A++: An Agent Oriented Programming Language

Deyi XUE
Department of Mechanical and Manufacturing Engineering
University of Calgary
Calgary, Alberta, Canada T2N 1N4

ABSTRACT

A new Agent-Oriented Programming (AOP) language called A++ is introduced in this research for developing agent-based distributed systems. In this work, agent-oriented programming is defined as a programming method with characteristics of distribution, autonomy, concurrency, and mobility. Both agents and objects can be modeled in A++. In addition to data and methods that can be defined in objects including classes and instances, each agent is also associated with an independent computing process in agent-oriented programming.

Keywords: Agents, Agent-Oriented Programming (AOP), Programming Language, Distributed Systems.

1. INTRODUCTION

The research on modeling agent systems was started based upon the advances of distributed computing, Internet/Web technologies, and artificial intelligence [1]. Although many agent systems have been developed for different types of applications, these systems were primarily implemented based upon the conventional computing techniques such as object-oriented programming, client-server computing, distributed object modeling, and so on.

The concept of Agent-Oriented Programming (AOP) was first introduced by Shoham with the development of an agent modeling language called AGENT-0 [2]. In this language, the state of an agent is composed of components including beliefs, decisions, capabilities, and obligations. Since then, many agent modeling languages have been developed [1].

In our previous research, a distributed system modeling approach has been introduced for mechanical engineering design [3,4]. In this approach, a class at a remote location can be used as a super-class for defining a new sub-class at the local site. A class defined at a remote location can be used for creating an instance at the local site. An instance at a remote location can be used for modeling the database at the local site. A super-set language of Smalltalk has also been developed in this research [5,6].

Despite the progress, the presently developed agent-oriented programming languages haven't been adopted as general tools due to their limited programming capabilities. The objective of this research is to introduce a new computer language for agent-oriented programming, based on the advances of the popular programming languages including C++, Java, and C#.

2. AGENT-ORIENTED PROGRAMMING

As the Object-Oriented Programming (OOP) can be characterized by its abstraction, encapsulation, inheritance, and polymorphism, the Agent-Oriented Programming (AOP) is characterized by its *distribution*, *autonomy*, *concurrency*, and *mobility*.

(1) Distribution of Agents
Distribution of agents allows different agents to be modeled at different locations and associated into an integrated environment.

(2) Autonomy of Agents
Autonomy of agents makes the activities of one agent to be independent of the activities of other agents.

(3) Concurrency of Agents
Concurrency of agents allows the activities of different agents to be conducted simultaneously and coordinated effectively.

(4) Mobility of Agents
Mobility of agents means the programs implemented at remote locations can be used at the local site.

3. AGENTS VS. OBJECTS

As the objects, including classes and instances, are used as the primitives in object-oriented programming, agents, including class agents and instance agents, serve as the primitives in agent-oriented programming. The agent concept in A++ was rooted from the object concept with considerations to include the advances of the well-established object-oriented programming method into the agent-oriented programming method. However agents are different from objects due to their distinctive nature of distribution, autonomy, concurrency, and mobility, which are not provided in objects. Both agents and objects can be modeled in agent-oriented programming using A++.

An object is characterized by its data and methods to access these data. An agent, on the other hand, is characterized by:

Agent:
- Data
- Methods to access the data
- Process to control the activities

Agents are also defined at two different levels: class level and instance level. All the characteristics of objects, including

abstraction, encapsulation, inheritance, and polymorphism, are part of the characteristics of agents. When agents with independent computing processes are not used, the programming is then considered as object-oriented programming.

Although objects are actually sub-sets of agents (i.e., an object is a special case of agent without an independent computing process), we organize agents and objects in a tree shown in Fig. 1. The agents and objects share some common characteristics. Objects are not considered as agents or special cases of agents. Both agents and objects are defined as sub-classes of the Primitive class. Therefore agents and objects are two different types of primitives. The Agent class and the Object class are two built-in classes that cannot be modified by users.

