Object Oriented BeNet Programming for Data-focused Bottom-up Design of Autonomous Agents

Tetsushi Oka Junya Tashiro Kunikatsu Takase
Univ. of Electro-Communications Chofugaoka 1-5-1, Chofu Tokyo 182-8585, Japan
{oka,junya,takase}@taka.is.uec.ac.jp

Abstract. In this paper, we propose an approach to developing autonomous agents that behave intelligently in unpredictable environments and a programming method for describing such agents. In our data-focused bottom-up design, the designer develops an intelligent agent as a program which updates a finite set of data objects at regular intervals, modifying the description step by step to extend its behavioral repertoire. In our object oriented BeNet programming, an agent is described in an object oriented programming language based on a real-time parallel computational model. The designer stipulates the roles of the data objects of various classes and assigns them to parallel computational modules. The proposed approach and method are suitable for developing agents which are difficult to develop based on a top-down approach or learning techniques alone. We present the basic idea, the programming tools and the methodology for our approach, showing examples.

1 Introduction

There has been less attempts to develop an independent autonomous agent in a dynamic environment or a complete software for an autonomous robot than inquiries into essential functions of them. Although several control architectures [2][3][5][6][7] have been proposed, it is still difficult to design a desirable agent that interacts with a complex dynamic environment.

There are some reasons for this reality. First, it is not facile to describe the agent's desired properties in simple words or expressions. Secondly, we cannot test an agent in a short period of time, if it is expected to behave well in diverse circumstances and contexts. Thirdly, the environment is often too complex to describe completely and it is difficult to analyze mathematically how the agent behaves.

These significant points lead us to a conclusion that it is almost impossible to acquire the whole description of every desirable agent automatically based on a top-down approach, learning algorithms or searching techniques alone, since all these methodologies proposed until now need a simple description of the desired properties, a complete description of the environment or numbers of tests of different concrete agents.

In a top-down approach, the properties of a desired agent must be described completely on the first stage and the description is converted into a concrete agent. Such kind of direct *agentification*[15] apparently has a limit. We cannot rely on off-line learning or searching algorithms for acquiring the whole description of a desirable agent. As for on-line learning techniques, it will be difficult to evaluate the behavior of the agent using cheap algorithms.

How can we develop a desirable intelligent agent, then? There is no doubt that we need to describe a concrete agent, actualize it, test it and improve it by ourselves. It is essential to understand why it works in a certain situation and doesn't work in another, in order to modify its description to realize a better one. Therefore, the description of an existing agent is a treasure and it will be not trivial how we describe agents. So it must be readable, modular, without ambiguity and easy to modify and extend. In this paper, we propose a bottom-up designing approach and a programming method appropriate for developing modular software of autonomous agents.

2 Models for Autonomous Agents

In this section, we give a theoretical basis for our designing approach and programming method which we propose below. Mathematical models of autonomous agents embedded in dynamic environments are illustrated.

2.1 Dynamic World

In this paper, we assume that an agent is a part of a dynamic world which interacts with the rest of the world exchanging information through sensors and effectors.

Let a vector $\boldsymbol{w}(t)$ denote the state of the world at time t. The world is regarded as a system that is always changing the state $\boldsymbol{w}(t)$. The behavior of the world depends on the state of the world itself since it is a closed system. Thus, the world is described as follows.

$$\dot{\boldsymbol{w}}(t) = f_w(\boldsymbol{w}(t)) \tag{1}$$

$$\boldsymbol{w}(0) = \boldsymbol{w_0} \tag{2}$$

where $\boldsymbol{w_0}$ is the initial state of the world, f_w is a function that maps a vector to another.

