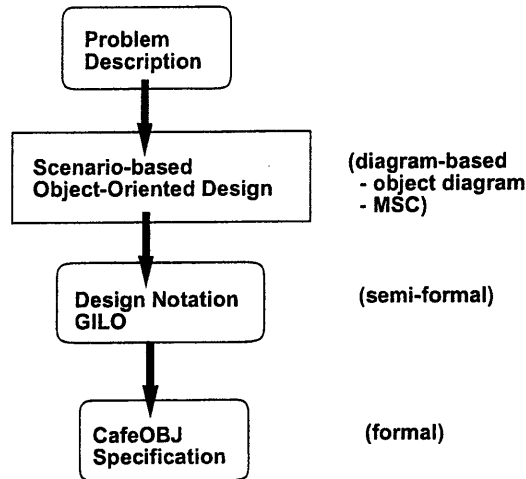


Figure 7: Overview of Development Steps

GILO provides the third component to define such common vocabulary by using order-sorted algebra. Figure 6 shows definitions for *YESNO* and *LIST*. As seen from the examples, common vocabulary is introduced as an abstract datatype definition.

4 Method Integration with CafeOBJ

This section presents our approach to a method integration of GILO and CafeOBJ. We assume a development process as one illustrated in Figure 7 for the method integration. The approach is basically to provide a set of translation rules from GILO description to CafeOBJ counterpart. In a word, the translation is simply a matter of encoding of the operational semantics of GILO in CafeOBJ. The resultant CafeOBJ description consists of a set of modules which faithfully reflect the analyzed structure of the problem. Descriptions in GILO are supposed to reflect the result of problem analysis. The appendix describes a brief introduction of CafeOBJ from specifier's viewpoint for readers not familiar with the language.

4.1 Translation to CafeOBJ

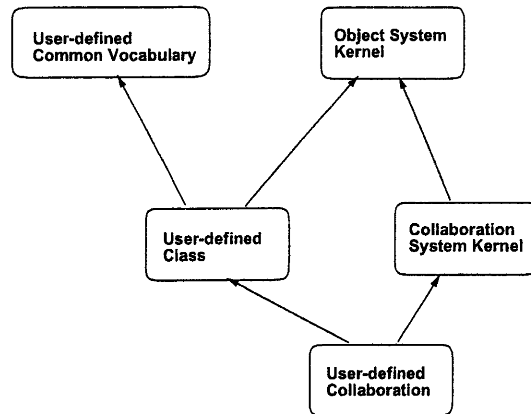


Figure 8: Module Relationship Overview

The section discusses how to derive CafeOBJ descriptions from GILO representations. Figure 8 illustrates several groups of CafeOBJ modules. They constitute *runtime* modules to represent the CafeOBJ modules derived from GILO [38].

Object System Kernel is a set of CafeOBJ modules to encode the object model. It is basically an encoding of the Maude concurrent object model, and includes the definitions of concurrent object, message, and configuration that manages the former two components. *Collaboration System Kernel* provides a machinery to encode collaboration. The idea is to define a special class of concurrent object for representing CPN. The implementation makes use of *Object System Kernel*. *User-defined* modules are categorized into *Common Vocabulary*, *Class*, and *Collaboration*. *User-defined Class* uses *Object System Kernel*, and *User-defined Collaboration* is based on *Collaboration System Kernel*. Each of the user-defined module has a direct counterpart in the GILO description.

4.1.1 Collaboration System

Collaboration System Kernel provides a basic machinery for executable user-defined collaboration. It makes use of the object system and defines a concurrent object to represent a CPN interpretation of the labeled transition system used in the collaboration.

In order to simulate *markings* in CPN, module MARKING is introduced to represent execution snapshots of a state-machine.

```
mod! MARKING {
  extending (ATTR-VALUE)
  [ State,  Marking ]
  [ State < Marking < AttrValue ]

  signature {
    op empty : -> Marking
    op (_,_) : Marking Marking -> Marking {assoc comm id: empty}
  }
}
```

The CafeOBJ representation of collaboration is an object belonging to Class Machine, that is implemented by the module MACHINE. A Machine object has two attributes; marking refers to the snapshot mentioned above, and participants denote a list of object identifiers (OIds), each one being a participant of the collaboration. Further, a Machine object can respond to two messages, fire and on, to proceed state transitions. Every CafeOBJ module for user-defined collaboration imports the module MACHINE to become powered for execution.

```
mod! MACHINE {
  extending  (ROOT)
  protecting (MACHINE-MESSAGE)

  [ MachineTerm,  CIdMachine ]
  [ MachineTerm < ObjectTerm,  CIdMachine  < CId ]

  signature {
    op <(_:_)|_> : OId CIdMachine Attributes -> MachineTerm
    op Machine : -> CIdMachine
    op make-machine : OId Marking OIds -> MachineTerm
  }

  axioms {
    var O : OId      var M : Marking     var L : OIds

  eq make-machine(O,M,L)
  =  <(O : Machine)| (marking = M), (participants = L)> .
  }
}
```

Last, following two auxiliary modules are necessary to complete the definition.

```
mod! MACHINE-ATTR-VALUE {
  extending (AID)
  extending (BASIC-VALUE)
  using (COLLECTION[OID] * { sort Collection -> OIds })
  [ OIds < AttrValue ]
  signature {
    op marking : -> AId
    op participants : -> AId
  }
}

mod! MACHINE-MESSAGE {
  extending  (ROOT)
  protecting (MACHINE-ATTR-VALUE)
  protecting (MARKING)
  signature {
    op on : OId -> Message
    op fire : OId -> Message
  }
}
```

4.1.2 Translation Example

Below shows the CafeOBJ modules for the example GILO descriptions (Figures 3, 5 and 6). The modules are obtained by manual translation.

