Vol. 40 No. 4 Transactions of Information Processing Society of Japan Apr. 1999

Regular Paper

An Algebraic Approach to Specification and Analysis of the ODP Trader

SHIN NAKAJIMA[†] and KOKICHI FUTATSUGI[††]

The ODP trader is an important standard specification for a discovering service in distributed computing environments. We used CAFE specification environment to construct algebraic specifications of the information viewpoint of the ODP trader written in CafeOBJ. The resultant CafeOBJ specification is more constructive than the Z specification in the original standard document while both specifications are equally formal. Our experience has shown (1) that algebraic specification technique is useful to describe characteristics of the ODP trader and (2) that the resultant specification contributes to helping us understand the functionality of the ODP trader at an appropriate abstract level because specifications written in CafeOBJ are executable.

1. Introduction

The advancement of distributed software technology demands the establishment of common services for distributed computing environments. Examples of the common services include naming services, trading services, security services, and mobile agent facilities[19]. Since their specifications are consulted so often, the documents should be unambiguous and clear enough to be of great assistance in allowing us to understand the functionality easily.

ODP (Open Distributed Processing), a joint effort of ISO and ITU-T, aims to provide a general architectural framework for distributed systems in a multi-vendor environment[21]. Their activities involve the use and integration of FDTs (Formal Description Techniques) into the ODP from the early stages of defining RM-ODP (Reference Model of Open Distributed Processing)[11]. As a concrete service following RM-ODP, the ODP trader has been defined[1],[3]. The ODP trader is an important standard specification because it has recently become IS (the International Standard) and also it is technically aligned with the OMG specification for the trading object service[2],[23].

The present paper reports on our experience in using CAFE specification environment[13] and CafeOBJ (a multiparadigm algebraic specification language)[7],[8] in writing formal specifications of the ODP trader. Although the ODP trader has been extensively studied as an example case for applying FDTs[10],[11],[17], no specification using algebraic specification techniques has been published. We used CAFE specification environment to construct algebraic specifications of the information viewpoint of the ODP trader written in CafeOBJ. The resultant CafeOBJ specification is more constructive than the Z specification in the original standard document[1] while both specifications are equally formal. Our experinece shows (1) that algebraic specification technique is useful to describe characteristrics of the ODP trader and (2) that the resultant specifications contribute to helping us understand the functionality of the ODP trader at an appropriate abstract level because the specification is mechanically analyzable.

2. CAFE and CafeOBJ

CAFE is an advanced specification writing environment[13], and CafeOBJ is a powerful algebraic specification language which has clear semantics based on hidden-order sorted rewriting logic[7],[8]. The logic subsumes order-sorted equational logic[12],[14],[15], concurrent rewriting logic[18], and hidden algebra[16][☆]. Since CafeOBJ has clear operational semantics, specifications written in it can be executable.

Here is a simple example CafeOBJ specification LIST. The module LIST defines a generic abstract datatype List. The parameter TRIV specifies that the sort of the list element is Elt, which will be determined at the time of module instantiation. __ (juxtaposing two data in the specified sorts) is a List constructor. |_| re-

† NEC C&C Media Research Laboratories
†† Japan Advanced Institute of Science and Technology

☆ We only consider order-sorted equational logic of CafeOBJ because the logic has enough expressive power to specify the information viewpoint of the ODP trader.

turns the length of the operand list data and it is defined as a recursive function over the recursive structure of List. hd and tl are two standard accessor functions. The module also imports a library module NAT with protecting(NAT), by which all the definitions in NAT can freely be used in LIST. Since LIST is a parameterized module, we instantiate it with NAT to have a list of natural numbers NAT-LIST.

```
mod! LIST[X :: TRIV] {
  [ NeList List , Elt < NeList < List ]
  protecting (NAT)
  signature {
    op nil : -> List
    op __  : Elt List -> NeList {id: nil}
    op |_| : List -> Nat
    op hd  : NeList -> Elt
    op tl  : NeList   -> List
  }
  axioms {
    var X : Elt     var L : List

    eq | nil | = 0 .
    eq | X L | = 1 + | L | .
    eq hd(X L) = X .
    eq tl(X L) = L .
  }
}
mod! NAT-LIST { protecting (LIST[NAT]) }
```

We can load the file to the CAFE environment and validate the functionality of the specification. In the following, in is a file load command, select sets up a current module, and red is a command to start reduction of the operand term and return a normal form. The result is 5 that is the length of the given list.

```
CafeOBJ> in list.mod
CafeOBJ> select NAT-LIST .
NAT-LIST> red |(1 2 3 4 5)| .
5 : NzNat
```

3. The ODP Trader and FDTs

3.1 Overview of The Trading Function

Figure 1 shows the ODP trader and the participants in a trading scenario[1),3)]. Exporter exports a service offer. Importer imports the service offer and then becomes a client of the service. Trader mediates between the two by using the exported service offers stored in its own repository which is ready for any import request.

Every service offer has a service type which has the interface type of the object being advertised and a list of property definitions. A property definition consists of a property name, the type of the value, and a mode. The mode indicates whether the property value is mandatory, optional, or readonly. A service offer is a value which is consistent with the type informations in the property definitions of its service type.

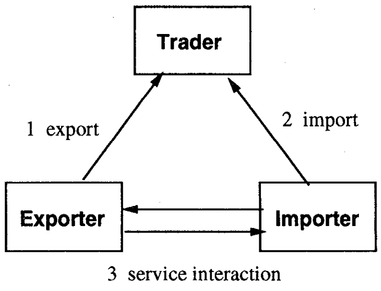


Fig. 1 The ODP trader and its context.