2.2 Agents as Continuous Time Systems

As an autonomous agent is a part of the world, its state is a part of the world state $\boldsymbol{w}(t)$. Let $\boldsymbol{ss}(t)$ the state vector of the agent. In general, the behavior of the agent also depends on the world state. Therefore,

$$\dot{\boldsymbol{ss}}(t) = f_{ss}(\boldsymbol{w}(t)) \tag{3}$$

$$\boldsymbol{ss}(0) = \boldsymbol{ss_0} \tag{4}$$

The vector, or a tuple, $\boldsymbol{ss}(t)$ of the agent can be divided into (sensory) input $\boldsymbol{i}(t)$, (motor) output $\boldsymbol{o}(t)$ and internal state $\boldsymbol{s}(t)$. Let $\boldsymbol{es}(t)$ the state of the environment, i.e. the rest of the world. And the agent's property in the environment is completely determined by the following expressions.

$$\dot{\boldsymbol{i}}(t) = f_i(\boldsymbol{es}(t)) \tag{5}$$

$$\boldsymbol{i}(0) = \boldsymbol{i_0} \tag{6}$$

$$\dot{\boldsymbol{s}}(t) = f_s(\boldsymbol{i}(t), \boldsymbol{s}(t)) \tag{7}$$

$$\boldsymbol{s}(0) = \boldsymbol{s_0} \tag{8}$$

$$\dot{\boldsymbol{o}}(t) = f_o(\boldsymbol{i}(t), \boldsymbol{s}(t)) \tag{9}$$

$$\boldsymbol{o}(0) = \boldsymbol{o_0} \tag{10}$$

The agent observes the world through the input vector, changes its state and its output which will affect the environment.

Thus, an autonomous agent embedded in a dynamic world can be completely specified by the description of the initial values of the vectors, $\boldsymbol{i_0}$,$\boldsymbol{s_0}$ and $\boldsymbol{o_0}$, and the two differential equations (7) and (9).

2.3 Agents as RTAA

When we realize an agent using digital circuits and computers, it will be better to design it as a discrete time system that changes the state at regular intervals of δt.

$$\boldsymbol{i}(t + \delta t) = f_i(\boldsymbol{es}(t)) \tag{11}$$

$$\boldsymbol{i}(0) = \boldsymbol{i_0} \tag{12}$$

$$\boldsymbol{s}(t + \delta t) = f_s(\boldsymbol{i}(t), \boldsymbol{s}(t)) \tag{13}$$

$$\boldsymbol{s}(0) = \boldsymbol{s_0} \tag{14}$$

$$\boldsymbol{o}(t + \delta t) = f_o(\boldsymbol{i}(t), \boldsymbol{s}(t)) \tag{15}$$

$$\boldsymbol{o}(0) = \boldsymbol{o_0} \tag{16}$$

We call this simple computational model of agents, RTAA (Real-Time Augmented Automaton) (see Fig.1). An RTAA can be simulated by a single-loop computer program that calculates data arrays for $\boldsymbol{s}(t) \in \boldsymbol{S}$ and $\boldsymbol{o}(t) \in \boldsymbol{O}$ at intervals of $\delta t \in \boldsymbol{R}$. Or it may be realized as a seqential logic circuit which synchronizes with a clock.

The description of an RTAA in a common programming language is better for the designers to understand the behavior of the agent and more useful for a simulation than a continuous time system described in differential equations. Therefore, in the rest of this paper we assume that an autonomous agent is an RTAA.

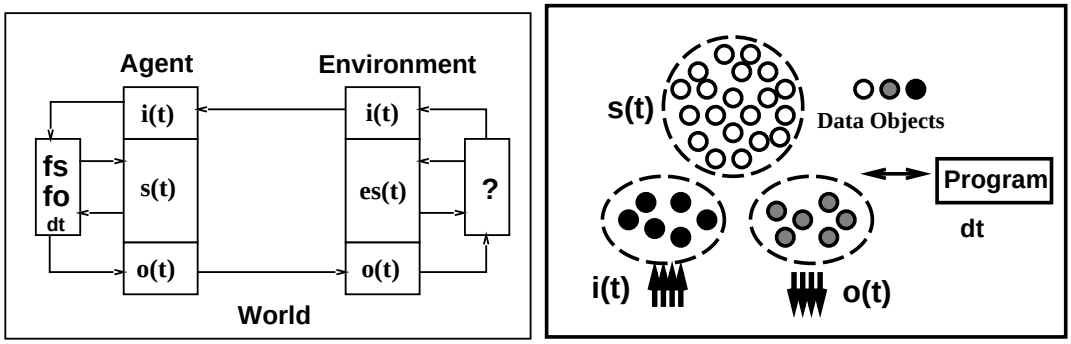

Figure 1: Embedded Agent as an RTAA(left)