A CafeOBJ module `SIMPLE-EXAMPLE` is a translation of Collaboration SimpleExample. Apart from the `ROOT`, it imports two modules. `SIMPLE-EXAMPLE-STATES` provides constant definitions for the necessary states (`q0`-`q5`, and `qf`). `SIMPLE-EXAMPLE-PARTICIPANTS` contains all the definitions of objects that are participant of the collaboration. The rewriting rules together simulate the transitions specified in the collaboration. For example, the third rule in `SIMPLE-EXAMPLE` has two messages `on(O)` and `return(O,V)` in order to express that the rule is fired only after the message `return(O,V)` is generated. Here the message `return(O,V)` is a result of a previous invocation of a participant object method. In the case of the third rule, the message `return(O,V)` is generated as a result of `find-key(dir,'local,O)` appeared in the RHS of the second rule. It is a message sent to `dir` object requesting a `NodeList` object that is indexed with `'local`.

```
mod! SIMPLE-EXAMPLE {
  extending (MACHINE)
  protecting (SIMPLE-EXAMPLE-STATES)
  protecting (SIMPLE-EXAMPLE-PARTICIPANTS)
  axioms {
   var O : OId  var M : Marking  var P : OIds  var REST : Attributes

   trans fire(O)
      <(O : Machine)|(marking = (q0, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q1, M)), (participants = P), (REST)>
      on(O)  .

   trans on(O)
      <(O : Machine)|(marking = (q1, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q2, M)), (participants = P), (REST)>
      find-key(dir, 'local,O) on(O) .

   trans on(O) return(O,V)
      <(O : Machine)|(marking = (q2, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q3, M)), (participants = P), (REST)>
      set(nodes, V,O) on(O) .

   trans on(O) void(O)
      <(O : Machine)|(marking = (q3, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q4, M)), (participants = P), (REST)>
      end?(nodes, O) on(O) .

   ctrans on(O) return(O,V)
      <(O : Machine)|(marking = (q4, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (qf, M)), (participants = P), (REST)>
      notify(user, 'complete, O) on(O) if V == Yes .

   ctrans on(O) return(O,V)
      <(O : Machine)|(marking = (q4, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q5, M)), (participants = P), (REST)>
      next(nodes, O) on(O) if V == No .

   trans on(O) return(O,V)
      <(O : Machine)|(marking = (q5, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q3, M)), (participants = P), (REST)>
      turn-on(V, O) on(O) .
  }
}
```

The next CafeOBJ module is a translation of Class NodeList (Figure 5). Rewriting rules (**trans**, **ctrans**) corresponds to the three methods.

```
mod! CLASS-NODE-LIST {
  extending (ROOT)
  protecting (NODE-LIST-MSG)
  [ NodeListTerm, CIdNodeList ]
  [ NodeListTerm < ObjectTerm, CIdNodeList < CId ]
  signature {
    op <(_:_)|_> : OId CIdNodeList Attributes -> NodeListTerm
    op NodeList  : -> CIdNodeList
  }
  axioms {
    vars O R  : OId    vars L L' : List    var REST  : Attributes

    trans set(O,L,R) <(O : NodeList)|(ptr = L'), (REST)>
    => <(O : NodeList)|(ptr = L), (REST)> void(R) .

    ctrans end?(O,R) <(O : NodeList)|(ptr = L), (REST)>
    => return(R, Yes) <(O : NodeList)|(ptr = L), (REST)> if null(L) .

    ctrans end?(O,R) <(O : NodeList)|(ptr = L), (REST)>
    => return(R, No) <(O : NodeList)|(ptr = L), (REST)> if not null(L) .

    ctrans next(O,R) <(O : NodeList)|(ptr = L), (REST)>
    => return(R, head(L)) <(O : NodeList)|(ptr = tail(L)), (REST)>
    if not null(L) .
  }
}
```

Common vocabulary of GILO is just a syntax-suger of CafeOBJ module. A translation to CafeOBJ is straightforward.

4.1.3 Multithreaded Collaboration

Although it is not covered in the previous sections, multithreaded collaboration is easily encoded in CafeOBJ. As the primitives of multithreaded collaboration, we consider fork and join.

A transition in the whole system is encoded in a rewriting rule that changes the marking data, where the marking is a multiset of state markers and each marker represents an execution snapshot of one thread of execution (Section 4.1.1). Figure 9 shows a CPN version and the following CafeOBJ descriptions simulate the same transition. For the fork, marking on the

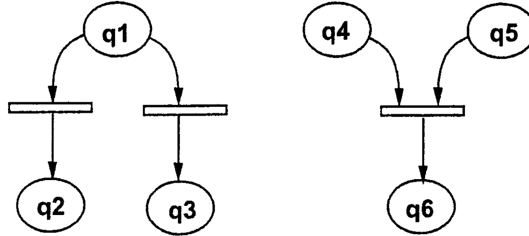


Figure 9: Fork and Join

RHS contains all the states that the execution forks to (q2 and q3). For the join, all the states that should be synchronized are specified explicitly in the marking on the LHS.

```
var REST : Marking .
trans on(O) <O:Machine | marking = (q1,REST)>
=> <O:Machine | marking = (q2,q3,REST)> .

ctrans on(O) <O:Machine | marking = (q4,q5,REST) >
=> <O:Machine | marking = (q6,REST) > .
```

Simple diagram representation sush as one in Figure 3 is not available, however, the above rules illustrate that multithreaded collaboration is easily encoded in CafeOBJ.

4.2 A Case Study

This section presents a modeling case in which CafeOBJ specifications are derived by following the proposed development steps (Figure 7). Key points of the steps are (1) to construct scenarios by identifying participant objects and their interactions, and (2) to construct GILO descriptions. The case illustrates the role of three GILO components in the overall specification.

4.2.1 *SAKE* Warehouse Problem

The *SAKE* Warehouse problem is a standard common problem [52]. Since its first appearance in the literature, it has been used as a standard benchmark problem for comparing various design methodologies in the software

engineering community in Japan [45][52]. The problem is compact but has essential features commonly found in a lot of business application software. The following is an English translation of the *SAKE* Warehouse Problem [45].

> A warehouse of X Sake Retailing Company accepts several containers everyday. Each container contains sake bottles, possibly of multiple brands. The number of brands that can be mixed in one container is up to ten. The total number of brands to be treated is about 200.
>
> A warehouse keeper stores each container carried into the warehouse without any rearrangement and sends a container contents notice to a clerk. He also ships out sake bottles by the shipment direction forwarded from the clerk. Stored bottles are never repacked into another container, nor kept in another place. An emptied container is immediately carried out of the warehouse.
>
> container contents notice:
> container number (5 digits)
> carried-in time (hour/day, month/year)
> brand, quantity (repeat)
>
> The clerk receives dozens of shipment orders per day and sends a shipment direction to the keeper for each order. An order comes by an order form or by telephone and each order must designate just one brand. If the brand is out of stock or in short for the ordered quantity, the clerk will tell it to the customer and adds the order to the waiting list. And when the designated brand is supplied to meet the order, the clerk will issue a shipment direction.
>
> In a shipment direction, containers that will become empty are notified.
>
> Develop a system that supports the work of the clerk (notifying out of stock status, issuing shipment direction forms and listing the outstanding orders).
>
> shipment direction form:

order number
customer number
container number
brand, quantity
empty mark

waiting list:

customer name
brand, quantity

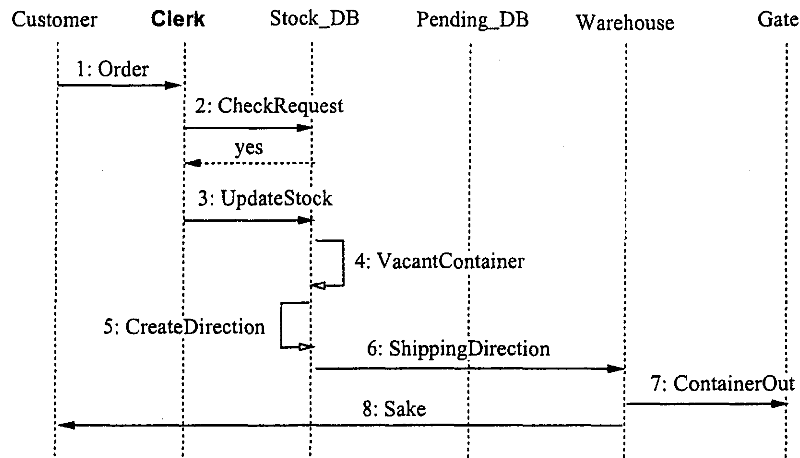
- No loss of sake will occur either during the transportation or during the storage.
- As some part of the problem description may not be realistic, sophisticated functions such as exception handling can be minimal.
- Ambiguities may be resolved by appropriate interpretation.