The importer specifies a list of pairs of property name and value for service offers that it tries to import. Then, the trader searches its repository to find service offers whose property values match the importer's request. The importer can further specify a preference information to specify that the matched offers are sorted according to the preference rule. Last, the sorted offers are returned to the importer to initiate service interactions. The trader defines a standard constraint language for specifying both the property and the preference in order to provide common understanding between the participants. Further, the ODP trading function is a complex specification since it has the concept for service subtyping and federation between traders. The subtyping rule is used to find offers that do not exactly match the request. The trader can be a member of a federated trader group that interworks to manage and handle service offers.

A concrete example might be a help in understanding a trading scenario☆. Some of the properties that the offer for the printer service has are the location of the printer server, resolution, color or black/white, the printer's name, and length of the printer queue. Of these properties, the length of the printer queue is dynamic and its exact value is obtained at the time of the request.

The importer tries to obtain the service offers of the printer based on a request which has a set of property values and a preference value.

☆ The example scenario, importing service offers of a printer service, is taken from Ref. 23).

For example, the importer specifies the location of a floor lower than the third in Building A, and true color properties in addition to a rule saying that a printer with a short queue is preferred. The trader searches its repository for offers that match the importer request. For values of properties that are not specified, the trader either uses default values that the trader has or just ignores them if the properties are optional. After the trader successfully finds a set of matched offers, it also sorts the set according to the preference rule. Thus, the trader returns the list of offers for the color printers with the shortest printer queue first. The importer receives the list of printer offers, and initiates a printing service on the desired printer.

3.2 The Standard Document and CafeOBJ

RM-ODP[21] introduces the concept of viewpoints for describing the system from various concerns. The five viewpoints are enterprise (requirement capture and early design of distributed systems), information (conceptual design and information modeling), computational (software design and development), engineering (system design and development), and technology (technology identification, procurement and installation). Of the five viewpoints, the information and computational are the most important ones in view of the application of FDTs and the standardization activities. The others are intrinsically informal (enterprise) or implementation-dependent (engineering and technology).

The standard document of the ODP trading function[1] follows RM-ODP and it describes three viewpoints; enterprise, information and computational. In particular, the information viewpoint sees the trading system from the viewpoint of a centralized system. It uses the Z notation[22] to define basic concepts, the trader state, and the set of the top-level operations that is visible from the outside. The computational viewpoint describes the functional aspects of the trading system, and provides a decomposition of the overall functionality into several components and their interactions. This viewpoint uses IDL[19] to describe basic datatypes and top-level operation interfaces, and it supplements all behavioral aspects in terms of natural language explanations. In this paper, we will concentrate on the information viewpoint of the ODP trader because the standard document well describes all the specifications, while the computational viewpoint part of the document leaves much to further design activities which is beyond the scope of the present paper.

Since the Z notation is a model-oriented formal specification language[22] that is based on axiomatic set theory (Zermelo set theory), the required mathematical background is not so advanced and thus software engineers who have adequate training in mathematics do not find it difficult to understand Z specifications. However, some specifications employ a lot of *idioms* specific to the Z notation. The description is almost a result of *hacking* at the worst. Further, it is hard to mechanically analyze specifications written in the full Z notation because of the richness of its background mathematics. For naive software engineers who are not familiar with mathematics, specifications written in an executable or operational manner are much more accessible. Actually, we can obtain executable specifications in CafeOBJ.

Because of the nature of axiomatic set theory, it is not always easy to translate Z specifications into CafeOBJ directly[25]. The Z notation allows higher-order specifications in the sense that expressions can be evaluated to be types (sets), while CafeOBJ, being an algebraic specification language, is essentially first-order: namely type (sort) and value are clearly distinct. Further, the Z notation can specify an infinite set by using either an existential quantifier or a set comprehension that employs some predicates to characterize the elements of the defining set. Contrarily it is difficult to encode infinite sets in CafeOBJ. Therefore, it is necessary to give some *interpretation* to each specification fragment in regard to how we use the fragment in the other part of the specifications. Since the specific translation will be discussed in Section 4 with detailed explanation on each decision, we only give some general rules for the translation in **Table 1**.

3.3 Related Work

Some literature discusses that the FDTs to be used in the standard specification documents such as the ODP trader should also be standardized and thus that Z, SDL or LOTOS are good candidates[5),11]. However, these FDTs are based on research long before RM-ODP and the ODP trader appeared, and there have been many new advances in FDT research since then. For example, the abstract data part of LOTOS is many-sorted while CafeOBJ has order-sorted

Table 1 General rules for translation.

	Z Notation	CafeOBJ
1	given name	sort symbol
2	state schema	sort symbol, constructor function symbol
3	operation schema	function symbol, equational axiom
4	property part of schema	function symbol, equational axiom

algebra☆. Moreover, the Z notation itself has not been approved to be IS (the International Standard) yet. We think that it is not necessary to stick to old FDTs. Instead, a new FDT with advanced features like CafeOBJ is sometimes better suited for specifying complex specifications such as the ODP trader.

Other researchers have proposed new FDTs for RM-ODP[4),6),9)]. Concerns here are (1) consistency between the information and computational viewpoints and (2) the possibility of mechanical analysis for the specificand. Most of the work has only concentrated on the first aspect. However, we see that mechanical analysis is valuable in allowing us to understand the specificand as well as providing consistent specifications. Although designing a new specification language for RM-ODP is valuable, it would also be worth investigating to study how existing formal specification languages are employed. It is because we can make use of existing techniques and tools for mechanically analysing specifications.

