Using Algebraic Specification Techniques in Development of Object-Oriented Frameworks

Shin Nakajima

NEC C&C Media Research Laboratories, Kawasaki, Japan

Abstract. This paper reports experience in using CafeOBJ (a multi-paradigm algebraic specification language) in the development of object-oriented frameworks of the ODP trader that is implemented with Java and JavaIDL. We first identify several aspects in the target problem before applying the known techniques of developing object-oriented frameworks. We use CafeOBJ to describe each aspect solution to mechanically check the integrity of the descriptions when all the aspects are put together. Although the experience is based on a particular problem only, the proposed method is clear enough to give a systematically and sharply focused help in reaching the solution, and to illustrate practice of using formal methods in the process.

1 Introduction

Formal methods are finding increasingly widespread use in the development of complex software systems and several notable projects are reported that have used the technology successfully [9]. The technology, however, has not yet reached the level where most software engineers use formal methods in their daily work. The basis of formal methods is mathematically-based languages for specifying and verifying software systems. Generally a specification language is used for writing functional properties and is able to handle only one aspect of the system. Thus, we often rest satisfied with specifying essential or critical properties of the system even when we use formal methods.

In real world software systems, however, characteristics other than functional properties such as ease of customization or maintenance are equally important. Software system, at the same time, can be viewed as an aggregate of various heterogeneous, often interrelated, subproblems [17]. Identifying aspects that can be tackled with a formal specification language is sometimes the most difficult task. Establishing the methodological principles of selecting and applying formal software development techniques is important[7].

This paper reports experience in using an algebraic specification language CafeOBJ [11][13] in the development of the ODP trading service [1], and in particular focuses on the design method. The trading server is implemented as object-oriented frameworks [10][18] so that it has a well-organized architecture making it easy to customize. We first identify a set of distinct aspects in the problem. Next, we refine and elaborate each aspect by using various specification

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1664–1683, 1999.

techniques for each one. Because the basic idea of identifying distinct aspects is decomposition, completing the system requires an integration of all. We use CafeOBJ to write solutions of each aspect, and thus checking the integrity is made possible when all the aspect descriptions are put together.

The present paper is organized as follows: section 2 explains the ODP trading service server. Section 3 reports on our experience in design and implementation of the ODP trading service server. Section 4 concludes the paper, and the appendix gives small example CafeOBJ specifications as an introduction to the language.

2 Trading Service Server

To facilitate the construction of distributed application systems, common service objects for distributed computing environments are being established [2]. The ODP trader provides functionalities to manage service in an open globally distributed environment [1].

2.1 The ODP Trader

The ODP trader is widely accepted because it is defined following the RM-ODP [24] (a result of a longterm joint effort by ISO and ITU-T) that aims to provide a general architectural framework for distributed systems in a multi-vendor environment. The standard document of the ODP trading function follows the guideline of the RM-ODP, and describes three viewpoints: enterprise (requirement capture and early design), information (conceptual design and information modeling), and computational (software design and development). The information viewpoint uses the Z notation to define basic concepts, the trader state, and a set of top-level operations visible from the outside. The computational viewpoint provides a decomposition of the overall functionality into several components and their interactions. It uses IDL [2] to describe the basic datatypes and the top-level operation interfaces, and it supplements the descriptions of all functional properties by using natural language. Actually, the specification written in IDL comprises five major functional interfaces: `Lookup`, `Register`, `Admin`, `Link`, and `Proxy`. The computational viewpoint is technically aligned with the OMG trading object service [3], although the OMG trader is restricted to manage service implemented as CORBA objects[1].

2.2 Trading Functions and Design Aspects

Figure 1 shows a trader and the participants in a trading scenario. The `Exporter` exports a service offer. The `Importer` imports the service offer and then becomes

[1] In this paper, the ODP/OMG trader refers to the computational viewpoint specification.

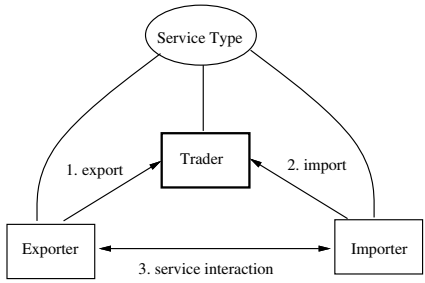

Fig. 1. Trading Scenario

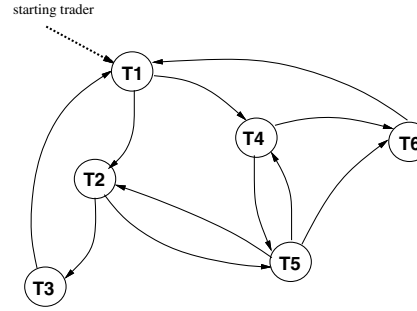

Fig. 2. Federated Trader Group

Aspect (example)	Specification Technique
common concept (service type)	abstract datatype
policy (scoping policy)	functional programming
algorithm (query, federation)	stream-style programming
language (constraint language)	denotational semantics style
functional object (Lookup)	concurrent object
architecture	concurrent object

Table 1. Aspects and Specification techniques

a client of the service. The `Trader` mediates between the two by using the exported service offers stored in its own repository which is ready for import requests. Every service offer has a service type and is considered to be its instance. The service type holds the interface type of the object being advertised and a list of property definitions. A property is a triple of name, type of the value and mode which indicates whether the property is mandatory or optional. Further, a subtype relation is defined between service types. The relation is useful both in importing offers that do not exactly match the request and in defining a new derived service type from the existing ones. The common concept defined in the ODP/OMG trader specification document, such as the service type, the service offer or the property definition, plays a central role in the trading scenario, and thus proper understanding of the concepts is important in designing the trader server.