4.2.2 Identifying Scenarios

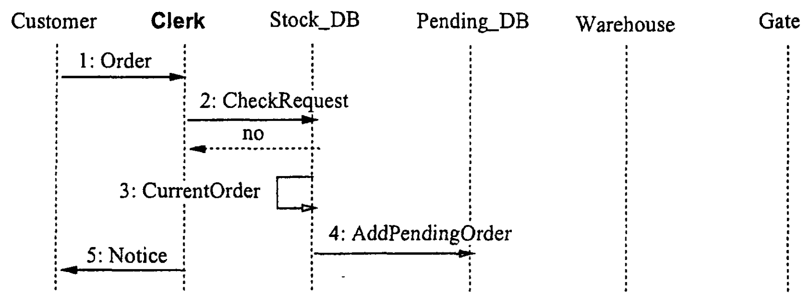
The modeling step starts with the scenario construction. Analysis of the problem description results in identification of two main scenarios, *Order Arrival from Customer* and *Container Arrival*. Scenario in general has more than one subscenarios: one for a main flow and others for handling exceptional cases. For example, the main flow of the *Order Arrival from Customer* scenario is to deliver requested bottles of sake, while the order is added in a waiting list when enough stock is not available.

Figure 10 shows two subscenarios for the *Order Arrival from Customer* by using Message Sequence Charts (MSC); (a) the retailing company has enough stock to fulfill the order, and (b) the order is added to the pending order database because the stock is insufficient. In the normal case (a), the arriving order initiates the subscenario (step a1). This step is followed by a check of whether there is enough stock to fulfill the order (step a2). If so, the stock database is updated (step a3). After empty containers are collected (step a4), a shipping direction to the warehouse keeper is created (step a5). In the case (b), on the other hand, the order is added to the pending order database (step b4) and a notification is issued that the order is in the waiting list (step b5).

In the process of constructing the above MSCs, the responsibility or abstract functionality of each participant object is identified. Of the six



(a) In Stock



(b) Out of Order

Figure 10: Order Arrival from Customer

participants, the two key objects in the subscenarios are the Stock Database and the Pending Order Database. Other objects may be considered auxiliary and constitute the environment in which the whole scenarios are described completely. Constructing another scenario, Container Arrival, also helps elaborate the definitions of two database objects.

4.2.3 GILO Descriptions

The second step involves construction of the GILO specification. Since GILO has three components, division of labor between them is a key aspect of the specification construction. First, since collaboration is responsible for the global flow of messages between participant objects, it is constructed by combining MSCs of subscenarios that together constitute one scenario. Second, since an object has states and is modeled in terms of state changes, writing GILO class involves to find attribute data that a participant object maintains internally and methods that operate on the data. Third, common vocabulary modules are introduced. The modules provide interpretation of symbols used in object methods and are purely functional (no side-effect).

First, Figure 11 is a GILO description of Collaboration OrderArrival. It is constructed by combining the two MSCs in Figure 10 with an introduction of appropriate conditional branching at the state q2. The transition sequence from q2 to q5 corresponds to the normal case shown in Figure 10 (a) while the sequence from q2 to q7 corresponds to Figure 10 (b).

In formalizing GILO model of collaboration, arguments of message are also identified so that all the information necessary to define object interfaces is determined. Note that a collaboration does not explicitly specify the initiator object (an object that sends a message) but shows only the sequence of message events. Collaboration is, therefore, somewhat more abstract than a MSC that explicitly specifies the message sender as well as the receiver.

Figure 12 is a partial description of Class StockDatabase. It defines the interface specification of the class that is in accordance with the collaboration. In identifying the interface specification of Class StockDatabase, we have taken into account Collaboration ContainerArrival as well as Collaboration OrderArrival, although the current discussion presents the latter one only. Of the seven method in the definition, *addNewStock* and *selectPendingOrders* come from Collaboration ContainerArrival.

The next step is to elaborate the internal structure of the class and the functional specification of each method (Figures 13 and 14). The StockDatabase consists of several attributes that constitute the object states. The

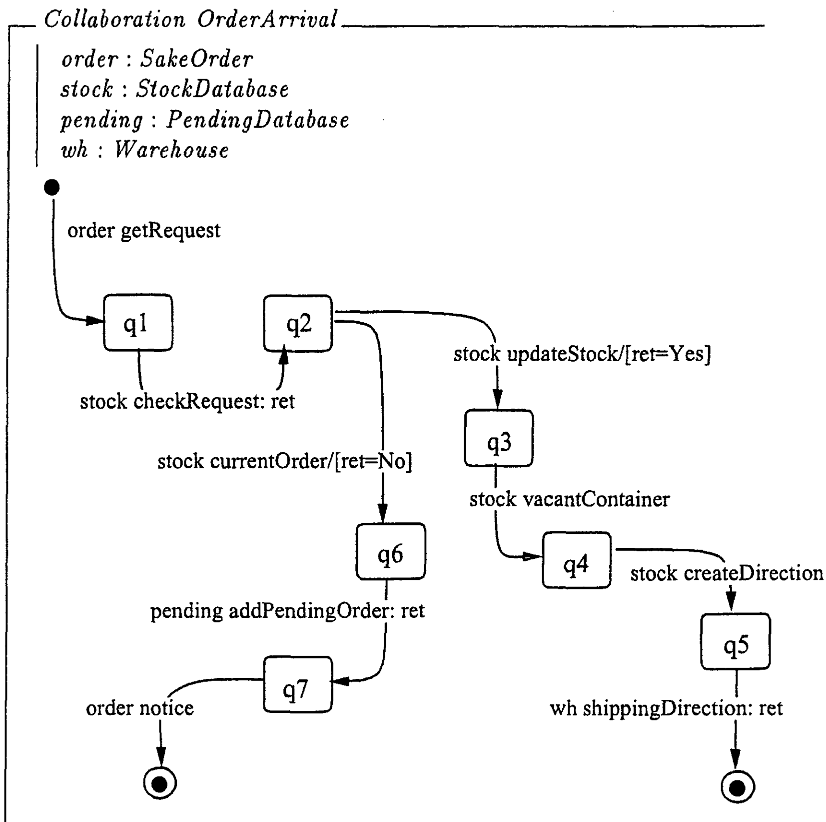


Figure 11: Collaboration OrderArrival

Class StockDatabase

$addNewStock : List\ of\ Stock \rightarrow List\ of\ Stock$
$checkRequest : Request \rightarrow YesNo$
$updateStock : void \rightarrow void$
$vacantContainer : void \rightarrow void$
$createDirection : void \rightarrow ShippingDirection$
$selectPendingOrders : List\ of\ Request \rightarrow List\ of\ Request$
$currentOrder : void \rightarrow Request$

Figure 12: Interface Specification

attribute *contents* maintains the content of the database, and *stock_number* keeps track of a value that gives a unique identification number to each stock that the database has. Three other attributes are used to store values that are processed in the course of executing the collaboration.

