25

Algebraic Specification of Distributed Systems based on Concurrent Object-Oriented Modeling

Shusaku Iida, Kokichi Futatsugi and Takuo Watanabe
Graduate School of Information Science,
Japan Advanced Institute of Science and Technology (JAIST)
15 Asahidai, Tatsunokuchi, Ishikawa, 923-12, JAPAN
Phone: +81-761-51-1255, Fax: +81-761-51-1149
E-Mail: s_iida@jaist.ac.jp, kokichi@jaist.ac.jp and takuo@jaist.ac.jp

Abstract

We propose a new executable algebraic specification method for object-oriented concurrent and distributed systems. We formalize a concurrent object-oriented model that can explicitly handle communication networks. In this model, a system is described as a collection of *primitive objects* and *network objects*. We use the algebraic specification language CafeOBJ[Futatsugi and Sawada 1995] [Sawada and Futatsugi 1995] for describing specifications. Since specifications using our method can be executed, the CafeOBJ processor aids semi-automatic verifications. We illustrate some actual verifications via an example.

Keywords

Formal Method, Algebraic Specifications, Concurrent Object-Oriented Model, Concurrent and Distributed Systems

1 INTRODUCTION

Concurrent and distributed systems are the basis for many powerful computing environments. But it is more difficult to grasp the properties of these systems than the properties of sequential systems. Hence, the ability of verifications in the first stages of development processes becomes more important. The purpose of this research is to propose a method for verifying formal specifications of concurrent and distributed systems which is based on algebraic specification techniques.

Dividing large specifications into several parts such that each of them can be easily understood seems to be almost the only way in which we can reduce their complexity. We adopt a concurrent object-oriented model for this division, because objects have the ability of naturally representing the processing elements of concurrent and distributed systems. Also, concurrent object-oriented

E. Najm et al. (eds.), *Formal Methods for Open Object-based Distributed Systems*

models can simulate the behaviour of concurrent and distributed systems having several running paths.

There is a formalization of the concurrent object-oriented model based on rewriting logic in the algebraic specification language Maude[Meseguer 1990]. Rewriting logic has the power of naturally representing the behaviour of concurrent objects. In this paper, we formalize a concurrent object-oriented model that supports various kinds of communication networks using CafeOBJ[Futatsugi and Sawada 1995][Sawada and Futatsugi 1995] which is based on rewriting logic. CafeOBJ syntax is very similar to Maude syntax. Examples are presented in order to show the effectiveness of our formalism. We especially focused on how to automate verification processes.

2 A FORMALIZATION OF CONCURRENT OBJECT-ORIENTED MODEL

In concurrent object-oriented models, systems are represented as objects and messages. These models, like the Actor model [Agha 1986], can be represented in as Figure 1. Meseguer showed how to represent these models in rewriting logic. We first briefly explain rewriting logic and then explain how Meseguer formalizes concurrent objects in rewriting logic.

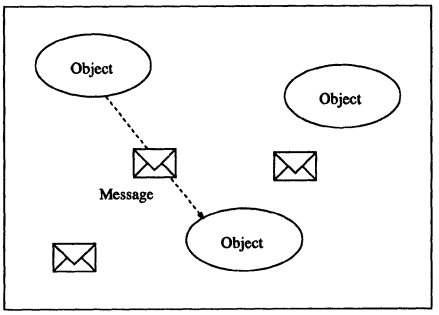

Figure 1 Concurrent object-oriented model

2.1 Rewriting logic

In order to formalize concurrent objects in an algebraic specification language, we need rewriting logic[Meseguer 1990][Meseguer 1993] rather than equational logic. A theory in equational logic is called equational theory, and it is defined as tuple $\langle \Sigma, E \rangle$, where Σ is a signature and E is a set of equations. A theory in rewriting logic is called rewrite theory, and it is defined as $\langle \Sigma, E, L, R \rangle$, where L is a set of labels for rules and R is a set of rules. In rewriting logic there are four deduction rules, (1)reflexivity, (2)congruence, (3)replacement, (4)transitivity.