Apart from the Z specification of the standard document, Refs. 10), 11) and 17) provide formal specifications for the ODP trader. Fischbeck, et al.[10)] employ a combination of IDL and SDL to describe the computational viewpoint specification. Their tool generates executable codes of either C++ or Java. Thus, functional aspects of the computational viewpoint specifications can be validated by executions. Since the approach is based on generating programs, the emphasis is on the relationship between the computational and engineering viewpoints. Fischer, et al.[11)] is one early work applying FDTs to the specifications of the ODP application. They investigate Z, LOTOS and SDL in writing the specifications and conclude that no single FDT can specify the richness of the ODP trader. This work has had much impact on the current standard document[1)]. Lecero and Quemada[17)] present the E-LOTOS specification for the computational viewpoint of the ODP trader. Their purpose is to show how new language constructs of E-LOTOS are used to specify ODP applications. Since the language belongs to the LOTOS family, the specification style is process-oriented (the ODP trader consists of a set of E-LOTOS processes). It presents specifications that are organized differently from those in the standard document.

Last, Nakajima and Futatsugi propose to use algebraic specification technique to specify the ODP trader and present an overview of the project[20)]. Although the paper deals with both the information and computational viewpoints of the ODP trader, it does not include the CafeOBJ specifications in enough details or any concrete example analysis results. This paper concentrates on the information viewpoint and gives a detail account of specification and analysis of the CafeOBJ specifications.

4. A CafeOBJ Specification of the ODP Trader

In this section, we will show some specification fragments in the Z notation drawn from the standard document[1)] and the CafeOBJ counterparts. Our goals are (1) to write executable specifications so as to help us understand the functionality of the ODP trader at an abstract level, and (2) to write modular specifications so as to make it clear the correspondence of the modules with the elements in the standard document and at the same time to make it clear the role that each module has.

4.1 Some Specification Fragments

4.1.1 Basic Concepts

A given name introduces a new basic concept; *InterfaceSignatureType* and *Name* are examples of this. A free type definition specifies a set with elements in the defining set; the set *Mode* has four constants.

[*InterfaceSignatureType*, *Name*]
Mode ::= *normal* | *readonly* | *mandatory*
| *readonlymandatory*

A given name is encoded as a sort in CafeOBJ, and a free type is a sort with appropriate (constant) constructors.

☆ Actually, these advances have motivated a new addition to the LOTOS family, E-LOTOS[17)].

```
mod! SERVICE-TYPE {
  [ ServiceType ]
  protecting (INTERFACE-SIGNATURE-TYPE)
  protecting (PROPERTY-DEFINITIONS)
  signature {
   op [signature=_, prop-defs=_] : InterfaceSignatureType PropertyDefinitions -> ServiceType
   op _.signature  : ServiceType -> InterfaceSignatureType
   op _.prop-defs : ServiceType -> PropertyDefinitions
  }
  axioms {
    var I : InterfaceSignatureType    var P : PropertyDefinitions

    eq ([signature=(I), prop-defs=(P)]).signature = I .
    eq ([signature=(I), prop-defs=(P)]).prop-defs = P .
  }
}
```

Fig. 2 Module SERVICE-TYPE.

```
mod! INTERFACE-SIGNATURE-TYPE {
  [ InterfaceSignatureType ]
}
mod! MODE {
  [ Mode ]
  signature {
    ops normal readonly mandatory : -> Mode
    op readonlymandatory : -> Mode
  }
}
```

Some schemata represent a basic concept that is an aggregate of known concepts; *ServiceType* has two named components, *signature* of type *InterfaceSignatureType* and *prop_defs* of type *Name* $\nrightarrow$ (*ValueType* $\times$ *Mode*).

```
┌─ ServiceType ───────────────
│ signature : InterfaceSignatureType
│ prop_defs : Name ⇸ (ValueType × Mode)
└─────────────────────────────
```

Since *ServiceType* can be considered as structured data, its CafeOBJ specification employs a style similar to a record structure. The SERVICE-TYPE (**Fig. 2**) introduces a new sort ServiceType and one constructor which provides a record-like syntax, and two accessor functions. The axioms part gives definitions for the accessors by using equations. The module also imports two modules INTERFACE-SIGNATURE-TYPE and PROPERTY-DEFINITIONS that provide definitions for the symbols used.

4.1.2 Basic Library for Executable Specifications

Any executable specification in CafeOBJ should have an initial algebra model[7),8),12)]. A rule of thumb is that the definitions of basic data structures is to follow recursive structures; namely, they are based on recursively defined LIST (see Section 2) and they add appropriate utility functions to simulate other high-level functionalities such as the set-like collection of data. The parameterized module COLLECTION[X :: TRIV] is the one we provide. By using this library module, we have an executable CafeOBJ module for a type $\mathbb{P}$ *ElementData* in the Z notation; the module SERVICE-OFFER-S is such an example, which is $\mathbb{P}$ *ServiceOffer* in the Z specification.

```
mod! SERVICE-OFFER-S {
  protecting (COLLECTION[SERVICE-OFFER]
    *{ sort Collection -> ServiceOffers })
}
```

Actually the module COLLECTION introduces a sort Collection to represent (homogeneous) collection of some values. In defining the module SERVICE-OFFER-S with instantiating COLLECTION, we have *renamed* Collection to be ServiceOffers by using *the view mechanism* of CafeOBJ[8)].

In order to construct a basic library for the original Z specification having function types ($\nrightarrow$), we have introduced another parameterized module ENVIRONMENT. This follows the observation that the function types in Z are basically power sets of some relation, although not strictly so[☆].

$$X \nrightarrow Y \quad \Rightarrow \quad \mathbb{P}(X \times Y)$$

The ENVIRONMENT[X :: TRIV, Y :: TRIV] employs COLLECTION and 2TUPLE[14)] as its internal representation.