Class StockDatabase also declares that it uses the common vocabularies, STOCK_DB, YESNO, and NAT, to describe its own behavioral specification. The module STOCK (Figure 15) defines an abstract datatype to represent stock, and provides a constructor *stock* and other functions such as *container*.

The module STOCK_DB (Figure 16) defines functions to realize the main functionality of Class StockDatabase. The module STOCK_DB is basically a parameterized DB module that has STOCK as the actual parameter. The DB module is a basic one common to both STOCK_DB and PENDING_DB, the latter of which provides vocabulary for Class PendingDatabase. The module STOCK_DB adds further auxiliary functions to the module DB so that Class StockDatabase is defined in a compact manner.

4.2.4 CafeOBJ Descriptions

The final step of the modeling process is simply to translate the GILO descriptions into the CafeOBJ modules. One can validate the functional behavior of the design artifact, such as the scenarios shown in Figure 10, by test execution. As a result, one can enjoy rapid-prototyping of the GILO design artifact by making use of CafeOBJ.

The resultant CafeOBJ descriptions for the Sake Warehouse Problem

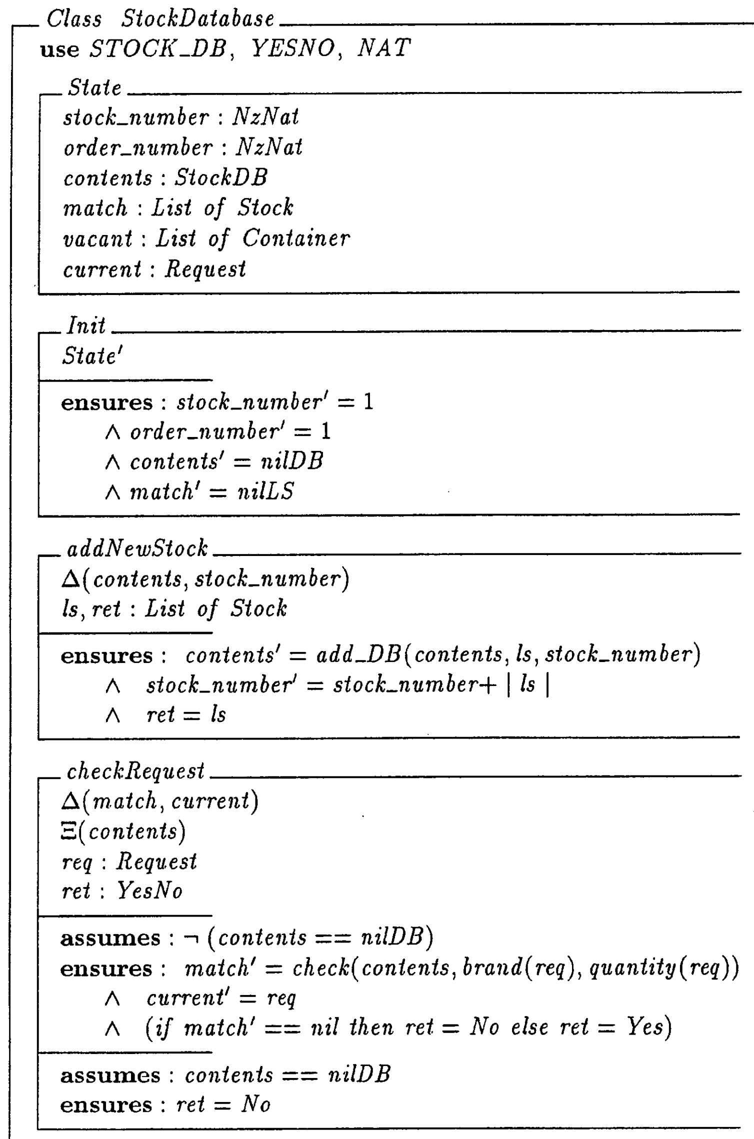


Figure 13: Class StockDatabase

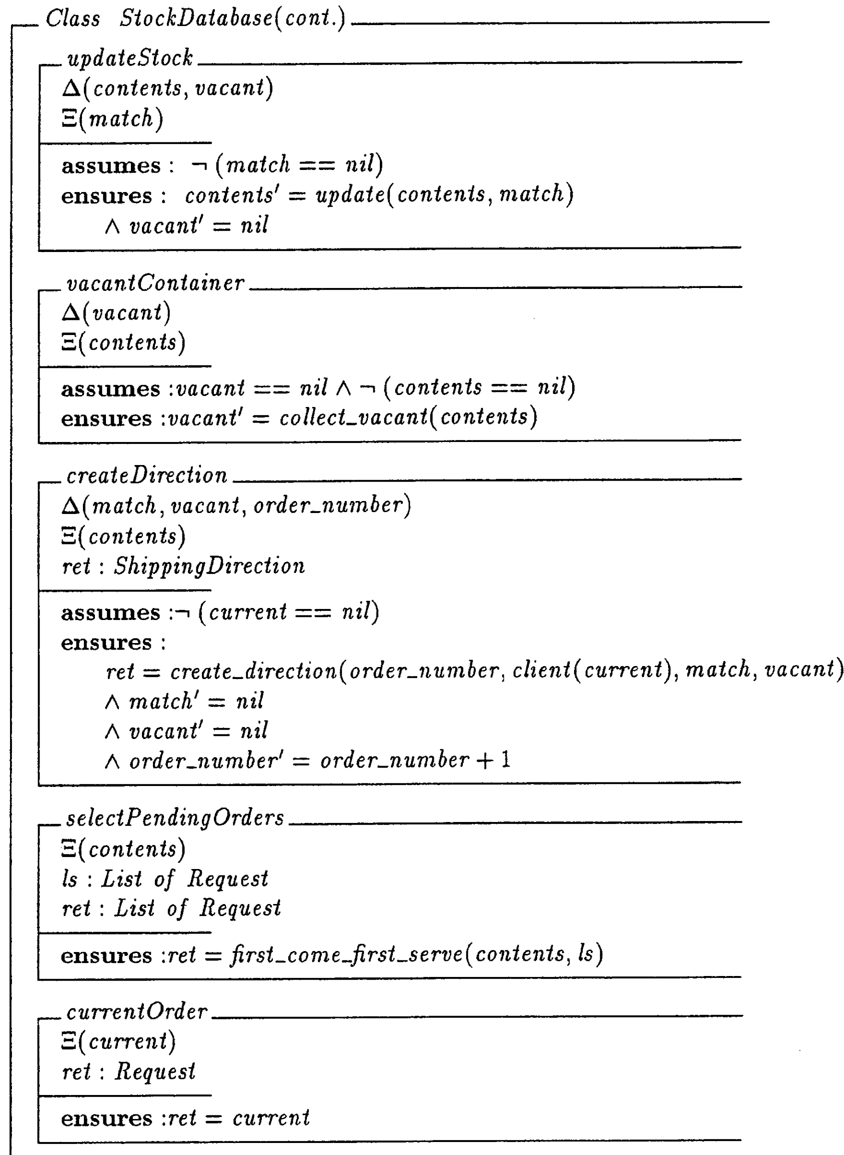


Figure 14: Class StockDatabase (cont.)