When we write specifications in an algebraic specification language based on equational

logic, such as OBJ3 [Goguen, Winkler, Meseguer and Futatsugi 1993], the executable specifications must satisfy the confluence and terminating properties. There are many kinds of nondeterministic behaviour in concurrent object-oriented model, and their behaviour does not satisfying these two properties. In rewriting logic we can use rules for describing this kind of behaviour.

In Maude, the transitions of objects is done in parallel because Maude is based on concurrent rewriting. But CafeOBJ is based on sequential term rewriting. Hence, in our formalism, concurrent behaviour of objects will be simulated with sequential term rewriting. We think this kind of simulation is a necessary step in the verification process.

2.2 Meseguer's model

Meseguer showed how to deal with concurrent object-oriented models in Maude[Meseguer 1990][Meseguer 1993]. In his method, the state of a concurrent object-oriented system is represented as a "configuration" which is a multi-set of objects and messages. Computation is done by transitions between configurations using the rules of rewriting logic. Concurrent rewriting is used as a deduction in rewriting logic. A configuration is defined in the following way:

$m, n \in Nat$
$O \in Object$
$M \in Message$

$Configuration : [O_1 ... O_m \; M_1 ... M_n]^*$

Transition rules are defined over this configuration. Generally, when objects and corresponding messages are in configuration then the messages disappear, the state and the class of the objects may change, all other objects vanish, maybe several new objects and messages are created.

2.3 Supporting communication networks

Concurrent object-oriented models seem to suit the process of describing concurrent and distributed systems. But in many cases, it is convenient to directly use more functional, highly abstracted communication networks. In Meseguer's formalization it is possible to have communication networks using objects. But since wo are interested only in concurrent and distributed systems, we can consider a model that is more suitable for describing them. Communication networks are necessary infrastructure for concurrent and distributed systems. So we propose a model like Figure 2.

In this model, there are two types of objects: *primitive objects* and *network objects*. Processing elements of concurrent and distributed systems are represented as *primitive objects* and communication networks are represented as *network object*. *Primitive objects* can communicate only with *network objects*, and these communications are all done in a synchronous way. *Primitive objects* behave actively and *network objects* behave passively. So only *primitive objects* decide

*$[e_1 ... e_n]$ denotes a multi-set

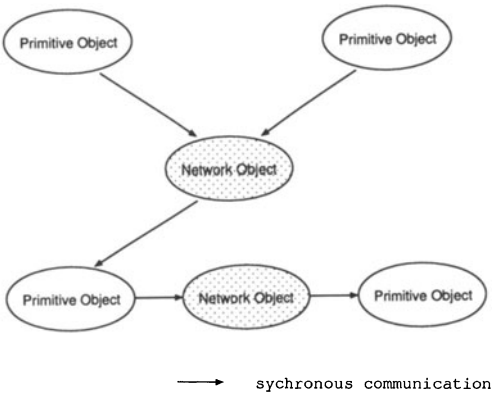

Figure 2 Our model

when they receive and put a message from *network objects*. In Meseguer's model, there are two kinds of communication mechanisms: asynchronous message passing and synchronous communication. In our model, there is only one communication mechanism. In our model, asynchronous message passing can be represented as a *network object* that has a multi-set as an attribute. This model has following properties:

- we assume no shared global state;
- an object can deal with only one message at the same time;
- each object has an unique identifier;
- when an object receives a message then it changes its own state and create nothing or several objects and messages;
- the state of an object changes only when receiving a message;
- *primitive objects* are communicate only with network objects using synchronous communication; and
- *network objects* can receive any messages from the objects connected to it;

In this model we only have objects, so the configuration is a set consists of *primitive objects* and *network objects*. The following is the definition of the configuration.

$m, n \in Nat$
$O \in Primitive\ objects$
$N \in Network\ objects$

$Configuration : \{ O_1 ... O_m\ N_1 ... N_n \}$

2.4 Using CafeOBJ

In this section we are going to explain the whole framework of our formalization using CafeOBJ. First we give a brief explanation of CafeOBJ. CafeOBJ is an algebraic specification language and it is a member of OBJ[Goguen, Winkler, Meseguer and Futatsugi 1993] [Futatsugi, Goguen and Jouannaud and Meseguer 1985] language family. It is based on rewriting logic and its type structure is based on order sorted algebra. Since CafeOBJ is executable, it can be used for prototyping and automatic theorem proving.