Because importing, implemented as the `Lookup` interface, is the most complex and interesting function, this paper focuses on its design and implementation. The following IDL fragment shows a portion of a `query` operation of the `Lookup` interface. It is the operation for importing.[2]

```
typedef Istring ServiceTypeName;
typedef Istring Constraint;
typedef Istring Preference;
```

[2] Parameters not relevant here are omitted for brevity.

```
void query(
  in ServiceTypeName type,
  in Constraint constr,
  in Preference pref,
  ...
  out OfferSeq offers,
  out OfferIterator offer_itr,
  ...
) raises ( ... )
```

The first parameter `type` specifies the service type name of requested offers. The parameter `constr` is a condition that the offers should satisfy and is a *constraint language* expression that specifies the condition in a concise manner. The expression describes semantically a set of property values of service offers that the client tries to import. The trader searches its repository to find service offers whose property values satisfy the constraint expression. Understanding the search process requires an explicit formulation of the constraint language, and its formal definition would be valuable.

The parameter `pref` is preference information that specifies that the matched offers are sorted according to the preference rule. The sorted offers are returned to the importer in the `out` parameters: `offers` and `offer_itr`. The standard specification also defines a set of *scoping policies* to provide the upper bounds (cardinalities) of offers to be searched at various stages of the search process. Actual values of the cardinalities are determined by a combination of the importer's policies and the trader's policies. Understanding the role of each scoping policy requires grasping the global flow of the base query algorithm.

The ODP/OMG trader defines the specification for interworking or federation of traders to realize scalability. The use of a federated trader group enables a large number of service offers to be partitioned into a set of small offer sets of manageable size. One trader is responsible for each partition and works with the other traders when necessary. Figure 2 shows an example of a federated trader group. The traders T1 to T6 are linked as indicated by the curved arrows. When a `query` is issued on the starting trader T1 and a federated search is requested, traders T2 to T6 also initiate local searches. All the matching offers are collected and returned to the client importer.

The federation process uses a set of policies controlling the graph traversal. A simple one is the `request_id` that cuts out unnecessary visits to the same trader, and another is the `hop_count` that restricts the number of traders to visit. A set of policies called the *FollowOption* controls the traversal semantically. For example, a link marked with `if_no_local` is followed only if no matched offer is found in a trader at the source of the link. Again, the role of each policy is hard to understand without referring to the global flow of the base federation algorithm.

The ODP/OMG standard describes the query and federation algorithm and the role of each policy by using illustrative examples. In particular, the explanation adapts a stream processing style of selecting appropriate offers from an initial candidate set. The overall picture, however, is hard to grasp because the

descriptions are informal and scattered over several pages of the document [1][3]. A concise description is needed to prepare a precise design description of the algorithm, and functional programming style is a good candidate.

After the analysis of the trader specification mentioned above, we come to a conclusion that the six aspects in table 1 are mixed together to form the whole system. Table 1 shows the aspects and the accompanying specification techniques. In summary, the ODP/OMG trader is a medium scale, non-trivial problem that has six heterogeneous subproblems. Since the aspects are quite distinct in nature, no general-purpose methodology is adequate. Using specification techniques suitable for each aspect is a better approach to a systematically and sharply focused help in reaching the solution.

3 Design and Implementation

We use Java [5] as the implementation language and JavaIDL [19] as the ORB (Object Request Broker) to implement the trading service server. We also adapt the object-oriented framework technology to construct a highly modular system that aims to allow future customizations and ease of maintenance.

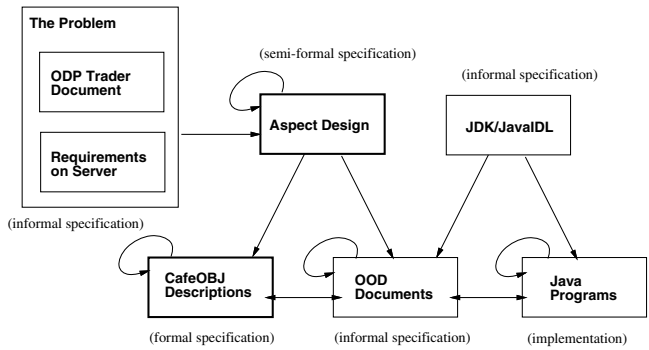

Fig. 3. Development Process

3.1 Overview of Development Process

Figure 3 summarizes the development process, which is one adapted from a process based on the parallel-iterative model of software development [26].

The aspect-centered design phase starts with the analysis of the ODP document and of what is required on the target system. Then the specification techniques that best describe the characteristics of each of the aspects are selected (Table 1). From the semi-formal description of the aspect design, informal specification descriptions are obtained with the help of standard techniques using

collaboration-based design [6][8][22] and design patterns [14][18]. In particular, the collaboration-based design, sometimes called scenario-based design, focuses on the analysis of interaction patterns between participant objects, and promotes to use notations such as MSC (Message Sequence Chart). During this phase, the specifications of the JDK library and JavaIDL are referred to. While the informal specifications and the implementation are being prepared, CafeOBJ descriptions of the aspects are also being prepared as the formal specification documents.