Module STOCK

protecting *SAKE_BASICS*, *NAT*

[*Stock*]

$Stock ::= stock : Container \times Brand \times Nat \times Nat$
$container\ _ : Stock \rightarrow Container$
$brand\ _ : Stock \rightarrow Brand$
$quantity\ _ : Stock \rightarrow Nat$
$id\ _ : Stock \rightarrow Nat$
$decr_stock_quantity\ _ : Stock \times Nat \rightarrow Stock$

$C : Container$
$B : Brand$
$Q\ Q'\ I : Nat$

$container(stock(C, B, Q, I)) = C$
$brand(stock(C, B, Q, I)) = B$
$quantity(stock(C, B, Q, I)) = Q$
$id(stock(C, B, Q, I)) = I$
$decr_stock_quantity(stock(C, B, Q, I), Q') = stock(C, B, Q - Q', I)$

Figure 15: Module STOCK

Table 1: Module Summary

	Category	CafeOBJ	
		total	direct
1	GILO Mechanism	13	–
2	Common Vocabulary	15	15
3	Class	15	12
4	Collaboration	3	3
5	Main	1	–
	Total	47	30

```
Module STOCK_DB
using DB[STOCK]
using LIST[STOCK]*(sort List to ListStock, op nil to nilLS)
using LIST[REQUEST]*(sort List to ListRequest, op nil to nilRQ)
using LIST[CONTAINER]*(sort List to ListContainer, op nil to nilC)

  add_DB _ : Database × ListStock × Nat → Database
  stock_to_DB _ : ListStock × Nat → Database
  check _ : Database × Brand × Nat → ListStock
  collect_brand _ : Database × Brand × ListStock → ListStock
  check_quantity _ : Database × Nat × ListStock → ListStock
  update _ : Database × ListStock → Database
  update_aux _ : Database × ListStock × Database → Database
  collect_vacant _ : Database → ListContainer
  first_come_first_serve _ : Database × ListRequest → ListRequest

  D : Database
  LS : ListStock
  I J Q : Nat
  C : Container
  B : Brand
  ----------------
  add_DB(D, LS, I) = (stock_to_DB(LS, I) D)
  stock_to_DB(nilLS, I) = nilLS
  stock_to_DB(stock(C, B, Q, J)LS, I) = (stock(C, B, Q, I) stock_to_DB(LS, I + 1))

  D : Database
  B : Brand
  X : Stock
  LS LS' : ListStock
  I Q : Nat
  ----------------
  check(D, B, Q) = check_quantity(collect_brand(D, B, nilSL), Q, nilLS)
  collect_brand(nilDB, B, LS) = LS
  collect_brand((X D), B, LS)
      = if (brand(X) == B) then collect_brand(D, B, (X LS))
              else collect_brand(D, B, LS)
  check_qauntity(nilLS, Q, LS) = if Q < 0 then nilLS else LS
  check_quantity((S LS'), I, LS) = check_stock((S LS'), I - quantity(S), LS)
          ...(omitted)...
              ...(omitted)...
```

Figure 16: Module STOCK_DB

consist of 47 modules that have some 1,100 lines of CafeOBJ code. Table 1 summarizes the figures.

The entry GILO Mechanism includes both Object System Kernel and Collaboration System Kernel that basically defines the module `MACHINE` (section 4.1). Of fifteen Common Vocabulary modules, four are general-purpose such as `YES-NO` and `DB`, while the rest eleven modules are specific to the present problem. The latter includes `STOCK` and `STOCK-DB` and can be considered as *domain-specific vocabulary*.

The specified class and collaboration for the case is four and one respectively. For describing a GILO class, three CafeOBJ modules (a `mod!` module for the class body, two `mod*` modules for the names of attributes and messages) are introduced according to the guideline in Section A.2. Thus, of the fifteen modules in the category Class, twelve ($12 = 3 \times 4$) have direct correspondence with GILO descriptions. The rest three are `mod!` modules to provide concrete representation for the attribute names, the attribute value type, and the message terms. A GILO collaboration, in turn, needs three CafeOBJ modules. It implies that the traceability of the collaboration is quite clear.

5 Related Work

This section discusses comparison with related works. Comparison is made in two areas: (1) scenario-based object-oriented modeling methods, (2) method integration.

OOSE [24] and RDD [49] are two pioneers that adopt scenario-based object-oriented modeling method. Booch and Rumbaugh's unified method [5] and UML [39] also support the scenario concept. The interaction diagram of OOSE and the event trace diagram correspond to the collaboration of GILO. They, however, can describe typical execution traces only, and are diagram-based notations having less rigorous semantics.

In order to express collaboration, Fusion [9] uses the object interaction graph, which is basically the same as the event trace diagram of OMT. In addition, Fusion promotes the use of an operation model and a life-cycle model. The former corresponds to method behavior of the GILO class and the latter to the GILO collaboration. The life-cycle model uses a regular expression whose alphabet represents a set of events. The operation model offers guidelines for representing behavioral aspects of a method or an operation in terms of the pre- and post-conditions. Unfortunately the conditions

are described informally in natural language. No rigorous relationship between the life-cycle and operation models is established.

Catalysis [12] is a new modeling method that focuses on the object interactions and formal description techniques. The key idea is to treat objects and actions equally, and thus to provide interaction between objects as a first-class modeling tool. Catalysis introduces a joint action, which is a series of related actions, as a common modeling tool for use-cases and collaborations, which is in accordance with the idea of GILO. As for expressing the pre- and post-conditions or other forms of behavior description, Catalysis uses semi-formal notations OCL (Object Constraint Language) of UML [39]. Although activities on formalizing OCL is underway, OCL of Catalysis itself is not based on rigorous semantics. Catalysis, however, has an important notion of refinement, which provides a systematic guideline to transform an abstract design artifact into concrete ones in a stepwise fashion. Unfortunately, GILO does not provide any guideline for the refinement.