In CafeOBJ, **module** is the basic structure for describing specifications and this **module** consists of two parts: a signature part and an axiom part. The signature part is described within **signature{ }** and a axioms part is described within **axioms{}**. A signature part is for specifying a signature that consists of sort definitions and operator definitions. The axiom part is for specifying some equations and rewrite rules. The following is a specification of natural number:

```
module NAT {
  signature {
    [ Nat ]
    op 0 : -> Nat
    op s_ : Nat -> Nat
    op _+_ : Nat Nat -> Nat
    attr _+_ {assoc comm}
  }
  axioms {
    vars N N' : Nat
    eq s N + N' = s (N + N') .
    eq N + 0 = N .
  }
}
```

Sorts are defined with brackets and the subsorting relation defined by using **<**. The lines beginning with **op** are definitions of operators. An operator is defined by arguments and a return sort. Operators without arguments are constants. One can specify attributes of an operator using parenthesis after the definition of an operator. The lines beginning with **attr** also define the attributes for particular operators. For an attribute one can specify associativity, commutativity, idempotent law, identity, etc. In the axiom part, we define equations and rewrite rules. The definition of an equation begins with **eq** and the definition of a rewrite rule begins with **rule**, and **ceq** for a conditional equation and **crule** for a conditional rewrite rules. One may inherit other modules using **protecting()** or **extending()**. When we using **protecting()** the inherited modules are unchanged. We use **extending()** for extending inherited modules. We abide by the convention of CafeOBJ that variables should be represented in capitals and sorts should be represented by a word beginning with a capital letter. For operators we use a word beginning with a lower case.

Firstly, we define the structure of an object. An object has an unique identifier (OId) and is made from a class. Any class has an identifier (CId). OId is a pair consisting of a class identifier and a natural number. An object (OBJ) is pair consisting of OId and the attributes of the object. Object identifiers and objects are defined in CafeOBJ in the following way:

```
module OID {
  protecting(NAT)
  signature {
    [ OId CId < Identifier ]
    op nilOId : -> OId
    op <_,_> : CId Nat -> OId
  }
}

module OBJ {
  protecting(OID)
  signature {
    [ Obj Attr AId AValue ]
    op nullAttr : -> Attr
    op _=_ : AId AValue -> Attr { prec 6 }
    op _,_ : Attr Attr -> Attr { assoc comm id: nullAttr }
    op [_|_] : OId Attr -> Obj
  }
}
```

In this definition, if **AId** and **AValue** are properly defined then the following term is recognized as **Obj**.

```
[ < apple , 1 > | color = red , weight = 200 ]
```

A message is a tuple consisting of three elements: OId (of the sender object), OId (of the destination object), and the content of the message. A message is defined in CafeOBJ in the following way:

```
module MSG {
  protecting(OBJ)
  signature {
    [ Tag MValue Content Msg ]
    op nullCont : -> Content
    op _=_ : Tag MValue -> Content { prec 6 }
    op _,_ : Content Content -> Content { assoc comm id: nullCont }
    op <_|_|_> : OId OId Content -> Msg
  }
}
```

Configuration is a set consisting of objects.

```
module OBJ-SET {
  pr (SET [ X <= view to OBJ
            { sort Elt -> Obj } ])
}

module CONFIGURATION {
  protecting(OBJ)
```

```
  protecting(MSG)
  protecting(OBJ-SET)
  signature {
    [ Config ]
    op {_} : Set -> Config
  }
}
```

All objects are made of class definitions. Class definitions consist of attribute definitions and rewrite rules definitions (Figure 3).

```
module FOO-CLASS {

  *** attribute definitions

      [ Nat < AValue ]
      op counter : -> AId

  *** transition rules

      rule
          trans O N => O' N'

}
```

Figure 3 Class definitions

3 EXAMPLE

In this section we are going to explain how to build specifications in our formalism. We also show the effectiveness of our new definition of a configuration for the case of the verification processes.