The aspect-centered design is quite useful because the ODP/OMG trading service server is a complicated specification. Identifying six important aspects and refining and elaborating each one with an appropriate design technique individually facilitates the design and implementation of the object-oriented frameworks. Aspect design alone, however, comes with one drawback.

The basic idea of the aspect design is decomposition of a complicated problem into a set of various aspects. Conversely, completing the system requires integration of the solutions of each aspect. Without the formal specification descriptions that can be mechanically analyzable, we only have a combination of mostly analyzable functional programs and unanalyzable graphical notations such as MSC. What could do is human design review only. Instead we use a specification language to describe solution descriptions of each aspect, and thus make it easy to check integrity when all the aspect descriptions are put together.

Our choice is a multiparadigm algebraic specification language CafeOBJ, which has clear semantics based on hidden order-sorted rewriting logic [11][13]. The logic subsumes order-sorted equational logic [12][15], concurrent rewriting logic [20], and hidden algebra [16].[3] Being an algebraic specification language, CafeOBJ promotes a property-oriented specification style: the target system is modeled as *algebra* by describing a set of properties to be satisfied. By introducing suitable specification modules (*algebra*), various computational models ranging from MSC[4] to functional programming and concurrent objects can be encoded in CafeOBJ. Further because CafeOBJ has clear operational semantics, specifications written in it are executable. It helps much to validate functionality of the system. The appendix provides a brief introduction to CafeOBJ.

3.2 CafeOBJ Descriptions of Aspect Solutions

This section deals with some example CafeOBJ descriptions of the aspect solution.

Common Concepts Of the entries in Table 1, the common concepts such as `ServiceType` and `PropertyDefinition` are easily translated into CafeOBJ modules because abstract datatype technique provides a concise way to model such basic vocabularies.

[3] We will not consider hidden algebra in this paper.

[4] Basically a transition system. The use of MSC in object-oriented modeling and its encoding method in CafeOBJ are described in [22].

The module SERVICE-TYPE-NAME introduces a new sort ServiceTypeName that specifies a set of service type name.

```
mod! SERVICE-TYPE-NAME { [ ServiceTypeName ] }
```

Service type is defined as an abstract data type. The module SERVICE-TYPE provides a concise notation to represent record-like terms and accessor functions. The module SERVICE-SUBTYPING defines the service subtyping relationship. Actually _is-subtype-of_ is a predicate used to calculate whether the operand service types satisfy the subtyping relationship.

```
mod! SERVICE-TYPE {
  [ ServiceType ]
  protecting (PROPERTY-DEFINITION)
  protecting (INTERFACE)
  protecting (SERVICE-TYPE-NAME)
  signature {
   op [name=_, interface=_, properties=_] :
       ServiceTypeName Interface PropertyDefinition -> ServiceType
   op _.name : ServiceType -> ServiceTypeName
   op _.interface : ServiceType -> Interface
   op _.properties : ServiceType -> PropertyDefinition
   op _.property(_) : ServiceType PropertyName -> ModeAndType
   op _.names : ServiceType -> Seq<PropertyName>
  }
  axioms {
    var S : ServiceTypeName  var T : ServiceType
    var N : PropertyName  var I : Interface  var PS : PropertyDefinition

  eq ([name=(S), interface=(I), properties=(PS)]).name = S .
  eq ([name=(S), interface=(I), properties=(PS)]).interface = I .
  eq ([name=(S), interface=(I), properties=(PS)]).properties = PS .
  eq ([name=(S), interface=(I), properties=(PS)]).property(N) =
                                                          lookup(PS,N) .
  eq (T).names = names((T).properties) .
  }
}

mod! SERVICE-SUBTYPING {
  protecting (SERVICE-TYPE)
  signature { op _is-subtype-of_ : ServiceType ServiceType -> Bool }
  axioms {
    vars T1 T2 : ServiceType

  eq (T1) is-subtype-of (T2)
  =   (((T1).interface is-subinterface-of (T2).interface)
  and ((T1).names includes (T2).names))
  and (mode-strength((T2).names,(T1).properties,(T2).properties)) .
  }
}
```

Query Algorithm and Policy The *policy* of the ODP/OMG trader is just a parameter that modifies the behavior of both local and federated query algorithms. It is hard to understand the meaning of policies without referring to the basic algorithm. Additionally in order to grasp the global behavior of the algorithm at a glance, a concise notation is needed. Notation borrowed from the functional programming language StandardML[21] is used, and some symbols for describing and handling set-like collections of data are added. The query processing is in particular viewed from a stream-based functional programming style. This viewpoint is in accordance with the informal presentation in the ODP/OMG document [1][3]. Below CafeOBJ modules are explained with referring to the pseudo StandardML descriptions.