The second area is on the integration of the informal object-oriented modeling methods and formal specification languages. One approach is to have object-oriented extension of existing specification languages [1][6][17][28][29][44]. Most of the works, however, has concentrated on incorporating basic object-oriented concepts such as state encapsulation, property inheritance, and polymorphism into the respective host specification language. The issue on modeling collective behavior (Collaboration) is out of scope.

Larch [20] is a two-tiered specification language, in which the algebraic specification provides common vocabulary. An interface language component uses the vocabulary to describe behavioral aspects of functions or procedures. Larch/C++ [30] is one of the Larch family languages. It is primarily intended to be used for writing the interface specifications of C++ member functions in the state-oriented style, and thus it does not provide collaboration.

Giovanni and Iachini [16] and Hall [21] use object-oriented modeling method as a guideline for finding *objects* in the analysis phase and then obtain descriptions in the Z notation. Descriptions in the Z notation is hardly mechanically analyzable. Recently, Jackson [25] proposes Alloy as a rational reformulation of the Z notation and UML class notation, and that descriptions in Alloy can be mechanically reasoned about by the model checking technique. Their primary concern is the structural aspect and does not consider scenario in which dynamic aspect is essential.

NASA has conducted several case studies on the lightweight use of formal methods in the requirement modeling [13], which includes an integration of

OMT and PVS. The OMT diagram descriptions are manually translated into specification fragments that can be fed into PVS. Then, properties that the OMT description must hold are reasoned about by using PVS. Meyer and Souquières [32] proposes a set of templates that translate OMT descriptions into specifications written in the B method. Since OMT is a data-driven modeling method, the emphasis is put on consistency checking of structural aspects of the model such as multiplicity of association links. Thanks to the B method tool, most of the checking can be done automatically. Both [13] and [32] concentrate on structural aspects and pay less attention to collective behavior of objects. Further, it requires to construct and conduct manual proof for application specific properties.

Wirsing [50] proposes a formal object-oriented design based on Jacobson's OOSE [24] and in which a Maude-based formal object model is encoded in an algebraic specification language Spectrum. The emphasis is put on the importance of the stepwise refinement with discussions on the role of proof checking in the refinement process. Since the development process uses the interaction diagram (a scenario) as a guiding tool just for obtaining object specifications, scenario diagrams do not have rigorous semantics. Later, Wirising and Knapp [51] use process algebra to give formal accounts of the dynamic aspects by extending Maude with process expression. Process expression controls how messages are sent to particular objects. Thus, the process expression corresponds to GILO collaboration. We use transition system that also has diagram representation to represent the control aspect.

As for integrating the proposed GILO method with the stepwise refinement, one may integrate Catalysis with GILO/CafeOBJ for a start. It needs an algebraic formulation of Catalysis, which involves establishing a rigorous semantic basis for UML and OCL in an algebraic manner. And then, the stepwise refinement process may adopt the techniques reported in [51].

6 Discussion and Conclusion

By a careful study on existing scenario-based object-oriented modeling methods, we came up with a semi-formal intermediate design notation GILO. We first enumerated two of essential aspects to describe scenarios: (1) the information about participant objects, and (2) the information about the sequence of their interactions. Then we elaborated the concepts to crystalize rationalized design notation that was rigorous enough to be amenable to mechanical checking. In some sense, we showed that GILO was rigorous

by giving a set of rules translating GILO descriptions into CafeOBJ counterparts. The resultant CafeOBJ description are executable and consists of a set of modules which faithfully reflect the analyzed structure of the problem. Thus, rapid prototyping at early stages of software development is achieved. One drawback is that we cannot translate CafeOBJ descriptions back to GILO. This hinders us from seamless *debugging* activities to point out deficiencies in GILO descriptions from the execution trace of CafeOBJ counterparts.

In order to effectively use formal specification languages such as CafeOBJ, one generally has to prescribe a development process and the role of the language in the overall process. Our use of CafeOBJ in this paper is a tool for rapid-prototyping in early stages of the development as shown in Figure 7, and executability is a key feature. However, one has to be very careful to obtain executable CafeOBJ modules of abstract datatype specification since it requires that the module should have initial algebra. From a viewpoint of specification writer, a rule of thumb is that one first introduces a basic data structure as a recursively defined term, and second provides utility functions to follow the recursive structure[2]. It is similar to a functional programming style as in, for example, Standard ML [33]. Further CafeOBJ has a rewriting engine based on a subset of concurrent rewriting logic, which enables one to write executable specifications. Actually, Maude aims to be a language for describing various symbolic processing systems, which have clear logical semantics [8].

We have some experience in using ML-like notation in the specification of object-oriented design [36][37], which shows that the notation can compactly describe algorithmic aspects of the design. The experience also includes that every description in the pseudo ML can be encoded in CafeOBJ with a suitable interpretation so that the descriptions are executable. It is partly because we can encode various computational entities as suitable *algebras* thanks to the property-oriented style of specification writing. We observe that CafeOBJ, algebraic specification languages in general, is an adequate tool for rapid prototyping.

One thing to note is that the GILO notation is multiparadigm. The whole GILO specification has a global state consisting of (1) the states in collaboration and (2) the states in all the participant objects. One might argue that both components are state-based and thus the multiparadigm element is minimal. Although both class and collaboration are state-based,

[2]The appendix A.1 presents examples to follow such a specification writing style.

the roles of states are different in each component. More importantly, each has its own syntax that expresses the essential aspects of the computational model in a very concise manner. Thus GILO can be thought of being multiparadigm from the viewpoint of the notational suitability.

Last, the idea of scenario-based object-oriented modeling method with GILO has been successfully adapted in the development of several distributed object-oriented software systems [3][46][47]. Thus, the effectiveness of the modeling method can be said confirmed.

A Specifier's Introduction to CafeOBJ

CafeOBJ is a new algebraic logic language of the OBJ family, and has clear semantics based on hidden order-sorted rewriting logic [10][11]. The logic subsumes equational logic, on which OBJ has its semantic basis [14][18]. By incorporating (a subset of) rewriting rules of Maude [31], CafeOBJ makes the algebraic specification language expressive enough to provide a clear model for state changes.

From the specifier's viewpoint, CafeOBJ has two kinds of axioms[3] to describe functional behavior. An equational axiom (*eq*) is based on equational logic and thus is suitable for representing static relationships and purely functional behavior. A rewriting axiom (*trans*) is based on a subset of concurrent rewriting logic and is suitable for modeling state changes.