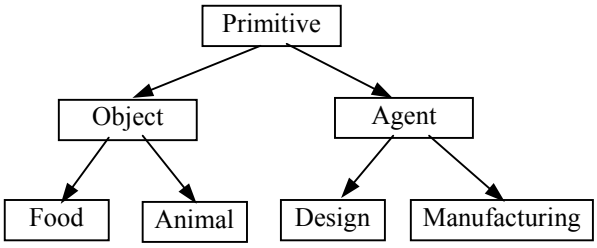

Fig. 1: A Tree of Object Classes and Agent Classes

Execution of an application with objects starts from a method of an object and terminates to this method. Only one computing process is usually required for the execution of these objects in the application. Execution of an application with agents is conducted through parallel computing processes. When an agent is created, a computing process is then generated automatically to control the activities of this agent until the agent is deleted.

4. CLASSES AND INSTANCES

Both agents and objects are defined by classes and instances. Instances are created using classes as the templates. A class is defined by

```
class <class>: <super-class>
{
    // class body
};
```

such as

```
class Optimization: Agent
{
    // Optimization class body
};
```

A class is defined by its members including variable members and method members. Variables and methods are classified into class variables/methods and instance variables/methods. A class variable is shared by all the instances of this class. An instance variable for an instance of the class is not shared with other instances. A class method is executed by the class, while an instance method is executed by the instance of this class.

An instance is created by

```
new <class>;
```

The instance agent is kept active all the time without additional loop defined by the user. The computing process of an agent can be terminated by asking the instance agent to execute a method called AgentTerminate().

5. AGENT-ORIENTED PROGRAMMING IN A++

Many new functions have been introduced in A++ for providing the characteristics of agent-oriented programming: distribution, autonomy, concurrency, and mobility.

5.1. Distribution Functions in A++

In A++, the classes and instances are preserved at different locations called *Internet nodes*, or simply *nodes*. Each node is identified by a node name with a prefix @, such as

```
@node1
```

Each node is unique in the Internet. For instance, a node can be defined by a Domain Name Server (DNS), such as enme.ucalgary.ca and a port number, such as 9876. The nodes are linked by the Internet, as shown in Fig. 2.

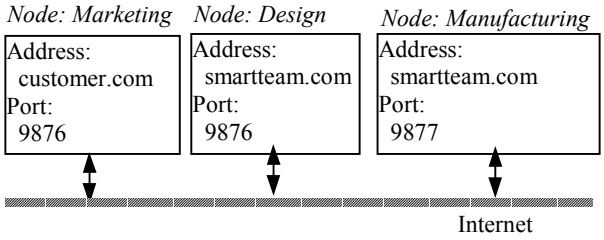

Fig. 2: Internet Nodes Linked by Internet

In A++, the inheritance mechanism and the instantiation mechanism are extended to the classes and instances distributed at different nodes.

In A++, a class at a remote location is described by

```
<node>.<class>
```

such as

```
@node1.ClassA
```

When a remote class is used as the super-class to define a sub-class at the local site, the new class is defined by

```
class <sub-class>: <node>.<class> {...};
```

such as

```
class B: @node1.ClassA {...};
```

When a class is used for creating an instance, all the instance variables defined in the class and its super-classes, both at the local node and at the remote nodes, are instantiated to this instance automatically.

In A++, an instance at the remote location is described by

```
<node>.<instance>
```

such as

```
@node1.instance1
```

Since instances are usually preserved in the instance variables of other instances, global variables are usually used to describe the instances, such as

```
MyObject instance1; // a global variable
class A: Object {
        public void method1() {
                instance1 = new MyObject();
                    ... ...
        }
    ... ...
}
```

An instance at a remote node can be assigned to an instance variable of an instance at the local site. In the example shown in Fig. 3, two mechanisms, a pulley-belt drive mechanism and a gear-pair mechanism, defined at two locations are integrated at the node Mechanism using a connection component. In this model, the instance pulleyBeltDrive1 in node PulleyBeltDrive is composed of 5 instances described by its instance variables. The instance gearPair1 in node GearPair is composed of 4 instances described by its instance variables. In the node Mechanism, these two instances are linked by an instance called connection1. By defining the relations among instance variables of the instances, the 12 instances in the three nodes are integrated into the same environment. When the rotational speed of the shaft2 in node PulleyBeltDrive is changed, this change is propagated to the rotational speed of the shaft1 in node GearPair.