The top-level function `IDLquery(T,I)`, which is invoked as an IDL request takes the following form. All the function definitions are supposed to come in the lexical context (as `fun` $\cdots$) of the `IDLquery(T,I)`. They use `T` and `I` freely as global constants, where `T` refers to the trader state and trader's policy and `I` is the importer's request and policy.

```
fun IDLquery(T,I) =
  fun query() = if valid-trader()
    then if valid_id() then (select ∘ federation ∘ search)(T.offers) else φ
    else IDLquery(remote_trader(T),I)
  fun ···
  in
    query()
  end
```

`IDLquery(T,I)` calls `query` to check whether the request is on the trader itself. Then, it invokes the body of `query` function, which is a stream-style processing consisting of `search`, `federation`, and `select`. The module `QUERY-ALGORITHM` is a CafeOBJ description of `IDLquery(T,I)`.

```
mod! QUERY-ALGORITHM [X :: TH-TRADER-STATE, Y :: TH-IMPORTER-REQUEST ] {
  signature {
    op query' : TraderState Request Set<Offer> -> Seq<Offer>
    op valid-trader : TraderState TraderName  -> Bool
    op valid-request-id : TraderState RequestId -> Bool
  }
  axioms {
    var T : TraderState  var I : Request var S : Set<Offer>

    eq query'(T,I,S) = if valid-trader(T,(I).starting-trader)
                     then (if valid-request-id(T,(I).request-id)
                           then select(T,I,federation(T,I,search(T,I,S)))
                           else empty<Offer> fi)
                      else delegate(T,I) fi .
  }
}
```

The function `search` collects candidate offers. The candidate space is then truncated according to appropriate policies on cardinality. The `search` uses two such cardinality filters. The functions `select` and `order` describe the specifications for the preference calculation.

fun search(R) = (match_cardinality_filter ∘ match
∘ search_cardinality_filter ∘ gather)(R)
fun select(R) = (return_cardinality_filter ∘ order)(R)
fun order(R) = order_on_preference(R,I.preference)

The module `SEARCH-CARDINALITY`, for example, defines the way that the *search cardinality* is calculated by using a trader's policy and an importer's policy.

```
mod! SEARCH-CARDINALITY [X :: TH-TRADER-POLICY, Y :: TH-IMPORTER-POLICY]{
  signature {
    op search-cardinality : TraderPolicy ImporterPolicy -> Cardinality
  }
  axioms {
    var T : TraderPolicy   var I : ImporterPolicy

    eq search-cardinality(T,I)
    = if exist((I).search-card)
         then min((I).search-card,(T).max-search-card)
         else (T).def-search-card fi .
  }
}
```

The function `federation(R)` controls a federated query process. It first checks whether further IDL query requests to the linked traders are necessary by consulting the trader's policy on the `hop_count`.

fun federation(R)
= let val new_count = new_hop_count()
in
if new_count ≥ 0 then traversal((I with new_count),R) else R
end

The function `traversal` is invoked with a modified importer policy (`J`) and the offers obtained locally (`R`), and it controls invocations on the target trader located at the far end of the specified link. The control again requires a scoping policy calculation, which involves the link policies as well as the trader's and the importer's policies. The two functions `new_importer_follow_rule(L,J)` and `current_link_follow_rule(L,J)` show how to use the `FollowOption` policy. Finally, the function `dispatch` shows the use of the `FollowOption` rule. The rule defines three cases – `local_only`, `if_no_local`, and `always` –. And how the final offers are constructed depends on the case.

fun traversal(J,R)
= $\bigcup_{\forall L \in T.links}$ dispatch_on(current_link_follow_rule(L,J), L,
(I with new_importer_follow_rule(L,J)),R)

fun current_link_follow_rule(L,J)
= if exist(J.link_follow_rule)
  then min(J.link_follow_rule, L.limiting_follow_rule, T.max_follow_policy)
  else min(L.limiting_follow_rule, T.max_follow_policy, T.def_follow_policy)
fun dispatch_on(local_only,L,J,R) = R
  | dispatch_on(if_no_local,L,J,R) = if empty(R) then follow(L,J) else R
  | dispatch_on(always,L,J,R) = follow(L,J) ∪ R

The module FOLLOW-OPTION describes *FollowOption* policy. Basically it provides min functions, and other functions are omitted for brevity. The module NEW-LINK-OPTION shows an example of calculating the *FollowOption* policy, which is used in the federation process.

```
mod! FOLLOW-OPTION {
  [ FollowOption ]
  signature {
    ops local-only if-no-local always : -> FollowOption
    op min : FollowOption FollowOption -> FollowOption
    op min : FollowOption FollowOption FollowOption -> FollowOption
    op _<_ : FollowOption FollowOption -> Bool
    ... (omitted) ...
  }
  axioms {
    vars F1 F2 F3 : FollowOption

    eq (F1) < (F2) =  (((F1 == local-only) and (not (F2 == local-only)))
                       or ((F1 == if-no-local) and (F2 == always))) .
    ceq min(F1,F2) = F1  if (F1)<(F2) .
    ... (omitted) ...
  }
}

mod! NEW-LINK-OPTION [X :: TH-TRADER-POLICY, Y :: TH-IMPORTER-POLICY,
                      Z :: TH-LINK-POLICY ] {
  protecting (FOLLOW-OPTION)
  signature {
    op current-link-follow-rule :
       TraderPolicy ImporterPolicy LinkPolicy -> FollowOption
  }
  axioms {
    var T : TraderPolicy   var I : ImporterPolicy  var L : LinkPolicy

    eq current-link-follow-rule(T,I,L)
    = if exist((I).link-follow-rule)
      then min((I).link-follow-rule, (L).limiting-follow-rule,
               (T).max-follow-policy)
      else min((L).limiting-follow-rule, (T).max-follow-policy,
               (T).def-follow-policy) fi .
  }
}
```