4.1.3 State Schema

The schema *TradingSystem* is the main state schema and it defines the global state space of federated traders. It defines data structure representing the state space with a set of well-formedness conditions as its properties. The

☆ The definition of the partial function in terms of the relationship is given in Ref. 22).

```
mod! TRADING-SYSTEM-AXIOMS {
  protecting (TRADING-SYSTEM)
  protecting (TRADING-SYSTEM-LIBRARY)
  signature {
    op trading-system-axiom : TradingSystem -> Bool
    op service-offer-equality : ServiceOffer ServiceOffer -> Bool
  }
  axioms {
  var T : TradingSystem   vars V1 V2 : ServiceOffer

  eq trading-system-axiom(T)
  =     ((dom-partition((T).partition)) equal-to ((T).offers))
    and ((ran-partition((T).partition)) is-subsumed-by ((T).nodes))
    and ((all-nodes-of ((T).edges)) is-subsumed-by ((T).nodes))
    and ((dom-edge-prop((T).edge-properties)) equal-to ((T).edges))  .

  eq service-offer-equality(V1,V2)  =  ((V1).offer-id == (V2).offer-id) .
  }
}
```

Fig. 3 Module TRADING-SYSTEM-AXIOMS.

schema shows that the state space consists of five components and that four predicates be asserted for the state space to be well-formed. The last line starting ∀ defines equality of *ServiceOffer* instances.

TradingSystem

$offers : \mathbb{P}\ ServiceOffer$
$nodes : \mathbb{P}\ Node$
$partition : ServiceOffer \twoheadrightarrow Node$
$edges : Node \leftrightarrow Node$
$edge\_properties : (Node \times Node) \twoheadrightarrow \mathbb{P}\ Property$

$\mathrm{dom}\ partition = offers$
$\mathrm{ran}\ partition \subseteq nodes$
$\mathrm{dom}\ edges \cup \mathrm{ran}\ edges \subseteq nodes$
$\mathrm{dom}\ edge\_properties = edges$
$\forall p, q : ServiceOffer \bullet p.service\_offer\_identifier$
$= q.service\_offer\_identifier$
$\Leftrightarrow p = q$

The declaration part is directly translated to a record style structure, which is in the same manner as the *ServiceType*. We defined TRADING-SYSTEM module for *TradingSystem*.

When we stand at the specification execution side, one way of using the property part of the *TradingSystem* is just to check the well-formedness of the *TradingSystem* record structure after invocation of, say an *export*. In other words, the property part is interpreted as a boolean-valued function that accepts a *TradingSystem* record instance as its argument.

Figure 3 shows a CafeOBJ module definition in accordance with this idea. The function `trading-system-axiom` is a conjunction of the four predicates of the Z specification. In addition, all the symbols such as `dom-partition` are defined by using either COLLECTION or ENVIRONMENT.

4.1.4 Operation Schema

The main trading functions are implemented as operations on the state schema *TradingSystem*. *Export* is an example function that exports a new service offer. The offer is subsequently stored in a repository the trader maintains, and it is actually stored in some components of the schema *TradingSystem*.

The following schema *ExportOK* defines a part of the definition of the export function☆. It accepts two input parameters and returns a value of *ServiceOfferIdentifier*, and it also leaves some changes in the *TradingSystem* state space. Further, the first two predicates are preconditions and the rest are postconditions; *offers* and *partitions* are updated accordingly while the others are left unchanged.

ExportOK

$\Delta\, TradingSystem$
$new\_offer? : ServiceOffer$
$node? : Node$
$service\_offer\_identifier! : ServiceOfferIdentifier$

$\forall s : offers \bullet s.service\_offer\_identifier$
$\neq new\_offer?.service\_offer\_identifier$
$node? \in nodes$
$offers' = offers \cup \{new\_offer?\}$
$partition' = partition \cup \{new\_offer? \mapsto node?\}$
$service\_offer\_identifier!$
$= new\_offer?.service\_offer\_identifier$
$nodes' = nodes$
$edge\_properties' = edge\_properties$
$edges' = edges$

☆ A case where the precondition becomes false is also documented [1].

```
mod! EXPORT-BEHAVIOR {
  protecting (TRADING-SYSTEM)
  signature {
    op value    : TradingSystem ServiceOffer Node -> ServiceOfferIdentifier
    op state    : TradingSystem ServiceOffer Node -> TradingSystem
    op pre-cond : TradingSystem ServiceOffer Node -> Bool
  }
  axioms {
    var S : TradingSystem     var O : ServiceOffer     var N : Node

  ceq value(S,O,N) = (O).offer-id if pre-cond(S,O,N) .
  ceq value(S,O,N) = void         if not pre-cond(S,O,N) .

  ceq state(S,O,N) = [offers=(add((S).offers,O)), nodes=((S).nodes),
                      partition=(add((S).partition,O,N)), edges=((S).edges),
                      edge-properties=((S).edge-properties)]   if pre-cond(S,O,N) .
  ceq state(S,O,N) = S                                  if not pre-cond(S,O,N) .

  eq pre-cond(S,O,N) = ((N) is-member-of ((S).nodes))
                       and (new-id ((S).offers, (O).offer-id)) .
  }
}
```

Fig. 4 Module EXPORT-BEHAVIOR.

The fact that the properties can be divided into preconditions and postconditions is the basis for writing the CafeOBJ specification.

Since the *ExportOK* and other operation schemata not only return some specific values, but also leave changes in the state space, we could not model the operation in a purely functional manner. The module BASIC-STATE has functionalities whereby the operation can return values and update the state space at the same time. The function to represent a top-level operation will take the following form:

TradingSystem × InputArgs
    → ReturnValue × TradingSystem .

Actually, the sort ValueState is defined as a tuple of ReturnValue and TradingSystem.