3.1 Alternating Bit Protocol

The alternating bit protocol (ABP) is a protocol that realizes an ideal network using two FIFO but unreliable communication networks. The ideal network (IN) is a FIFO and reliable communication network. In concurrent object-oriented model ABP can be represented as in Figure 4.

The sender object has a queue called Sbuf and a boolean variable called Sflag. The receiver object also has a queue called Rbuf and a boolean variable called Rflag. When the system is in initial state, Sbuf and Rbuf are empty and Sflag and Rflag have different boolean values. When the sender object receives the message to be sent, the content of it is buffered in Sbuf. If there is an element in Sbuf, the sender object makes a pair of the first (oldest) element of Sbuf and a value of Sflag (tag) and sends it to the receiver. The sender object constantly sends this

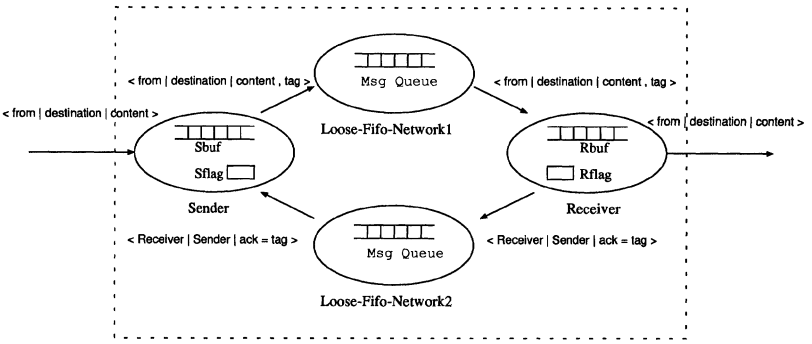

Figure 4 Alternating Bit Protocol

Table 1 behaviour table for the sender object

message	precondition	state	new messages
any messages		sbuf = sbuf ++ mes	
	sbuf != []		create-mes(head(sbuf), sflag)
ack(tag)	tag == sflag	sbuf = tail(sbuf), sflag = !sflag	
ack(tag)	tag != sflag		

message to the receiver object within a certain interval. When the receiver object receives the message from the sender object and if the tag of the message and the value of Rflag are not equal, then the content of the message is buffered in Rbuf and toggles Rflag, otherwise just throws the message away. The receiver object constantly sends the value of Rflag to the sender object for an acknowledgment(ack) within a certain interval. When the sender object receives an ack from the receiver object and if it is equal to Sflag, then it drops the first (oldest) element from Sbuf and toggles Sflag, otherwise just throws the message away. In Figure 4, the inside area of the rectangle can be regarded as one object. This object can be regarded as a network object.

The behaviour of the Sender object and the Receiver object is shown in Table 1 and Table 2. We call these tables "behaviour tables". The first column of this behaviour table represent the message which the object receives, the second column contains the preconditions for the rewrite rules, the third column contains the states after receiving the message (like postcondition), and the last column shows the newly created messages. The function "create-mes()" create new message from its arguments, and "ack(tag)" replesents a message carrying the value "tag".

We can directly derive the rewrite rules of class specification from this behaviour table. The first row of table 1 shows the behaviour of receiving a message from some other object connected

Table 2 behaviour table for the receiver object

message	precondition	state	new messages
			ack(rflag)
mes(tag)	tag != rflag	rbuf = rbuf ++ mes, rflag = !rflag	
mes(tag)	tag == rflag		

to the sender object. Because network objects behave passively, this behaviour is not suitable for the sender object. The specification of the second and the third row of table 1 can be described in CafeOBJ in the following way:

```
rule trans
   { [ < sender , 0 > | (Sbuf = ((< O | O' | C >) Q)) , (Sflag = F) ]
     [ < fifo , 0 > | (Fifo = Q') ] null }
=>
   { [ < sender , 0 > | (Sbuf = ((< O | O' | C >) Q)) , (Sflag = F) ]
     [ < fifo , 0 > | (Fifo = (Q' < O | O' | C , (Flag = F) >)) ] OBJS } .

crule trans
   { [ < sender , 0 > | (Sbuf = Q) , (Sflag = F) ]
     [ < fifo , 1 > | (Fifo = (< < receiver , 0 > |
           < sender , 0 > | (Cmd = Ack) , (Flag = F') > Q')) ] OBJS }
=>
   { [ < sender , 0 > | (Sbuf = (tail(Q))) , (Sflag = F') ]
     [ < fifo , 1 > | (Fifo = Q') ] OBJS }
   :if F == F' .
```

The operator **trans** takes a configuration and returns an other configuration corresponding to one step transition of a configuration.