Constraint Language Two functions (`match` and `order_on_preference`) used in the query algorithm involve evaluation of a constraint expression and a preference expression. Each function is defined in such a way that it calls an evaluation function (either $\mathcal{CE}$ or $\mathcal{PE}$).

$$\text{fun match(R)} = \mathcal{CE}\ [\![\ \text{I.constraint}\ ]\!]\ \text{R}$$
$$\text{fun order\_on\_preference(R,X)} = \mathcal{PE}\ [\![\ \text{X}\ ]\!]\ \text{R}$$

The constraint language is defined in a standard way according to the denotational description of language semantics. First, the abstract syntax of the language, a portion of which is shown below, is defined.

```
CExp ::= Pred
Pred ::=  L  | Exp == Exp  | exist L          | not Pred
       | Pred and Pred     | Pred or Pred     | ...
```

Then, a valuation function for each syntax category is introduced; $\mathcal{CE}$ is an example one for constraint expressions (`CExp`) and it further calls $\mathcal{LE}$ of the valuation function for predicates (`Pred`). `R` stands for a set of offers and `O` is an offer.

$$\mathcal{CE} : \text{CExp} \rightarrow \text{R} \rightarrow \text{R}$$
$$\mathcal{LE} : \text{Pred} \rightarrow \text{O} \rightarrow \text{Bool}$$

The specifications of the constraint language interpreter or evaluator are given by the definitions of the valuation function. It can be defined systematically by studying the meaning of each abstract syntax construct.

$$\mathcal{CE}\ [\![\ \text{E}\ ]\!]\ \text{R} = \{\ \text{O} \in \text{R} \mid \mathcal{LE}[\![\ \text{E}\ ]\!]\ \text{O}\ \}$$
$$\mathcal{LE}[\![\ \text{L}\ ]\!]\ \text{O} = \text{prop-val(O,L)}{\downarrow_{Bool}}$$
$$\mathcal{LE}[\![\ \text{E1} == \text{E2}\ ]\!]\ \text{O} = \mathcal{AE}[\![\ \text{E1}\ ]\!]\ \text{O} == \mathcal{AE}[\![\ \text{E2}\ ]\!]\ \text{O}$$
$$\ldots$$

The CafeOBJ description is straightforward because the denotational-style description of language definition is translated easily in a standard way into CafeOBJ. The module `PRED-SYNTAX` defines the abstract syntax tree and the module `PRED-EVAL` provides the valuation function.

```
mod! PRED-SYNTAX {
  protecting (EXP-SYNTAX)
  [ Pred,  Exp < Pred ]
  signature {
    op  _==_  : Exp Exp -> Pred
    op  exist : Exp -> Pred
    op  not   : Pred -> Pred
    ... (omitted) ...
  }
}
```

```
mod! PRED-EVAL {
  protecting (PRED-SYNTAX)
  protecting (EXP-EVAL)
  signature { op EP(_)_ : Pred ServiceOffer -> Bool }
  axioms {
    vars E1 E2 : Exp  var P : Pred  var N : PropertyName
    var  O : ServiceOffer

  eq EP(label(N)) O = EE(label(N)) O .
  eq EP(E1 == E2) O = (EE(E1) O) == (EE(E2) O) .
  eq EP(exist E) O  = exist-property(O,(EE(E) O)) .
  eq EP(not P) O    = not(EP(P) O) .
  ... (omitted) ...
  }
}
```

Functional Objects and Architecture A functional object describes the behavior of interfaces such as the `Lookup` and `Register`, while the architecture here refers to a global organization of functional objects. We use collaboration-based design methods to refine and elaborate these aspects in order to identify the responsibilities of constituent objects, each of which is then translated into a Maude concurrent object [20]. The Maude model is a standard encoding of concurrent objects in algebraic specification languages [22][27]. Please refer to the appendix for a brief explanation on how to encode the Maude concurrent object model in CafeOBJ.

The module `IDL-LOOKUP` represents the CORBA object implementing the `Lookup` interface with its behavioral specification written in CafeOBJ. Other functional objects are `IDL-REGISTER`, `IDL-LINK`, `IDL-ADMIN`, `TRADER-STATE`, `TYPE-REPOSITORY`, and `OFFER-REPOSITORY`. The architecture is just a collection of the concurrent objects (`Configuration`), each of which is described by the corresponding module such as `IDL-LOOKUP`.

A `Lookup` object receiving a `query(O,N,C,P,Q,D,H,R)` message converts the input parameters into the representation that the part of the algorithm assumes, then invokes the body of the query algorithm (`query'`), and finally translates the result to match the IDL interface specifications.

```
mod! IDL-LOOKUP[X :: TH-LOOKUP-AID, Y :: TH-LOOKUP-MSG] {
  extending (ROOT)
  protecting (LOOKUP-VALUE)
  [ LookupTerm < ObjectTerm ,  CIdLookup < CId ]
  signature {
    op <(_:_)|_> : OId CIdLookup Attributes -> LookupTerm
    op Lookup : -> CIdLookup
    op invoke-query : TraderState ServiceTypeName Constraint Preference
                Seq<Policy> Seq<PolicyName> Nat Set<Offer> -> Seq<Offer>
   }
  axioms {
    vars O R R' : OId          var REST : Attributes   vars T U : OId
    var N : ServiceTypeName    var C : Constraint      var H : Nat
```

```
    var P : Preference        var Q : Seq<Policy>
    var D : Seq<PolicyName>   var X : TraderState    var S : Set<Offer>

  trans query(O,N,C,P,Q,D,H,R)
       <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R')),(REST)>
  => trader-state(T,O) initial-offers(U,O) m-wait(O,U,T)
       <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R)),(REST)> .

  trans m-wait(O,U,T) return(O,U,S) return(O,T,X)
        <(O : Lookup)|(client=(R)),(REST)>
  => void(R) outArgs(invoke-query(X,N,C,P,Q,D,H,S))
        <(O : Lookup)|(client=(R)),(REST)> .

  eq invoke-query(X,N,C,P,Q,D,H,S) = query'(X,request(N,C,P,Q,D,H),S) .
  }
}
```