Figure 2: Agent's State represented by a Finite Set of Data Objects

3 Data-focused Bottom-up Design

3.1 Searching for a desirable RTAA

The problem of designing an intelligent autonomous agent is how to find a description of a good RTAA which interacts with its environment which is so complex that we cannot give a complete mathematical or computational model to it.

Especially, the whole real world is far more complex than the agent that the designer wants to realize. It is impossible to describe it completely to analyze the behavior of the agent mathematically or by simulation. Therefore, we cannot choose a top-down approach in order to make the whole system desirable like in designing traditional controllers. On the other hand, declarative descriptions of properties of the agent and the environment are insufficient for defining a RTAA completely, though there has been attempts reported in some papers[9][10][14]. Such descriptions are ambiguous to specify the agent's properties in its environment.

Basically, we have to search for a desirable RTAA by repeating generate-and-test cycles. In the generation stage, we describe an RTAA which is likely to behave desirably based on our knowledges, partial world models, know-hows, heuristics and intuition, guided by the description of the existing agents. In the testing stage, we observe both the external behavior and the internal behavior of the RTAA. Such observations may bring us knowledges and experiences useful for building a better agent by modifying the existing RTAA.

For acquiring meaningful knowledges for designing a desirable agent, it is important to analyze how the RTAA behaves in many situations it faces. For such an analysis we should know what kind of information it has in the memory, i.e. the state vector $\boldsymbol{s}(t)$ and how it computes the output vector $\boldsymbol{o}(t)$. If the roles of elements of $\boldsymbol{s}(t)$ are not clear, we cannot describe or modify the program which update their values appropriately while the agent interacts with the environment.

3.2 Data-focused Design

Suppose that we have a good RTAA that behaves well in its environment and want one that has a larger repertoire of behavior. Clearly, we can use the description of the existing RTAA. In the subsumption approach[3] proposed by R. Brooks, the designer tries to obtain an appropriate system adding layers to the existing system. In this process, the designer is modifying the internal state vector $\boldsymbol{s}(t)$ by adding new elements and modifying f_s and f_o in a very specific way using *suppression* and *inhibition*. Some programs of old layers become useless when new layers are added, and the description becomes more and more unreadable as the system goes large.

We propose a more suitable approach to finding a better RTAA using the description of the existing RTAA. In our data-focused bottom-up approach, we describe the state of an agent, $\boldsymbol{s}(t)$, as a fixed finite set of *data* and specify the mappings f_s and f_o with a program which updates it at regular intervals in real time (See Fig.2). We stipulate the role of each data object of the RTAA and modify the internal data representation to extend the behavioral repertoire not only by adding objects, but also by deleting inappropriate or useless objects.

After determining the data set which represents $\boldsymbol{s}(t)$ for the new RTAA, f_s and f_o will be designed for calculating $\boldsymbol{s}(t)$ as a program, utilizing the description of the existing RTAA. In this approach the focus of the designer will be on the *data objects* rather than the *layers* or *behaviors*. If all the data objects have distinct roles, it will be easier to find bugs and problems, understand how the agent works and modify the description.

4 Object Oriented BeNet Programming

4.1 BeNet Model

The BeNet model[12] was proposed for the purpose of describing a parallel modular software for an autonomous robot. A BeNet is a network of parallel RTAA which run asynchronously at their own real-time intervals(Fig.3). An RTAA in a BeNet is called a Behavior Unit(BU). The value of an element of a BU's output vector is sent to other BU's to change the value of an element of their input vectors. Each output element has a single priority value that is used for arbitration. If more than one output elements are connected to the same input element, the value of the element with the highest priority value will be set to the value of the input element.