4 VERIFICATION

The advantage of using our technique instead of others is that verifications can be done automatically. We apply the state mapping technique [Lynch and Tuttle 1987, Merrit 1989, Weihl 1993] to the concurrent object-oriented model using communication networks. In order to use the technique explained below, we assume that objects are not dynamically created or disappear. We first explain the state mapping technique and non-deterministic behaviour of concurrent and distributed systems, and then explain our verification method using CafeOBJ.

4.1 State mapping

State mapping is used for proofing the equivalence between two automata or state machine. In order to prove the two systems are equivalent, we must get an abstraction function explaining the relationship between them.

We again use ABP to illustrate our method. ABP can be seen as an implementation of an ideal network (IN). Ideal network is a FIFO and reliable communication network. IN is modeled as Figure 5.

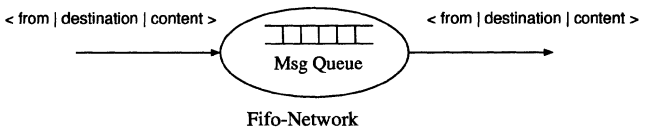

Figure 5 Ideal newtork

We apply the state mapping technique for showing that ABP is a correct implementation of IN. Firstly, we must define an abstraction function **abs-func** that maps ABP's state (configuration) to IN's state (Msg Queue). We also define **get-state-IN** that takes IN's configuration and returns the "Msg Queue" of IN. These functions are defined in following way:

```
eq get-stat-IN { [ < 'Fifo , 0 > | (Fifo = Q) ] OBJS } = Q .

cq abs-func
  ({ [ < 'Sender , 0 > | (Sbuf = SQ) , (Sflag = F) ]
     [ < 'Receiver , 0 > | (Rbuf = RQ) , (Rflag = F') ] OBJS })
= (RQ SQ) :if F =/= F' .

cq abs-func
  ({ [ < 'Sender , 0 > | (Sbuf = SQ) , (Sflag = F) ]
     [ < 'Receiver , 0 > | (Rbuf = RQ) , (Rflag = F') ] OBJS })
= (RQ (tail(SQ))) if F == F' .
```

Then we need a notion of "external message" and "internal message" and "same externally messages". Messages are external if they construct an interface of the system and they are internal if not. The sequence of messages M and M' are called externally the same if they are the same after removing all internal messages from both of them.

ABP is a correct implementation if:

1. the abstract function **abs-func** maps the initial state of ABP to the initial state of IN;
2. for all messages M_{ABP} accepted by ABP, there exists the same externally message sequence M_{IN} accepted by IN;
3. show if ABP's state **s** transforms to **s'** by M_{ABP} then M_{IN} transforms **abs-func(s)** to **abs-func(s')**; and
4. if there is no corresponding message for M_{ABP} then show **abs-func(s)** equals **abs-func(s')**.

The steps 3 and 4 are done by case analysis and (if needed) using invariants of the system. Usually, the number of cases for this case analysis become very large. In the case of ABP, there

are four messages: (1) messages from outside of ABP, (2) messages from the sender object to the receiver object, and (3) acknowledgement messages form the receiver object to the sender object, and (4) messages that send to outside of ABP. For the case (1), there are following cases: $\langle Sbuf = [\ ],\ Rbuf = [\ ]\rangle$, $\langle Sbuf = [\ ],\ Rbuf = [Queue]\rangle$, $\langle Sbuf = [Queue],\ Rbuf = [\ ]\rangle$, $\langle Sbuf = [Queue],\ Rbuf = [Queue]\rangle$. Each case has more combinations of $Sflag$ and $Rflag$, and also we must consider the value of a tag which added to the message from the sender object to the receiver object. So the total cases in ABP becomes about 100.