On receiving a `query` message, the `Lookup` object sends messages to the `OfferRepository` object (`U`) and the `TraderState` object (`T`) to obtain a set of potential offers and the trader state. The `Lookup` object, then, invokes `query'`. As shown before in the module `QUERY-ALGORITHM`, `query'` is defined algorithmically. The above module assumes that the module `LOOKUP-VALUE` imports *library* modules such as `QUERY-ALGORITHM`.

Putting Together The last step is to put together all the aspect solutions to have a whole design artifact of the ODP/OMG trader. Each solution consists of one or more CafeOBJ module(s), and the integration is just to introduce a top level module that imports all the necessary ones. The following CafeOBJ module illustrates a way of integrating what is necessary to describe the whole system.

```
mod! WHOLE-SYSTEM {
  protecting (IDL-LOOKUP[TRADER-AID, TRADER-MSG])
  protecting (IDL-REGISTER[TRADER-AID, TRADER-MSG])
  protecting (IDL-LINK[TRADER-AID, TRADER-MSG])
  protecting (IDL-ADMIN[TRADER-AID, TRADER-MSG])
  protecting (TRADER-STATE[TRADER-AID, TRADER-MSG])
  protecting (TYPE-REPOSITORY[TRADER-AID, TRADER-MSG])
  protecting (OFFER-REPOSITORY[TRADER-AID, TRADER-MSG])
}
```

For example, instantiating the parameterized module `IDL-LOOKUP` with the appropriate modules produces a *algebraic model* of the `Lookup` object. The CAFE environment automatically imports all the modules by recursively traversing the module import relations of each module such as `protecting(LOOKUP-VALUE)` or `extending(ROOT)`. The process automatically involves syntax and sort checking.

3.3 Resultant Frameworks Written in Java

The trading server consists of several object-oriented frameworks (subsystems) that are refined and elaborated from the aspect solutions. This section focuses on two such subsystems. Figures 4 and 5 show the resultant frameworks written in Java.

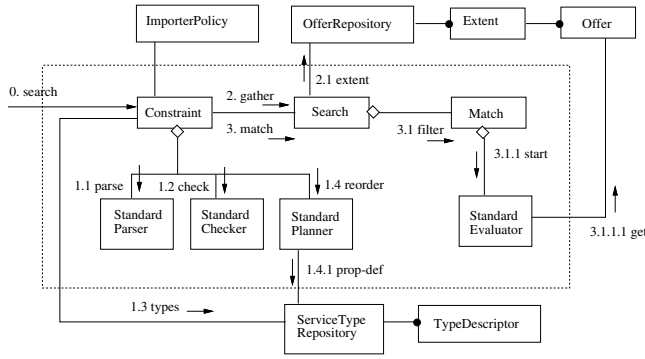

Fig. 4. Query Processing Framework

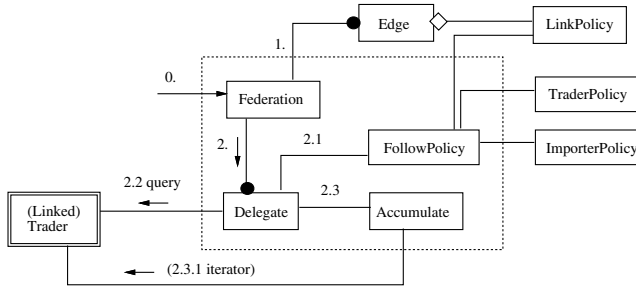

Fig. 5. Federation Framework

Based on the formal language definition presented early, designing the framework for constraint-language processing is straightforward (figure 4). This framework is a representative of using design patterns [14]: the composite pattern for representing the abstract syntax tree (AST) and the visitor pattern for representing the tree walkers such as a light semantic checker (`StandardChecker`), a filtering condition reorder planner (`StandardPlanner`), and an interpreter (`StandardEvaluator`).

Two offline support tools are used in implementing the constraint language processor: JavaCC [4](a public domain Java-based parser generator) and ASTG (a visitor-skeleton generator). ASTG accepts BNF descriptions of abstract syntax similar to the one in [25], and generates Java class definitions implementing

both the AST node objects and skeleton codes for tree-walking. The skeleton code follows a convention of the visitor design pattern. Completing the program is not difficult because the program code fragment that need to be written in the body part of the skeleton corresponds to the clauses of the valuation functions,

Figure 5 shows the object-oriented framework that implements the federation process. This subsystem is an example of using the collaboration-based design technique with an algorithmic representation of the aspect design as its input specification. The collaboration-based design focuses on the analysis of the interaction patterns or a sequence of messages between objects, and the algorithm expressed in the functional programming style can be considered to provide an abstract view of a sequence of messages between (potential) participant objects.