A.1 Abstract Datatype

Here is a simple example, a CafeOBJ specification of `LIST`. The module `LIST` defines a generic abstract datatype `List`. `_ _` (juxtaposing two data of the specified sorts) is a `List` constructor. Two accessor or observer functions `hd` and `tl` are the standard ones. `|_|` returns the length of the operand list data and is a recursive function over the structure of the list.

```
mod! LIST[X :: TRIV] {
  [ NeList, List ]   [ Elt < NeList < List ]
  protecting (NAT)
  signature {
    op nil : -> List
    op __ : List List -> List {assoc id: nil}
    op __ : NeList List -> NeList
```

[3] Hidden algebra [10][11][19] is not considered.

```
    op __ : NeList NeList -> NeList
    op hd : NeList -> Elt
    op tl : NeList -> List
    op |_| : List -> Nat
  }
  axioms {
    var X : Elt     var L : List

    eq hd (X L) = X .
    eq tl (X L) = L .

    eq | nil | = 0 .
    eq | X   | = 1 .
    eq | X L | = 1 + | L | .
  }
}
```

The module N-LIST imports the module LIST and adds definitions of some utility functions such as n-hd and n-tl. The function n-hd returns the specified number (N) of elements from the head of the list, and n-tl discards N elements.

```
mod! N-LIST[X :: TRIV] {
  protecting (LIST[X])
  signature {
    op n-hd : Nat NeList -> List
    op n-tl : Nat NeList -> List
    op rev  : List -> List
    op nhd-aux : Nat NeList NeList -> NeList
    op rev-aux  : List List -> List
  }
  axioms {
    var N : Nat     vars L L'  : List   var X : Elt

    eq n-hd (N, L) = nhd-aux (N, L, nil) .
    ceq nhd-aux (N, L, L') = rev(L')  if N == 0 .
    ceq nhd-aux (N, (X L), L') = nhd-aux ((N - 1), L, (X L')) if N > 0 .

    ceq n-tl (N, L) = L  if N == 0 .
    ceq n-tl (N, (X L)) = n-tl ((N - 1), L) if N > 0 .

    eq  rev L = rev-aux(L,nil) .
    eq rev-aux(nil,L') = L' .
    eq rev-aux((X L), L') = rev-aux(L, (X L')) .
```

```
  }
}
```

The above examples also show a typical use of modules in a structured way. (1) The module LIST defines a basic data structure (List) by providing constructors and observers. (2) Another module N-LIST introduces further utility functions with importing the LIST module. Such utility modules are expected to constitute a reusable library.

A.2 Concurrent Object

Representing object follows a style of Maude [31]. The core part of the Maude concurrent object can easily be encoded in CafeOBJ [38]. The Maude model relies on `Configuration` data and rewriting rules based on concurrent rewriting logic. `Configuration` is a snapshot of global states consisting of objects and messages at some particular time. Object computation (sending messages to objects) proceeds as rewriting on `Configuration`. In addition, Maude has a concise syntax to represent the object term (`<(_:_)|_>`) and some encoding techniques to simulate *inheritance*. The Maude model can be considered as a standard encoding for concurrent objects in algebraic specification languages [35][50][51].

Below is an example of object definition. The module ITERATOR defines an Iterator object, which maintains a list of data and returns the specified number of data when requested by a `next-n` message.

```
mod! ITERATOR[X :: TH-ITERATOR-AID, Y :: TH-ITERATOR-MSG] {
  extending (ROOT)
  protecting (ITERATOR-VALUE)
  [ IteratorTerm < ObjectTerm ]
  [ CIdIterator < CId ]
  signature {
    op <(_:_)|_> : OId CIdIterator Attributes -> IteratorTerm
    op Iterator : -> CIdIterator
  }
  axioms {
    vars O R : OId    var L : List    var N : NzNat
    var REST : Attributes

  ctrans next-n (O,N,R) <(O : Iterator)|(body = L), (REST)>
  =>  <(O : Iterator)|(body = n-tl(N,L)), (REST)>
      return(R,true) outArgs(R,n-hd(N,L))      if N <= |L| .
```

```
  ctrans next-n (O,N,R) <(O : Iterator)|(body = L), (REST)>
  => <(O : Iterator)|(body = L), (REST)> return(R,false)  if N > |L| .

  trans destroy(O,R)  <(O : Iterator)|(REST)> => void(R) .
  }
}
```

The ITERATOR is a parameterized module. Both TH-ITERATOR-AID and TH-ITERATOR-MSG provide specification of the parameter module. The former introduces the attribute name that an Iterator object has, and the latter defines all the messages that the object can respond to.

```
mod* TH-ITERATOR-AID {
  extending (AID)
  signature {  op body : -> AId   }
}

mod* TH-ITERATOR-MSG {
  extending (MESSAGE)
  signature {
    op next-n : OId NzNat OId -> Message
    op destroy : OId OId -> Message
   }
}
```

The module ITERATOR imports two other modules ROOT and ITERATOR-VALUE. The module ROOT is a runtime module that provides the symbols necessary to represent Maude concurrent objects [38]. That is, it provides the following sort symbols: Configuration to represent the snapshot, Message for messages, ObjectTerm for the body of objects which consists of Attributes (a collection of attribute name and value pairs), CId for class identifiers, and OId for identifiers of object instances.

As shown in the above example, a user-defined class should define a concrete representation of the object term (<(_:_)|_>) in a new sort (IteratorTerm) and a class identifier constant (Iterator) in another new sort (CIdIterator). The axioms part has a set of rewriting rules (either trans or ctrans), each of which defines a method body. In writing the method body, one often refers to symbols defined in other modules such as, for example, the sort List and the related utility functions. The module ITERATOR-VALUE is supposed to import all the modules such as N-LIST[NAT] necessary for the ITERATOR.

Acknowledgements

We would like to thank Professor Tetsuo TAMAI (The University of Tokyo) for helpful discussions.

References

[1] Alencar, A. and Goguen, J. : OOZE: An Object Oriented Z Environment, Proc. ECOOP'91, pp.180-199 (1991).

[2] Beck, K. and Cunningham, W.: A Laboratory for Teaching Object Oriented Thinking, Proc. OOPSLA'89, pp.1-6 (1989).

[3] Beppu, Y., Nakajima, S., Kumeno, F., Cho, K., Hasegawa, T., and Ohsuga, A. : A Directory Server for Mobile Agent Interoperability, Proc. IEEE EDOC 2000, pp.144-148 (2000).

[4] Booch, G. : Object-Oriented Development, IEEE Trans. Soft. Engin., vol. SE-12, no. 2, pp.211-221 (1986).

[5] Booch, G., Rumbaugh, J. and Hopkins, J. : *The Evolution of Object Methods*, Handout, Rational Software Corporation, 1995.

[6] Carrington, D., Duke, D., Duke, R., King, P., Rose, G., and Smith, G. : Object-Z : An Object-Oriented Extension to Z, Proc. FORTE'89, pp.281-296 (1990).

[7] Carroll, J.M. (ed.) : *Scenario-Based Design*, John Wiley & Sons 1995.

[8] Clavel, M., Eker, S., Lincoln, P., and Meseguer, J. : Principles of Maude, Proc. 1st Workshop on Rewriting Logic and its Applications (1996).

[9] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. : *Object-Oriented Development: The Fusion Method*, Prentice Hall 1994.

[10] Diaconescu, R. and Futatsugi, K. : *The CafeOBJ Report*, World Scientific 1998.