4.2 Non-deterministic behaviour

The process of the state mapping technique explained above can be automated by using CafeOBJ. But to do so, we must consider about non-deterministic behaviour. In concurrent and distributed systems there are two types of non-deterministic behaviour.

- global non-deterministic behaviour:
  - the order of messages that an object receives from the communication networks is non-deterministic and
  - objects making messages without receiving messages behave non-deterministically.
- local non-deterministic behaviour
  - when an object receives a message, two or more rewrite rules can be applied

When using CafeOBJ we must care of local non-deterministic behaviour and of the second case of global non-deterministic behaviour. In the case of specifications with local non-deterministic behaviour, we have to divide it, build several deterministic specifications, and test them all, because in CafeOBJ it is always decided which rule has to be used when more than one rule can be applied to a term. For the second case of global non-deterministic behaviour, one should remove the corresponding rewrite rules from the specification. The reason that we can remove these rewrite rules is that in the state mapping, our interest is only in local behaviour of objects so we don't have to care how messages are created.

4.3 Automatic verification using CafeOBJ

The following is a verification process:

1. build the specifications of ABP and IN using CafeOBJ and put them togerther into one specification;
2. do appropriate treatment for non-deterministic behaviour;
3. design the abstraction function;
4. make proof scores according to the state mapping technique;
5. execute the proof scores (first stage); and
6. check the system does not go into any of the cases returning false (second stage).

We call the step 5 as the first stage and we call thestep as the 6 second stage. The first stage can be done automatically by CafeOBJ, and then use its results in the second stage. Second stage

is done by humans. We already built the specification of ABP and IN so we begin with step 2. In ABP, there are two non-deterministic behaviour as following:

- the behaviour of unreliable networks (Loose-Fifo-Network)
- the sender object constantly sends messages to the receiver object
- the receiver object constantly sends ack to the sender object

The specification of Loose-Fifo-Network consists of the following:

```
module LOOSE-QUEUE [ pr X :: TRIV ] {
  signature {
    [ Elt < LQueue ]
    op nullLQueue : -> LQueue
    op __ : LQueue LQueue -> LQueue
          { assoc id: nullLQueue }
    op tail_ : LQueue -> LQueue
    op head_ : LQueue -> Elt
  }
  axioms {
    var Q : LQueue
    var E : Elt
    rule Q E => Q E .
    rule Q E => Q .
    eq tail (E Q) = Q .
    eq tail nullQueue = nullQueue .
    eq head (E Q) = E .
  }
}

moduel MSG-LOOSE-QUEUE {
  pr (LOOSE-QUEUE [ X <= view to MSG
                    { Elt -> Msg } ])
}

module LOOSE-FIFO-NETWORK-CLASS {
  protecting(MSG-LOOSE-QUEUE)
  signature {
  *** Attributes
    op Loose-Fifo : -> AId
    [ MsgQueue < AValue ]
  }
}
```

This specification must be split into two parts: a specification for FIFO and reliable communication network and a specification for the communication network which completely lost messages. For the second and third case we remove the corresponding rewrite rules from the specification.

We already define the abstraction function **abs-func**, so the process 3 is finished. Next process (process 4) consists of making a proof score. A proof score is a collection of predicates which we must prove. We must first define the predicate for the initial state of ABP and IN.

```
reduce
get-stat-IN
  { [ < 'Fifo , 0 > | (Fifo = nullMsgQueue) ] }
==
abs-func
  { [ < 'Sender , 0 > | (Sbuf = nullMsgQueue) , (Sflag = false) ]
    [ < 'Receiver , 0 > | (Rbuf = nullMsgQueue) , (Rflag = true) ]
    [ < 'Loose-Fifo , 0 > | (Loose-Fifo = nullMsgQueue) ]
    [ < 'Loose-Fifo , 1 > | (Loose-Fifo = nullMsgQueue) ] }
```