The design step involves (1) identifying participant objects in a heuristic manner, and (2) determining responsibility of each object [6][8] to be consistent with the algorithm description. For example, the algorithm described by the function `federation(R)` is divided into `Federation` and `Delegate`. Class `Delegate` corresponds to the body of the function `traversal(J,R)` and thus implements details of the algorithm. Class `Federation` is responsible for controlling the whole federation process and thus plays the role of Façade [14], which decouples the federation subsystem from the rest of the program and thus makes it easy for testing. Another example of the important design decision is encapsulating FollowOption calculation functions in class `FollowOption`, which aims to allow future customization of the policy.

3.4 Discussions

Reflection on Development Process The main development process consisted of three steps: the aspect design, the functional design, and the coding and testing. The aspect design step started with the study of the system requirements and ended with the semi-formal description of each aspects. The functional design step was one in which the collaboration-based object-oriented design method and the design pattern technique were used to produce design documents describing specifications of Java classes. It was followed by the coding and testing. We assigned one person (this author) to the aspect design and two engineers to the coding and testing. All three persons worked together to produce the design documents at the intermediate step. The engineers were not familiar with the collaboration-based object-oriented design technique and required *on the job* training. The functional design step involved technology transfer, and took far longer than initially planned. The time needed for coding and testing, however, was short for a program of this size.[5]

One advantage of the aspect design is that the resultant solutions contribute to provide a guideline for elaborating the design into object-oriented frameworks written in Java. Especially, the design is helpful to identify *hot spots* (program

[5] Roughly 25 K lines of Java, but the size is not constant because the program code is updated periodically.

points to be customized) and to determine the structure or the architecture of the Java program.

We could start to use CafeOBJ only when we almost reached a code complete stage of the first working prototype. It is partly because we had to finish most of the coding work as early as possible due to the time constraint of the financial support. We used the CafeOBJ descriptions to check the conformance of the informal design and the implementation against the ODP document.

Additionally, we think that most of the engineers would not accept CafeOBJ descriptions as their input source information because the engineers resisted even the semi-formal descriptions of the aspect design. Proper division of labor between project members is thus important when formal methods are used in real world projects.

Relation to Information Viewpoint As mentioned in section 2.1, the ODP trader recommendation follows the RM-ODP and presents three viewpoints: enterprise, information, and computational. Because the information and computational viewpoints deal with the ODP trader specification formally, establishing the relationship between the two viewpoints is also desirable. However, since the current recommendation uses more than one technique to specify essentially one functionality (the ODP trading function), there are some gaps between descriptions of the two viewpoints. Actually, the Z specification and the IDL specification define one concept through the use of respective language constructs. Since each language has different underlying computational models, the two kinds of specification are not easy to compare to see how both viewpoint specifications are really related. This is partly because each specification uses a lot of language-specific *idioms*.

We have presented CafeOBJ descriptions of the information viewpoint of the ODP trader elsewhere [23]. Together with the CafeOBJ descriptions in the present paper, these descriptions show that CafeOBJ can represent different abstract levels for different aspects of the specificand while providing the same specification fragments for a set of the common vocabulary such as `ServiceType` or `PropertyDefinition`. The CafeOBJ descriptions of the information viewpoint were used as *data dictionaries* in writing the present CafeOBJ descriptions.

Role of CafeOBJ Descriptions The basic idea of the aspect-centered design is to break a whole problem into a set of manageable subproblems that can be solved individually. Each aspect may have its own notation such as functional-style descriptions or message-sequence charts, and the descriptions of each aspect can be validated separately. On the other hand, the overall system cannot be described without integrating all the solution descriptions, and this is difficult when each aspect and its solution have a different notation.

We use CafeOBJ to write each aspect solution, and thus make it possbile to check the integrity when all the aspect descriptions are put together. First, we can get benefits from syntax and sort checking. This helps to identify what

is missing. Second, with appropriate input test terms, we can validate the design by test execution (specification animation). It also helps uncover logical inconsistency spreading over different aspect descriptions.

The CafeOBJ descriptions themselves form an artifact. It is an *analogic* model [17] in the sense that the artifact does not represent the Java program faithfully, but instead elucidates essential design in an abstract manner. We plan to use the model as a reusable design artifact when we develop a product line (a family of systems having similar functionalities) in future.

One of methodological advantages of CafeOBJ is to provide hidden algebra or behavioral equations[11][13][16]. Specifications using hidden algebra are basically statemachines, where sequences of allowable *events* describe the properties of the statemachine without defining the constituting states explicitly. The technique is adequate when certain properties are verified by using observational equivalence.

Contrarily, writing specifications in a constructive and operational manner is significant in the present development process. The CafeOBJ descriptions focus on the design aspects and the solutions such as the query and federation algorithm in detail. Such a detailed description acts as a starting point for the further refinement and elaboration. In summary, the CafeOBJ description is an *abstract* implementation in the present approach. It is worth investigating to compare the pro and cons of the two approaches (the present one and the hidden algebra approach) and to study their roles in the development process of object-oriented frameworks.

4 Conclusion

We have reported our experience in using CafeOBJ (a multiparadigm algebraic specification language) in developing object-oriented frameworks of the ODP/OMG trader that is implemented with Java and JavaIDL.