The module EXPORT defines the top-level operation export, which uses two auxiliary functions and returns a new ValueState value; value computes a return value and state gives a new state space value.

```
mod! EXPORT {
  protecting (BASIC-STATE)
  protecting (EXPORT-BEHAVIOR)
  signature {
    op export :
      TradingSystem ServiceOffer Node
      -> ValueState }
  axioms {
    var S : TradingSystem
    var O : ServiceOffer     var N : Node
  eq export(S,O,N)
  = new-state(value(S,O,N), state(S,O,N)) .
  }
}
```

The module EXPORT-BEHAVIOR in **Fig. 4** defines behavior for the two functions. Since the original Z specification for the *Export* schema handles an exceptional case as well as a normal one, the behavior of the two functions can be defined in terms of a set of conditional equations where the condition corresponds to the precondition of the *ExportOK* schema.

For example, the member offers is updated to be add((S).offers,O) which is a CafeOBJ representation of the following Z specification fragment in the *ExportOK* schema:

$$offers' = offers \cup \{new\_offer?\}.$$

Further, in order for the above specification to be executable, add((S).offers,O) should be evaluated to have a *natural* normal form of the specified sort ServiceOffers. This requires that the module SERVICE-OFFER-S defining the sort ServiceOffers should provide executable data structures (see Section 4.1.2).

4.1.5 Relation Schema as Function

The Z specification includes a schema which introduces a relation on some particular set(s). The relation *is_subtype_of* (**Fig. 5**) is one such example which defines the subtype relationship between two *ServiceType* instances.

In order to make the relation *is_subtype_of* executable, we model it as a function to ensure that the two arguments of the relation *is_subtype_of* satisfy the subtype relationship (**Fig. 6**). According to the Z specification, the relation is further decomposed into *is_sig_subtype_of*, $\subseteq$ and the $\forall$ part which also checks the two conditions by using

$$\begin{array}{|l}
\_\ \textit{is\_subtype\_of}\ \_ : \textit{ServiceType} \leftrightarrow \textit{ServiceType} \\
\hline
\forall\, a, b : \textit{ServiceType} \bullet b\ \textit{is\_subtype\_of}\ a \Leftrightarrow \\
\quad b.\textit{signature}\ \textit{is\_sig\_subtype\_of}\ a.\textit{signature} \\
\quad \wedge\ \mathrm{dom}\, a.\textit{prop\_defs} \subseteq \mathrm{dom}\, b.\textit{prop\_defs} \\
\quad \wedge\ (\forall\, n : \mathrm{dom}\, a.\textit{prop\_defs} \bullet \\
\qquad \textit{first}(a.\textit{prop\_defs}\ n)\ \textit{is\_value\_supertype\_of}\ \textit{first}(b.\textit{prop\_defs}\ n) \wedge \\
\qquad \textit{second}(a.\textit{prop\_defs}\ n)\ \textit{is\_mode\_supertype\_of}\ \textit{second}(b.\textit{prop\_defs}\ n))
\end{array}$$

Fig. 5 Schema *is_subtype_of*.

```
mod! SUBTYPING-RULE {
  protecting (SERVICE-TYPE)
  protecting (SIGNATURE-SUBTYPING)
  protecting (VALUE-MODE-SUPERTYPE-FLATTEN[PRED2SUBTYPING]
             *{ op andalso -> check-vm })
  signature {
    op _is-subtype-of_ : ServiceType ServiceType -> Bool
  }
  axioms {
    vars S1 S2 : ServiceType

    eq (S1) is-subtype-of (S2)
    =     ((S1).signature) is-sig-subtype-of ((S2).signature)
      and ((names((S1).prop-defs)) is-subsumed-by (names((S2).prop-defs)))
      and check-vm(names((S1).prop-defs), (S1).prop-defs, (S2).prop-defs) .
  }
}
```

Fig. 6 Module SUBTYPING-RULE.

```
var N : Name   vars W1 W2 : ValueMode  vars P1 P2 : PropertyDefinitions

eq pred-vm (N,P1,P2)  =  pvm(lookup(P1,N), lookup(P2,N)) .
eq pvm(W1,W2)  = (get-value-type(W1) is-value-supertype-of get-value-type(W2))
                 and (get-mode(W1) is-mode-supertype-of get-mode(W2)) .
```

Fig. 7 Predicate pred-vm.

is_value_supertype_of and *is_mode_supertype_of*.

Since the last condition concerns $\forall n$: dom *a.prop_defs*, we will model it as a function with n as its input parameter. More concretely, a new function check-vm is defined to have three arguments, the first runs through n in dom *a.prop_defs*, and the second and the third are the *prop_defs* components of the *ServiceType* instances. Since enumeration over n in dom *a.prop_defs* requires some auxiliary functions in CafeOBJ, we factor the definition of check-vm into another module VALUE-MODE-SUPERTYPE-FLATTEN that in turn uses a parameterized module to simulate the higher-order specification style. The following fragment illustrates the body of the function check-vm.

```
var N : Name  var L : Names
vars P1 P2 : PropertyDefinitions

eq andalso(nil,P1,P2) = true .
eq andalso((N L),P1,P2)
=  if p(N,P1,P2) then andalso(L,P1,P2)
   else false fi .
```

The function andalso is basically a procedural implementation of the following Z specification fragment.

$$\forall n\text{: dom } a.\textit{prop\_defs} \bullet p(n, ad(n), bd(n))$$
$$\Rightarrow\ \wedge_{(n \in \mathrm{dom}\ a.\textit{prop\_defs})}\ p(n, ad(n), bd(n))$$

The predicate p (or p) is actually given by pred-vm in **Fig. 7**.