If we describe an agent as a BeNet instead of a single RTAA, the description will be more readable because the program are decomposed into smaller parallel computational modules. And it will be plain to see which information in a BeNet is necessary for computing a particular element of $\boldsymbol{s}(t)$ from the program.

4.2 Object Oriented BeNet Specification

In the data-focused approach, we assume that the internal state of an agent is represented by a finite set of data objects and that it is updated in real time by a program. It is a good way to describe it in an object oriented language using various abstract classes of data. If we have useful classes for specifying $\boldsymbol{s}(t)$, f_s and f_o, it will be easier to develop desirable agents based on the data-focused approach. A data object in the agent will be realized as an instance of a class and it can be operated with methods of that class. This makes the whole description of the agent simple and readable. Class inheritance will be also useful.

Now we propose to describe an agent as a BeNet in an object oriented programming language and stipulate the roles of the data objects and the BU's. Every data object of the agent is allocated to either the local memory of a BU or the shared memory. The description of the agent will be modular and this programming method is suitable for the data-focused approach. We can focus on a data object of an abstract type and a BU which updates data objects periodically when we develop an agent. We usually don't have to care about the detail of the whole program. If the program has only a single thread, it will be hard to trace when an object will be updated. In addition, a parallel description fits a parallel hardware well.

5 Specifying BeNets in Java

Some programming environments based on BeNet model, BNRB/tp, BNRB/mt and BNS/cl[13] have been applied to exploitation of autonomous robots [12] and real-time agents. These environments help the designer to specify BeNets completely and briefly in C, C++ or Lisp. In these environments the messages between BU's are realized as data arrays rather than objects of arbitrary classes. A BU in a BeNet is not described as an object of a class. These are not desirable for the data-focused approach.

For this reason, we have developed a programming environment in which the designer can describe a BeNet briefly in Java for developing agents based on the data-focused approach. BNJ, our environment is implemented on top of JDK(Java^{TM} Development Kit). Using Java as the description language, we can use abstract data types for the shared data, the local data of a BU and the BU itself and encapsulate $\boldsymbol{s}(t)$, f_s and f_o of an RTAA into objects. The designer can define data structures and BU's extending existing classes and using class libraries.

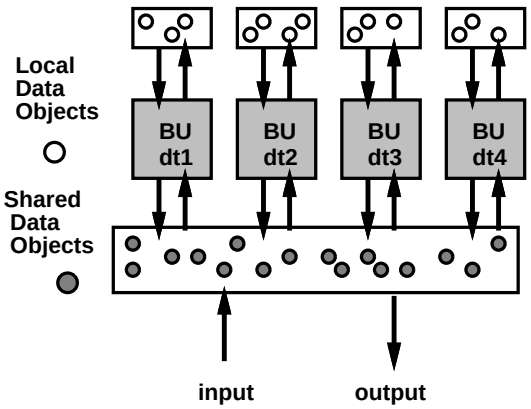

Figure 3: BeNet

5.1 Describing BU's in BNJ

In BNJ, the designer only needs to describe an extended class of the class *BehaviorUnit* for defining a particular BU. *BehaviorUnit* is an extended abstract class of *Thread*. Therefore, an instance of a class of a BU can run as a concurrent thread. In BNJ, a BU is described as follows.

```
class MyUnit extends BehaviorUnit{
// slots
BeNetMess mess;
OutData outdat;
InData indat;
int prio = 0;
...
// constructor
public MyUnit(BeNetMess m,long dt)
{
  super(dt); // setting interval
  mess = m;
  outdat = new OutData();
   ...
}
// method override
  synchronized void readInput(){
  indat = mess.indat.getData();
  ... // getting input
  }
  synchronized void writeOutput(){
  mess.outdat.setData(outdat,prio);
  ... // setting output
  }
  void initialize(){
  ... // s(0),o(0)
  }
  void rule() {
  ... // fs and fo
  }
}
```

The BU's $\boldsymbol{s}(t)$ consists of slot variables of the class. In the constructor, the interval of the BU is determined. The initial state $\boldsymbol{s}(0)$ will be specified by the constructor and the method *initialize*(). The method *rule*() which is the definition of the functions f_s and f_o will be called at intervals of δt. *readInput*() and *writeOutput*() are methods used for defining the network configuration. In the program above, the BU receives a message *mess.indat* every cycle which is stored in the shared memory and uses it as local data, *indat*. The BU tries to change *mess.outdat* in the shared memory to *outdat* with the priority value of *prio* every cycle. If another BU that has higher priority is locking the data, this actually leads to no effect.