Part of the proof score for the case when ABP receives a message from outside is given below:

```
op sbuf-abp : -> MsgQueue .
op rbuf-abp : -> MsgQueue .
op rbuf-in : -> MsgQueue .
op foo : -> OId .
op bar : -> OId .
op mes-con : -> Content .

var OBJS : Objects .

***> case: Sbuf = nullMsgQueue, Rbuf = nullMsgQueue
***>   case: Sflag = false, Rflag = true

***> abs-func(s)
let abs-func-s =
abs-func (
{ [ < 'Sender , 0 > | (Sbuf = nullMsgQueue) , (Sflag = false) ]
  [ < 'Receiver , 0 > | (Rbuf = nullMsgQueue) , (Rflag = true) ]
  [ < 'Loose-Fifo , 0 > | (Loose-Fifo = nullMsgQueue) ] }) .

***> predicate
reduce
(get-stat-IN
  { [ < 'Fifo , 0 > | (Fifo = (abs-func-s < foo | bar | mes-con >)) ] }
==
abs-func
  { [ < 'Sender , 0 > | (Sbuf = (nullMsgQueue < foo | bar | mes-con >)) ,
                        (Sflag = false) ]
    [ < 'Receiver , 0 > | (Rbuf = nullMsgQueue) , (Rflag = true) ]
    [ < 'Loose-Fifo , 0 > | (Loose-Fifo = nullMsgQueue) ] })
and inv1
  { [ < 'Sender , 0 > | (Sbuf = nullMsgQueue) , (Sflag = false) ]
    [ < 'Receiver , 0 > | (Rbuf = nullMsgQueue) , (Rflag = true) ] } .
```

The term `< foo | bar | mes-con >` represents the message which the sender object receives from outside of ABP. So this predicate checks the local behaviour of the sender object when the state of it is $\langle Sbuf = [], Rbuf = [], Slag = false, Rflag = true \rangle$. The proof score for ABP contains about 100 predicate like this.

In the next process, first stage, we execute the proof score of ABP. CafeOBJ returns the results (true or false) for each predicates. The cases returning true are ok, but for the other cases returning false, we must prove that the system does not go into any of these cases. This checking process is the step 6 (second stage). In the case of ABP only about $\frac{1}{4}$ cases return false. We have only to check these cases using invariants of the system. So we can greatly reduce the size of the process of verification. For example the following proof score returns false.

```
***> case: Sbuf = nullMsgQueue, Rbuf = nullMsgQueue
***> case: Sflag = false, Rflag = false
***> case: Flag = false

***> abs-func(s)
let abs-func-s =
abs-func (
{ [ < 'Sender , 0 > | (Sbuf = nullMsgQueue) , (Sflag = false) ]
  [ < 'Receiver , 0 > | (Rbuf = nullMsgQueue) , (Rflag = false) ]
  [ < 'Loose-Fifo , 0 > | (Loose-Fifo = nullMsgQueue) ]
  [ < 'Loose-Fifo , 1 > | (Loose-Fifo = nullMsgQueue) ] }) .

***> predicate
reduce
get-stat-IN
  { [ < 'Fifo , 0 > | (Fifo = (abs-func-s < foo | bar | mes-con >)) ] }
==
abs-func
  { [ < 'Sender , 0 > | (Sbuf = (nullMsgQueue < foo | bar | mes-con >)) ,
                        (Sflag = false) ]
    [ < 'Receiver , 0 > | (Rbuf = nullMsgQueue) , (Rflag = false) ]
    [ < 'Loose-Fifo , 0 > | (Loose-Fifo = nullMsgQueue) ]
    [ < 'Loose-Fifo , 1 > | (Loose-Fifo = nullMsgQueue) ] } .
```

In such a case (Rflag == Sflag), the receiver object must have received a value from the sender objet, but in this specification Rbuf is empty. So we know that there is no such case for this system. Similarly to this case, we must prove that the case cannot occur in this system for all the cases returning false.