4.2 Example Execution Trace

Below we show an example reduction for the case of EXPORT. After loading all the necessary CafeOBJ specifications, we can *execute* the specification with appropriate input terms (test data)☆.

```
TEST> let a1 = export((t), new-s, node) .
TEST> let a2 = export((t), new-s, node2) .
TEST> red 1*(a1) .
oid3 : ServiceOfferIdentifier

TEST> red trading-system-axiom(2*(a1)) .
```

☆ We used cafeobj (1.4b5), which can be obtained from the following URL: http://www.ldl.jaist.ac.jp/cafeobj/index.html.en

SearchOK

Ξ ***TradingSystem***
Ξ ***TradingSystemConstraints***
SearchRequest?
starting_point? : ***Node***
search_result! : $\mathbf{P}$ ***ServiceOffer***

starting_point? $\in$ ***nodes***
partition$(\!|$ ***search_restult!*** $|\!)$ $\subseteq$ $\{x : Node \mid (starting\_point?, x) \in edges^{+}\}$
search_result! $\subseteq$ ***importer_matching?*** $\cap$ ***trader_matching*** $\cap$ ***importer_scope?*** $\cap$ ***trader_scope***

Fig. 8 Schema *SearchOK.*

```
true : Bool

TEST> red 1*(a2) .
void : ReturnValue

TEST> red trading-system-axiom(2*(a2)) .
true : Bool
```

The construct `let` temporarily binds its right-hand side term to the left-hand side identifier. The label `t` refers to a `TradingSystem` value. The label `new-s` is a `ServiceOffer` value defined as a term similar to record structure in the module `SERVICE-OFFER`. The labels `node` and `node2` are `Node` values defined in the module `NODE`. Since the value of `export` is a tuple (see Section 4.1.4), we can obtain the return value of the `export` function by `1*(a1)` and the resultant new `TradingSystem` state value by `2*(a1)` where `1*` and `2*` access the first and second element of the tuple respectively[14].

In the above session, `a1` is a case of success in *Export* operation while `a2` corresponds to a case where precondition of *Export* is violated and thus leaves no changes in *TradingSystem.* For the first case, the return value is a new `ServiceOfferIdentifier` value `oid3`, which is generated by `export`. The second case returns `void` which means that no significant value is returned. In both cases, `trading-system-axiom` returns `true` as expected (see Section 4.1.3).

Last, the CAFE environment also provides a compiler that translates terms and rewriting rules into virtual machine codes in order to realize fast execution[13]. Typically, we can obtain 3 to 10 times performance gain by using the compiler.

4.3 Search and Select

The Z specification of the standard document defines *Search* and *Select* for importing. The basic idea is first to use *Search* for collecting all the service offers that match both the importer's request and the conditions that the trader has, and second to invoke *Select* for sorting the offers according to the importer's preference. As shown below and in **Fig. 8**, the original Z specification is very declarative and hard to understand its operational meaning at a glance.

MatchingCriteria == $\mathbb{P}$ *ServiceOffer*
ScopeCriteria == $\mathbb{P}$ *ServiceOffer*

TradingSystemConstraints

trader_matching : ***MatchingCriteria***
trader_scope : ***ScopeCriteria***
$\vdots$

SearchOK is meant to return an appropriate set of *ServiceOffer*s bound to the variable *search_result!*. The appropriateness is expressed in terms of the logical condition given to the variable; that is, the returned *ServiceOffer*s is a subset of intersection of the four *ServiceOffer* sets. Each set is a subset of the managed *ServiceOffer*s that matches a particular partitioning condition. For example *importer_matching?* is for those that match the condition in the importer request, while *trader_matching* refers to those that reflect the condition imposed by the trader. Thus, taking the intersection produces a set of *ServiceOffer*s that match all the conditions.

However, since all four sets are defined as a value of $\mathbb{P}$ *ServiceOffer* and are not mentioned further, it is not easy to grasp what the specification really means. This is partly because the modeling method that *ServiceOffer*s are partitioned *a priori* has a large gap with the specifications for the computational viewpoint: the viewpoint uses a constraint language to explicitly specify such conditions[1]. In sum, the original Z specification is abstract and thus further design decision is necessary to have an executable specification. Since introducing such design decisions is beyond the scope of this paper, our decision was not to provide executable

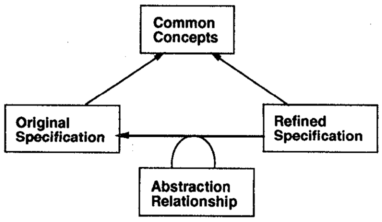


Fig. 9 Refinement.

specifications.

Actually we have studied two styles of CafeOBJ specifications for this part. Our decision for the first approach is only to provide signatures (declarations of symbols) and minimum axioms saying that some relationships exist between some of the symbols. In a word, the specification is a direct syntactical transcription of the Z specification. However, the CafeOBJ specification can be mechanically checked in view of syntax and sort (type), while the Z specification is hard to mechanically analyze. Namely, the specification fragment togther with the rest of the CafeOBJ descriptions can be shown consistent in view of sort. Our second approach is to study how CAFE/CafeOBJ provides mechanical support for specification refinement, which we will discuss below.

As the CafeOBJ specifications, we define two versions for *SearchOK* as shown in **Fig. 9**. One is what is called the original specification, and the other is a refined one that employs modeling with a hypothetical retrieval language. The retrieval language can be regarded as an abstraction of the constraint language of the computational viewpoint[1]). In addition, we introduce an abstraction relationship which maps elements in the refined specification to elements in the original if possible. The idea has commonly been used in verifying refinement of the Z specifications[22]).