5.2 Describing and Simulating BeNets

In BNJ, a BeNet is realized as a Java application program. All BU's are instances of classes extended from *BehaviorUnit*. In the top-level program, all the objects for the BU's are generated and started. The designer also describes data objects for the shared data. In the following example, *bm* is the object for the information exchanged between the two BU's, *mu* and *yu*.

```
public class MyNet{
 public static void main(String args[]){
 BeNetMess bm;
 MyUnit mu = new MyUnit(bm,10);
 YourUnit yu = new YourUnit(bm,10);

  mu.start();
  yu.start();
 }
}
```

Finally, the BeNet is compile by a compiler and simulated by a java machine.

```
% javac MyNet.java
% java MyNet
```

6 Tools and Methods for Designing Agents

6.1 classes for image processing

We have developed classes for image processing using Hitachi IP2000 image processing boards. Methods of the class *IP2000* are realized as native methods described in C which calls IP2000 library functions on Linux2.0. This class enables us to control IP2000 boards with a program in Java.

Class *Image2D* is designed for describing 2-D image processing in an object oriented manner. The methods of *Image2D* calls class methods of *IP2000*. Image data and two dimensional feature maps are represented by objects of this class and these data are operated with its instance methods. For example, program for thresholding a gray-scale image from the camera can be simply described as follows, using our class library.

```
Image2D im1 = new Image2D();
Image2D im2 = new Image2D();

im1.getCamera();
im2.binarizeFrom(im1,minThr,maxThr);
```

In addition to *Image2D*, we have more than 25 classes for image segmentation, visual tracking, feature extraction etc. Thanks to the libraries the designer need not describe a hardware specific program while using a hardware for image processing. It is possible to implement the same classes on other hardware systems.

6.2 motion control

The output vector, $\boldsymbol{o}(t)$ of an autonomous robot is a set of commands to the actuators. It is desirable for the designer if $\boldsymbol{o}(t)$ is described as a set of objects of classes like *Leg*, *Hand*, *Arm* etc. which have slots for the value of the corresponding elements of $\boldsymbol{o}(t)$. Using methods of such classes, programs for motion generation will be more readable. For example,

```
leftArm.stretchQuick();
leftShoulder.rotate(R,T);
```

We have developed classes for describing the motion system of a 4-legged mobile robot briefly. In addition to classes for motor commands, BU's for generating primitive motion patterns such as walking and turning, and BU's for generating actions are developed and applied to design autonomous robots. In BNJ, we can easily design a hierarchical motion system out of BU's[11], utilizing classes described above.

6.3 text/natural language processing

In order to communicate with people in a natural language, the agent has to extract something from the words and make utterances out of words of the natural language in real time. Input and output of such processing consist of words of limited number, the intentions of the agent or the person sending messages to the agent. Language processing is necessary for realizing a program like this.

```
yourIntention =
  yourMessage.extractIntention(context);
context.change(yourIntention);
myMessage = context.makeUtterance();
```

Classes for hash tables, word lists and dictionaries has been built and proved to be useful for describing a program to analyzing sentences by keyword matching and selecting utterances for communicating agents.

6.4 Decision Making

Computation for action selection and attention selection is described in Java in the following manner.

```
context.change(perception);
attention = context.selectAttention();
action = context.selectAction();
```

context, *attention* and *action* are instances of classes defined by the designer. For a communicating agents, messages from other agents and the utterances of the agent will be included in *perception* and *action*.