5 CONCLUSIONS

We presented some techniques for formally specifying concurrent and distributed systems in the algebraic specification language CafeOBJ. In our formalism, we can directly use communication networks in specifications. The message passing mechanism based on communication networks increases the expressive power of specifications and provides a powerful verification technique. When one specifies a large system there are several abstraction levels. In our formalism, this

abstraction level can be easily described. Assume we specify a system which uses an ABP for FIFO communication. The lowest level contains a full specification of ABP, and at the higher level we can simply use a FIFO communication network instead of ABP and can hide the details of the communication networks. If we build the specification of ABP in the formalization not using communication networks then the specifications of Loose-Fifo-Networks are spread over other objects and reusability and modularity becomes very low. Using our model, modularity of specifications naturally becomes high.

We use ABP as an example for illustrating the advantages of our formalization. We apply the state mapping technique for communication network based on the concurrent object-oriented model for verifying that ABP is the correct implementation of IN. The cases used in this verification can be systematically derived from the state of ABP. Once we build up a proof score, we can run it and get the faulty cases that we must prove. In the case of ABP, faulty cases are about $\frac{1}{4}$.

It is important how we handle the local non-deterministic behaviour when using CafeOBJ. In this paper we separate the non-deterministic specifications and make them deterministic and try them all. If we can handle these behaviour more directly and efficiently, the verification process of our technique becomes more useful. We think a environment which supports automated verifications in CafeOBJ is important and work is being done for building such an environment.

REFERENCES

Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

Kokichi Futatsugi. Trends in formal specification methods based on algebraic specification techniques – from abstract data types to software processes: A personal perspective –. In *Proceedings of the International Conference of Information Technology to Commemorating the 30th Anniversary of the Information Processing Society of Japan (Info Japan '90)*, pages 59–66, October 1990.

Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

Kokichi Futatsugi and Toshimi Sawada. Design considerations for Cafe specification environment. In *The 10th Anniversary of OBJ2*, October 1995.

Joseph Goguen, Timothy Winkler, José Mesegure, Kokichi Fututsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical report, SRI International, Computer Science Laboratory, 1993.

Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distrubuted algorithms. In F.B. Schneider, editor, *Sixth ACM Annual Symposium on Principles of Distributed Computing*, 1987.

Michael Merritt. Completeness theorems for automata. In *REX Workshop on Stepwise Refinement*. Springer-Verlag, 1989. LNCS Number 430.

José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming*, pages 101–115. ACM, 1990.

José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.

José Meseguer, Kokichi Futatsugi, and Timothy Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. Technical report, SRI International, Computer Science Laboratory, September 1992. also In Proc. of IMSA'92 International Symposium on New Models for Software Architecture, Tokyo, October 1992.

Toshimi Sawada and Kokichi Futatsugi. Basic features of CHAOS specification kernel language. In *The 10th Anniversary of OBJ2*, October 1995.

William E. Weihl. Specifications of concurrent and distributed systems. In Shape Mullender, editor, *DISTRIBUTED SYSTEMS*, chapter 3, pages 27–53. ACM PRESS, ADDISON-WESLEY, second edition, 1993.

6 BIOGRAPHY

Shusaku IIDA: Ph.D. student of Japan Advanced Institute of Science and Technology (JAIST). His current research interests are software engineering, formal methods, algebraic specifications and distributed systems. He is a member of JSSST and IPSJ.

Kokichi FUTATSUGI: Professor of JAIST. He is currently also a Senator of JAIST. In 1975, he entered ETL, MITI, Japanese Government. From 1985 to 1993, he was a section chief at ETL. He was assigned to a Chief Senior Researcher of ETL in April 1992. In April 1993, he was assigned to a professor at JAIST. His current research interests include formal methods for software development, declarative computer language systems, algebraic specifications and their applications to software methodology. He is a member of ACM, IEEE, JSSST and IPSJ.

Takuo WATANABE: Associate Professor of JAIST, joined the faculty in 1991. He received his Ph. D. from Tokyo Institute of Technology in 1991. From 1990 to 1992, he worked at Univ. of Tokyo and Univ. of Illionis at Urbana-Champaign as a JSPS junior researcher. His research interests include: metalevel architectures and computational reflection, object-oriented programming, distributed and mobile computing. He currently is investigating a theory of computational reflection based on rewriting. He is a member of ACM, JSSST and JSPS.