[11] Diaconescu, R. and Futatsugi, K. : Behavioural Coherence in Object-Oriented Algebraic Specification, Journal of Universal Computer Science, vol.6, no.1, pp.74-96, (2000). The first version appeared as a JAIST Technical Report IR-RR-98-0017F (June 1998).

[12] D'Souza, D.F. and Wills, A.C. : *Objects, Components, and Frameworks with UML*, Addison-Wesley 1998.

[13] Easterbrook, S., Lutz, R., Kelly, J., Ampo, Y., and Hamilton, D. : Experiences Using Lightweight Formal Methods for Requirements Modeling, IEEE Trans. Soft. Engin., vol. SE-24, no. 1, pp.4-14 (1998).

[14] Futatsugi, K., Goguen, J., Jouannaud, J-P., and Meseguer, J. : Principles of OBJ2, Proc. 12th POPL, pp.52-66 (1985).

[15] Futatsugi, K. and Nakagawa, A.T. : An Overview of CAFE Specification Environment, Proc. 1st IEEE ICFEM (1997).

[16] Giovanni, R. and Iachini, P. : HOOD and Z for the Development of Complex Software Systems, Proc. VDM'90, pp.262-289 (1990).

[17] Goguen, J. and Meseguer, J. : Unifying Functional, Object-Oriented and Relational Programming in Logical Semantics, in *Research Directions in Object-Oriented Programming (Shriver and Wegner ed.)*, pp.417-477, MIT Press 1987.

[18] Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., and Jouannaud, J-P.: Introducing OBJ, in *Software Engineering with OBJ (Goguen and Malcolm ed.)*, pp.3-167, Kluwer Academic Publishers 2000. The first version appeared as an SRI Technical Report SRI-CSL-92-03 (1992).

[19] Goguen, J. and Malcolm, G. : A Hidden Agenda, Theoretical Computer Science, 245 (1), pp.55-101 (2000).

[20] Guttag, J. and Horning, J. : *Larch: Languages and Tools for Formal Specification*, Springer-Verlag 1993.

[21] Hall, J. : Using Z as a Specification Calculus for Object-Oriented System, Proc. VDM'90, pp.290-318 (1990).

[22] Helm, R., Holland, I., and Gangopadhyay, D. : Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, Proc. OOPSLA/ECOOP'90, pp.169-180 (1990).

[23] Hutt, T.F. (ed.) : *Object Analysis and Design : description of methods*, John Wiley & Sons 1994.

[24] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. : *Object-Oriented Software Engineering*, Addison-Wesley 1992.

[25] Jackson, D. : Alloy: A Lightweight Object Modelling Notation, Technical Report, MIT (2000).

[26] Jensen, K. : *Coloured Petri Nets 1*, Springer-Verlag 1992.

[27] Kobryn, C. : UML 2001: A Standardization Odyssey, CACM vol.42, no.10, pp.29-37 (1999).

[28] Lano, K. and Haughton, H. (ed) : *Object-Oriented Specification Case Studies*, Prentice Hall 1994.

[29] Lano, K. : *Formal Object-Oriented Development*, Springer-Verlag 1995.

[30] Leavens, G. and Cheon, Y. : Preliminary Design of Larch/C++, Proc. 1st Workshop on Larch, pp.159-184 (1993).

[31] Meseguer, J. : A Logical Theory of Concurrent Objects and its Realization in the Maude Language, in *Research Directions in Concurrent Object-Oriented Programming (Agha, Wegner and Yonezawa ed.)*, pp.314-390, MIT Press 1993.

[32] Meyer, E. and Souquières, J. : A Systematic Approach to Transform OMT Diagrams to a B Specification, Proc. FME World Congress on FM'99, pp.875-895 (1999).

[33] Milner, R., Tofte, M., Harper, R., and MacQueen, D. : *The Definition of Standard ML (revised)*, MIT Press 1997.

[34] Nakajima, S. : Formalizing Object-Oriented Software with Algebraic Specification Techniques, in *Understanding Object-Model Concepts*, Bringham Young University, BYU-CS-93-12 (1993).

[35] Nakajima, S. and Futatsugi, K. : An Object-Oriented Modeling Method for Algebraic Specifications in CafeOBJ, Proc. ACM/IEEE ICSE'97, pp.34-44 (1997).

[36] Nakajima, S. : Using Algebraic Specification Techniques in Development of Object-Oriented Frameworks, Proc. FME World Congress on FM'99, pp.1664-1683 (1999).

[37] Nakajima, S. : Aspect-Centered Design of Object-Oriented Frameworks, Trans. IPS Japan, Vol.41, No.3, pp.758-766 (2000).

[38] Nakajima, S. : An Algebraic Approach to Object-Oriented Software Engineering, PhD Thesis, The University of Tokyo (2000).

[39] OMG : UML v1.3 (http://www.omg.org/uml/).

[40] Reenskaug, T, : *Working with objects: the oocam software engineering method*, Manning Publications 1996.

[41] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W.: *Object-Oriented Modeling and Design*, Prentice-Hall 1991.

[42] Sharble, R. and Cohen, S. : The Object-Oriented Brewery : A Comparison of Two Object-Oriented Development Methods, ACM SIGSOFT Soft. Engin. Notice, vol.18, no.2, pp.60-73 (1993).

[43] Spivey, J. : *The Z Notation (2ed edition)*, Prentice Hall 1992.

[44] Stepney, S, Barden, R. and Cooper, D. (ed) : *Object Orientation in Z*, Springer-Verlag 1992.

[45] Tamai, T. : How Modeling Methods Affect the Process of Architectural Design Decisions: A Comparative Study, Proc. 8th IWSSD, pp.125-134 (1996).

[46] Tomono, M., Yamanaka, A., Tonouchi, T., and Nakajima, S. : An Implementation of Customizable Services with Java/ORB Integration, Proc. IEEE GLOBECOM'97, pp.1719-1723 (1997).

[47] Tonouchi, T., Fukushima, T., Manki, A., and Nakajima, S. : An Implementation of OSI Management Q3 Agent Platform for Subscriber Networks, Proc. IEEE ICC'97, pp.889-893 (1997).

[48] Wirfs-Brock, R., and Wilkerson, B. : Object-Oriented Design: A Responsibility-Driven Approach, Proc. OOPSLA'89, pp.71-75 (1989).

[49] Wirfs-Brock, R., Wilkerson, B., and Wiener, L.: *Designing Object-Oriented Software*, Prentice-Hall 1990.

[50] Wirsing, M. : A Formal Approach to Object-Oriented Design, 1995.

[51] Wirsing, M. and Knapp, A. : A Formal Approach to Object-Oriented Software Engineering, Proc. 1st Workshop on Rewriting Logic and its Applications (1996).

[52] Yamasaki, T. : Surveys of Program Design Methods Using a Common Example Problem (in Japanese), Journal of IPS Japan, vol. 25, no. 9, p.934 (1984).

[53] Zave, P. and Jackson, M. : Conjunction as Composition, ACM Trans. Soft. Engin. Meth. 2(4), pp.379-411 (1993).