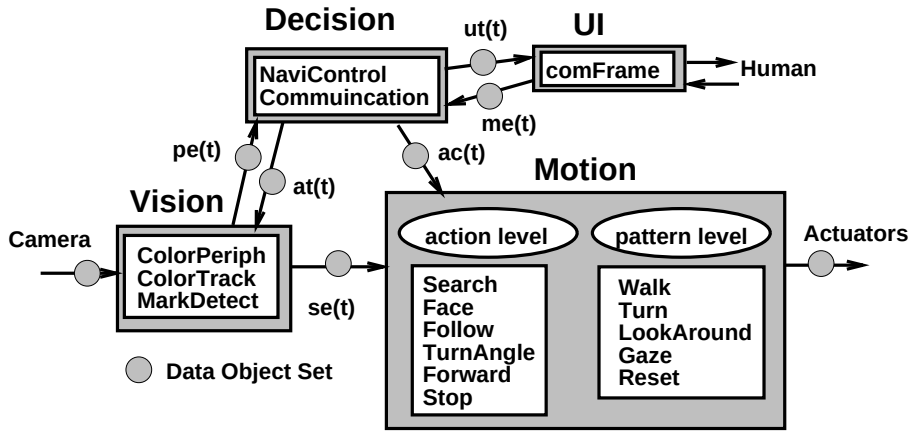

Figure 4: BeNet for a Communicating Autonomous Robot

Table 1: BNJ/user defined Classes for the Communicating Robot

System	Class name
Vision	Image2D,ImageSys BinTracker,YUVArea,ImageWindow Finder,BinaryImageFeatures
Motion	Leg, Neck, Actuator,ImageWindow
Decision	Hash, WordNum,Map,MesNum

7 Concrete Agents

So far, we have developed simulated communicating mobile agents and a real 4-legged communicating autonomous mobile robot in BNJ and conducted some experiments.

An agent described as a BeNet could communicate with a human and another agent to avoid conflicts while acting in the simulated world. The software of a 4-legged robot which makes verbal communications and visually guided navigation was also built and tested in BNJ. It is described as a BeNet in Java. It consists of 3 BU's for vision, 11 BU's for motion, 2 BU's for communication and action selection, and a BU for the window interface to a human(See Fig.4). These BU's interact each other by reading and writing shared data objects.

The vision system updates the object set for *sensation* $se(t)$ to the motion system and the object set for *perception* $pe(t)$ to the decision system based on the images from the camera and the objects for *attention* $at(t)$ from the decision system. The motion system updates the commands to the motors $O(t)$ based on $se(t)$ and the *action* $ac(t)$ from the decision system.

The decision system is designed based on *dynamic scenarios* for communication and visual landmark based navigation. The system updates $at(t)$, $ac(t)$ and the utterance of the robot $ut(t)$ based on the context $cx(t)$, $pe(t)$ and the message from the human $me(t)$. All these internal data objects are updated in real time BU's. Classes in BNJ and user defined classes used for this robot are shown in Table 1. In the presence of these classes and some BU's, we could develop the agent in less than 20 days. The amount of the source code in Java is 6317 lines or 18300 words.

8 Discussion

In the data-focused approach, the state of an agent is represented by a finite set of data objects. This restriction is not too strong because an actual computer has a finite size of memory and it can read and write only a part of the memory in a short period of time. As the objects change their state at regular intervals, the behavior of the agent does not depend on the performance of the hardware.

The CSP model[8] is not suitable for describing a parallel modular program for an autonomous agent based on the data-focused bottom-up design. In this model, the designer has to describe procedures for synchronizations between processes explicitly. This makes it difficult to add a new module. The internal behavior of the agent will be less predictable, because a process often waits for a message from another. Using the BeNet model, the designer can easily add a new data object or a new BU to an existing agent.

In Linda[1] it is also easy to add a data object or a concurrent process, but the restriction is too weak. As concurrent processes in Linda can update shared data at an arbitrary time, the behavior of the program will be difficult to predict. Even if those processes are allowed touch the shared data once in a fixed interval, it will be hard to understand how the whole program behaves if every process can equally touch every shared data. In BeNet, as the number of data objects a process can read and write is limited, it is possible to implement the agent efficiently on a parallel hardware system.