We have discussed the ways in which the introduction of an explicit design phase for the aspect analysis greatly helped us develop object-oriented frameworks for the trading server object. In addition, the use of a single specification language (CafeOBJ) made it possible to check integrity after all the aspect descriptions were put together. The formal descriptions in CafeOBJ contributed to raising credibility of the design description. Although our experience is based on the ODP/OMG trader only, the idea of the aspect design with a proper use of CafeOBJ is clear enough to illustrate practice of using formal methods in the development of object-oriented frameworks.

Acknowledgements

Discussions with Prof. Kokichi Futatsugi (JAIST) and Prof. Tetsuo Tamai (Univ. of Tokyo) were helpful in forming the idea presented in this paper. The comments from the anonymous reviewers helped much to improve the presentation.

References

[1] ITU-T Rec. X.950-1 : Information Technology - Open Distributed Processing - Trading Function - Part 1: Specification (1997).
[2] OMG : OMG CORBA (http://www.omg.org).
[3] OMG : CORBAservices, Trading Object Service Specification (1997).
[4] Sun Microsystems : JavaCC Documentation (http://www.suntest.com/JavaCC/).
[5] Arnold, K. and Gosling, J. : *The Java^{TM} Programming Language*, Addison-Wesley 1996.
[6] Beck, K. and Cunningham, W. : A Laboratory for Teaching Object-Oriented Thinking, Proc. OOPSLA'89, pp.1-6 (1989).
[7] Bjørner, D., Koussoube, S., Noussi, R., and Satchok, G. : Michael Jackson's Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools, Proc. 1st IEEE ICFEM (1997).
[8] Carroll, J.M. (ed.) : *Scenario-Based Design*, John Wiley & Sons 1995.
[9] Clarke, E.M. and Wing, J.M. : Formal Methods: State of the Art and Future Directions, ACM Computing Surveys (1996).
[10] Deutsch, L.P. : Design Reuse and Frameworks in the Smalltalk-80 Programming System, in *Software Reusability vol.2 (Biggerstaff and Perlis, ed.)*, pp.55-71, ACM Press 1989.
[11] Diaconescu, R. and Futatsugi, K. : *The CafeOBJ Report*, World Scientific 1998.
[12] Futatsugi, K., Goguen, J., Jouannaud, J-P., and Meseguer, J. : Principles of OBJ2, Proc. 12th POPL, pp.52-66 (1985).
[13] Futatsugi, K. and Nakagawa, A.T. : An Overview of CAFE Specification Environment, Proc. 1st IEEE ICFEM (1997).
[14] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. : *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1994.
[15] Goguen, J. and Malcolm, G. : *Algebraic Semantics of Imperative Programs*, The MIT Press 1996.
[16] Goguen, J. and Malcolm, G. : A Hidden Agenda, UCSD CS97-538 (1997).
[17] Jackson, M. : *Software Requirements & Specifications*, Addison-Wesley 1995.
[18] Johnson, R. : Documenting Frameworks using Patterns, Proc. OOPSLA'92, pp.63-76 (1992).
[19] Lewis, G., Barber, S., and Siegel, E. : *Programming with Java IDL*, John Wiley & Sons 1998.
[20] 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, The MIT Press 1993.
[21] Milner, R., Tofte, M., Harper, R., and MacQueen, D. : *The Definition of Standard ML (revised)*, The MIT Press 1997.
[22] Nakajima, S. and Futatsugi, K. : An Object-Oriented Modeling Method for Algebraic Specifications in CafeOBJ, Proc. 19th ICSE, pp.34-44 (1997).
[23] Nakajima, S. and Futatsugi, K. : An Algebraic Approach to Specification and Analysis of the ODP Trader, Trans. IPS Japan Vol.40 No.4, pp.1861-1873(1999).
[24] Raymond, K. : Reference Model of Open Distributed Processing (RM-ODP) : Introduction, Proc. ICODP'95 (1995).
[25] Wang, D.C., Appel, A.W., and Korn, J.L. : The Zephyr Abstract Syntax Description Language, Proc. USENIX DSL, pp.213-227 (1997).
[26] Wing, J. and Zaremski, A.M. : Unintrusive Ways to Integrate Formal Specifications in Practice, CMU-CS-91-113 (1991).
[27] Wirsing, M. and Knapp, A. : A Formal Approach to Object-Oriented Software Engineering, Proc. 1st Workshop on Rewriting Logic and its Applications (1996).

A CafeOBJ: The Specification Language

CafeOBJ has two kinds of axioms[6] to describe functional behavior [11][13]. An equational axiom (*eq*) is based on equational logic and thus is suitable for representing static relationships, whereas a rewriting axiom (*trans*) is based on concurrent rewriting logic and is suitable for modeling changes in some states.

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. `|_|` returns the length of the operand list data and is a recursive function over the structure of the list. The module `LIST` also defines some utility functions such as `n-hd` and `n-tl`.

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

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

The Maude concurrent object [20] can easily be encoded in CafeOBJ. The Maude model relies on a `Configuration` 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*. We regard the Maude model to be a standard encoding for concurrent objects in algebraic specification languages [22][27].

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. Actually, it is CafeOBJ encoding of the IDL iterator interface with functional behavior at an abstract level.

[6] We do not consider hidden algebra here.

```
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 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. 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 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, we often refer to symbols defined in other modules, for example the sort `List` and the related utility functions. The module `ITERATOR-VALUE` is supposed to import all the modules necessary for the `ITERATOR` such as `LIST[NAT]`.