Here, we will show that the search result expressed as `search(T,X,S)` is a subset of the intersection of the two predefined sets☆. The relationship was shown as a part of the original specification.

```
var T : TradingSystem
var S : SearchRequest
var X : TradingSystemConstraints

eq search(T,X,S) is-subset-of
    ((X).trader-matching
     cap (S).importer-matching) = true .
```

The refined specification below indicates how `search` is obtained in terms of executing the search procedure which employs the hypothetical retrieval language. The condition that the resulting search offers should meet the requirements of both *trader-matching* and *importer-matching* is expressed as a conjunction of the two conditions.

```
var T : TradingSystem
var S : BSearchRequest
var X : BTradingSystemConstraints

eq search(T,X,S)
= exec((T).offers, (S).bimporter-type,
      ((X).btrader-matching
       and (S).bimporter-matching)) .
```

We also have a set of abstraction functions which map elements in the refined specification to their counterparts in the original specification, though the detailed equations are omitted.

```
op abs : BSearchRequest -> SearchRequest
op abs : BTradingSystemConstraints
          -> TradingSystemConstraints
```

With appropriate lemmas, we can mechanically verify the relationship for the terms in the refined specification: the search result is a subset of the intersection of the two specified sets.

```
CafeOBJ> start abs(search(T*,XB*,SB*))
 is-subset-of ((XA*).trader-matching
           cap (SA*).importer-matching) .
CafeOBJ> apply red at term .
result true : Bool
```

The identifiers ended with `*` are all constants of appropriate sort which act as universally quantified variables☆☆. Actually, `T*` denotes a `TraderSystem`. `SB*` and `XB*` are `BSearchRequest` and `BTradingSystemConstraint` respectively, while `SA*` and `XA*` are the counterparts of the *original* specification in Fig. 9.

In addition to those relating to some general properties of set, lemmas include definitions for the hypothetical retrieval language. In particular, the equation for `exec` is the basis of mechanical checking above because it reflects the compositionality of retrieval conditions.

```
mod* B-RETRIEVAL-LANGUAGE {
  [ RetrievalLanguage ]
  protecting (A-SERVICE-OFFER-S)
  signature {
```

☆ For simplicity, we dropped the condition on the scoping (*importer_scope*? ∩ *trader_scope*) of the Z specification.

☆☆ Using a constant as a representative value for an universally quantified variable follows the Theorem of Constants[15]).

Table 2 Some metrics.

	Category	Z Notation	CafeOBJ	
			total	direct
1	*Basic Concepts*	13	23	13
2	*State Schema*	9	43	15
3	*Main API*	31	26	10
4	*Library*	–	5	0
5	(total)	53	97	38

```
  op _and_ : RetrievalLanguage
             RetrievalLanguage
               -> RetrievalLanguage [comm]
  op exec : ServiceOffers ServiceType
              RetrievalLanguage
                -> ServiceOffers
}
axioms {
  var P : ServiceOffers
  var T : ServiceType
  var O : ServiceOffer
  vars L L1 L2 : RetrievalLanguage

ceq ((O).service-type) is-subtype-of T
=   true  if member(O,exec(P,T,L)) .
eq exec(P,T,(L1 and L2))
= exec(P,T,L1) cap exec(P,T,L2)  .
  }
}
```

4.4 Summary and Discussion

The CafeOBJ specification we have written consists of 97 modules, and the number of equations is 290[☆]. **Table 2** summarizes some metrics of the specification. The column *Z Notation* shows the number of Z specification components (the standard document[1]) while the column *CafeOBJ* shows the number of CafeOBJ modules. The number of CafeOBJ modules that have direct correspondence with the Z counterpart is also shown. It depicts how much of the CafeOBJ modules are traceable from the Z specification, and how much are necessary for obtaining executable specifications.

The raw *Basic Concepts* is for primitive concepts defined by using given names, free types, and abbreviation definition (==) in the Z specifications. The raw *State Schema* is for structured data, relationship, and the main state schema *TradingSystem* defined by using schema in the Z specifications. The raw *Main API* is for operations visible outside; actually operation schema. The raw *Library* is for the common library modules that implement executable specifications and thus only for CafeOBJ exist.

In Table 2, we can see about 40% of the CafeOBJ modules are directly traceable from the Z counterparts. Primary reason that the CafeOBJ specification is larger than the original Z specification follows from the fact that we use property-oriented specification style for CafeOBJ while model-oriented style in the Z notation. In the property-oriented style, we have to provide all the necessary definitions[24]. On the contrary, we can assume or *use* predefined set of primitives in the Z notation. We can say that some of the CafeOBJ modules constitute the "model" that provides appropriate definitions which are considered to be equivalent to the predefined definitions of the Z notation, although the "model" is quite different since our aim is to have executablity. Further, detailed descriptions are necessary to have executable specifications (see Section 4.1.2). This also makes the CafeOBJ specification large.

As the most of *Basic Concepts* is just introducing set in the Z notation or sort in the CafeOBJ, the number of modules that has direct correspondence is good. Others are modules necessary for calculating membership of data in a certain type. In the Z notation, membership is very easy to state. Since a type of value is defined in terms of a set in the Z notation, such membership is expressed in terms of $\in$ predicate.

$\mathit{ValueType} == \mathbb{P}\ \mathit{Value}$

$a\colon \mathit{ValueType} \wedge b\colon \mathit{Value} \wedge b \in a$

For the CafeOBJ specification, both type and data are ordinary terms belonging to certain sorts, the CafeOBJ version requires some auxiliary functions to calculate the membership.

Most of *hacking* for CafeOBJ to have executability is in *State Schema* because functions and/or relationships are fallen in this category and thus lots of auxiliary modules are necessary. For *Main API*, the Z specification has three or five schemata for each operation, while in the CafeOBJ specification we write one module for one operation and other auxiliary modules as

☆ For the specification of *Search* and *Select*, the number includes the first approach briefly mentioned in Section 4.3.

necessary. The traceability of the top-level operations is clear.

Last, in Section 4.3, we have seen that CAFE/CafeOBJ provides adequate supports for mechanical verification of specification refinement. The verification activity usually involves using lemmas, which often require further proof sessions. As it is always the case, finding appropriate lemmas is a difficult task that human should take care of. However, we think that studying one specification from various aspects helps us understand its functionality. Writing specifications at two different abstraction levels and establishing refinement relationships between them is one such approach when executability is incompatible.