We extended *java.lang.Thread* to realize the common properties of BU's. In C++ or CLOS we cannot realize a concurrent control thread as an object or realize the abstract class *BehaviorUnit* which simplifies the description of a BU to a considerable extent. BNRB/mt was implemented with multi-thread library functions of Solaris 2.X. A BU was described as a function rather than a class. It was necessary to describe a class for the local data of the BU. In BNJ, we can specify a BeNet in a more natural way.

A description of an agent as a BeNet in Java is independent of the hardware and it will be useful for building another agent. For instance, many BU's and data classes in the BeNet in Fig.4 can be used as components of the software of another robot. BU's for the motion system can be applied without lots of changes as long as the body of the robot is unchanged. Adding a new computational module is not a hard task because all the data objects and BU's have distinct roles.

A BeNet can be efficiently simulated by a MIMD machine with a global/local shared memory and local memory/cache. If a BU is fixed to a single processor, its local data can be stored in the local memory. The shared data set should be stored in the global shared memory or in the memory shared by two or more processors. It will be a worthwhile challenge to develop a system on multi-processors that simulates a BeNet in Java more efficiently.

9 Concluding Remarks

We illustrated how to describe autonomous agents as BeNets in Java based on the data-focused bottom-up approach. In this programming method, the state of the agent are realized as a finite set of objects of abstract types which are updated in real time by computational modules of distinct roles. Stipulating the roles of the data, decomposing data into parallel modules, and using abstract data types, the description will be readable and easy to modify the description and analyze the behavior of an existing agent. By developing concrete agents in BNJ, we confirmed that the validity of our method. The BeNet description of an intelligent agent will also help us to understand what makes it behave intelligently.

References

[1] S.Ahuja, N.Carriero and D.Gelernter, Linda and Friends, *IEEE Computer*, August pp.26–34, 1986

[2] R.P.Bonasso and D. Kortenkamp, Characterizing an Architecture for Intelligent, Reactive Agents, *AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*, 1995

[3] R.A.Brooks, A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation*, pp.14–23, 1986

[4] R.A.Brooks and L.A.Stein, Building Brain for Bodies *Autonomous Robots*, Vol. 1, pp.7–25, 1994

[5] D.Chapman, Vision, Instruction and Action, *MIT Press*, 1991

[6] E.Gat, Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-world Mobile Robots, *In Proc. of National Conf. on Artificial Intelligence*, 1992

[7] B.Hayes-Roth, An Architecture for Adaptive Intelligent Systems, *Artificial Intelligence*, vol.72, pp.329–365, 1995

[8] C.A.R. Hoare, Communicating Sequential Processes, Prentice/Hall International, 1985

[9] I.Horswill, Real-time Control of Attention and Behavior in a Logical Framework,*Proc. of 1st International Conf. on Autonomous Agents*, 1997

[10] P.Maes, Situated Agents can Have Goals, In *Designing Autonomous Agents*, P.Maes ed., pp.49–70, MIT Press, 1990

[11] T.Oka M.Inaba and H.Inoue, Describing a Modular Motion System based on a Real Time Process Network Model, *Proc. of the 1997 IEEE/RSJ International Conf. on Intelligent Robots and Systems*, pp.821–827, 1997

[12] T.Oka K.Takeda M.Inaba and H.Inoue, Designing Asynchronous Parallel Process Networks for Desirable Autonomous Robot Behaviors, *Proc. of the 1996 IEEE/RSJ International Conf. on Intelligent Robots and Systems*, pp.178–185, 1996

[13] T.Oka M.Inaba and H.Inoue, Programming Environments for Developing Real Time Autonomous Agents based on a Functional Module Network Model, *Proc. of the 3rd ECPD International Conf. on Advanced Robotics, Intelligent Automation and Active Systems*, pp.326–332, 1997

[14] S.J.Rosenschein and L.P.Kaelbling, A Situated View of Representation and Control *Artificial Intelligence*, 73-1, pp.149–174, 1995

[15] Y.Shoham, Agent-oriented Programming, *Artificial Intelligence*, Vol.60, pp.51–92, 1993