5. Conclusion

We presented CafeOBJ specifications of the information viewpoint of the ODP trader. Our experience has shown (1) that algebraic specification technique is useful to describe characteristrics of the ODP trader and (2) that the resultant specifications contribute to helping us understand the functionality of the ODP trader at an appropriate abstract level because specifications in CafeOBJ are executable. As shown in Section 4.2, we can study the ODP trader functionality by test executions. In addition, as discussed in Section 4.3, writing specifications at different abstraction levels and establishing their relationships contributes to help us understand the functionalities.

Although our method seems to provide a specific framework only applicable to the specifications of the information viewpoint of the ODP trading function, we think our specification itself is valuable in the following sense. First, since the ODP trading function is an important specification widely used, we expect our specification to be consulted often. Second, since the presentation is based on actual examples, the present paper shows a concrete guideline on how we write CafeOBJ specifications for a large distributed object-oriented software.

Last, since the CAFE environment has a facility to distribute HTML documents in which CafeOBJ specifications coexist with other informal explanations (even with graphics), it is considered to provide means for a new style of circulating standard documents. The standard (such as the ODP trader) will take a form of HTML document that has both informal explanations and formal specifications. The formal part would be linked to a backend reduction engine (CafeOBJ in this case) to execute mechanical analysis. Applying the idea to the ODP trader specification is one of future directions of the present work.

Acknowledgments This work was supported in part by the Advanced Software Enrichment Project of the Information-Technology Promotion Agency, Japan (IPA).

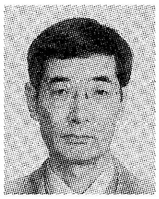
References

1) ITU-T Rec. X.950-1: Information Technology – Open Distributed Processing–Trading Function – Part 1: Specification (1997).
2) OMG: CORBAservices, Trading Object Service Specification (1997).
3) Bearman, M.: Tutorial on ODP Trading Function, University of Canberra (1997).
4) Bernardeschi, C., Dustzadhe, J., Fanttechi, A., Najm, E., Nimour, A. and Olsen, F.: Transformation and Consistent Semantics for ODP Viewpoints, *Proc. FMOODS '97* (1997).
5) Bowman, H., Derrick, J., Linington, P. and Steen, M.W.A.: FDTs for ODP, Computer Standards and Interfaces (17), pp.457–479 (1995).
6) Bowman, H, Boiten, E.A., Derrick, J. and Steen, M.W.A.: Viewpoint Consistency in ODP, A General Interpretation, *Proc. FMOODS '96* (1996).
7) Diaconescu, R. and Futatsugi, K.: Logical Semantics for CafeOBJ, JAIST Research Report IS-RR-96-22S (1996).
8) Diaconescu, R. and Futatsugi, K.: *The CafeOBJ Report*, World Scientific (1998).
9) Dustzadeh, J. and Najm, E.: Consistent Semantics for ODP Information and Computational Models, *Proc. FORTE/PSTV '97* (1997).
10) Fischbeck, N., Fischer, J., Holz, E., Lowis, M., Kath, O. and Schroder, R.: Improving the Development and Validation of Viewpoint Specifications, *Proc. FMOODS '97* (1997).
11) Fischer, J., Prinz, A. and Vogel, A.: Different FDT's Confronted with Different ODP-Viewpoints of the Trader, *Proc. FME '93*, pp.332–350 (1993).
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) Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K. and Jouannaud, J.-P.: Introducing OBJ, SRI-CSL-92-03 (1992).
15) Goguen, J. and Malcolm, G.: *Algebraic Se-*

mantics of Imperative Programs, MIT Press (1996).
16) Goguen, J. and Malcolm, G.: A Hidden Agenda, UCSD CS97-538 (1997).
17) Lecero, G.F. and Quemada, J.: Specifying the ODP trader in E-LOTOS, *Proc. FORTE/PSTV '97* (1997).
18) Meseguer, J.: A Logical Theory of Concurrent Objects and its Realization in the Maude Language, *Research Directions in Concurrent Object-Oriented Programming*, Agha, Wegner and Yonezawa (Eds.), MIT Press (1993).
19) Mowbray, T.J. and Zahavi, R.: *The Essential CORBA*, John Wiley & Sons (1995).
20) Nakajima, S. and Futatsugi, K.: A CafeOBJ Specification of the ODP Trader (in Japanese), *Computer Software*, to appear (1999).
21) Raymond, K.: Reference Model of Open Distributed Processing (RM-ODP): Introduction, *Proc. ICODP '95* (1995).
22) Spivey, J.: *The Z Notation* (2nd edition), Prentice Hall (1992).
23) Vogel, A. and Duddy, K.: *Java Programming with CORBA*, Wiley (1997).
24) Wing, J.: A Specifier's Introduction to Formal Methods, *IEEE Computer*, pp.8–24 (1990).
25) Yatsu, H. and Futatsugi, K.: Verification of Z Specifications using Algebraic Specifications (in Japanese), *Computer Software*, Vol.13, No.6, pp.26–42 (1996).

(Received July 17, 1998)
(Accepted December 7, 1998)

Shin Nakajima is a principal researcher with C&C Media Research Laboratories at the NEC Corporation, Japan. He received B.A. and M.Sc. degrees in physics from the University of Tokyo. His research interests include distributed software engineering, algebraic specifications, and meta-level architecture. He is a visiting lecturer at Tokyo Metropolitan University.



Kokichi Futatsugi is a Professor of Graduate School of Information Science, Japan Advanced Institute of Science and Technology (JAIST). He worked for ETL, MITI from 1975 to 1993 and was assigned to a Chief Senior Researcher of ETL in 1992. He got a full professorship at JAIST in 1993. His research interest includes algebraic formal methods and their application to software engineering, and language design as a foundation for system design.