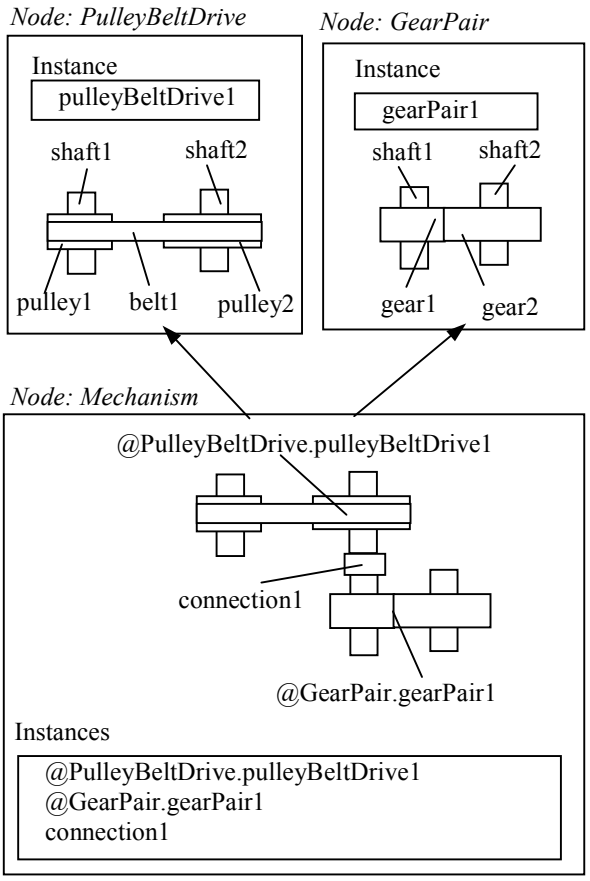

Fig. 3: Distributed Instances

5.2. Autonomy Functions in A++

The autonomy functions of A++ are primarily realized by the independent processes of agents. An application with only objects is usually executed within only one computing process, as shown in Fig. 4 (a). An application with agents is usually executed with many computing processes of these agents, as shown in Fig. 4 (b).

In the conventional object-oriented programming, variables and methods defined in a super-class are inherited by its sub-classes, and variables and methods define is a class are instantiated to its instances. Changes of the descriptions in a class lead to the changes of the behaviors of the sub-classes and instances created from these classes. In A++, since the agents are also defined by classes and instances, a static inheritance mechanism and a static instantiation mechanism are introduced to maintain the autonomy of agents.

In A++, two inheritance mechanisms, a dynamic inheritance mechanism and a static inheritance mechanism, are provided. A dynamic inheritance relation between a super-class and a sub-class is defined by

```
dynamic class <sub-class>: <super-class> {...};
```

The dynamic inheritance mechanism is the same as the traditional inheritance mechanism used in C++, Java, and C#. Due to this nature, the keyword dynamic is optional.

A static inheritance relation between a super-class and a sub-class is defined by

```
static class <sub-class>: <super-class> {...};
```

When static inheritance mechanism is used, all the variables and methods defined in the super-class are inherited permanently to this new sub-class. Changes of the descriptions in the super-class are not propagated to the new sub-class. Fig. 5 (a) and (b)

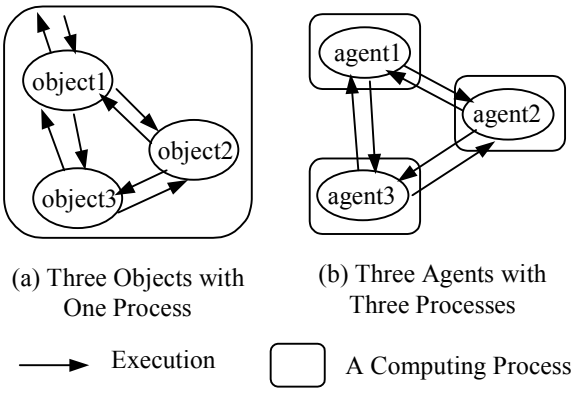

Fig. 4: Computing Processes of Objects and Agents

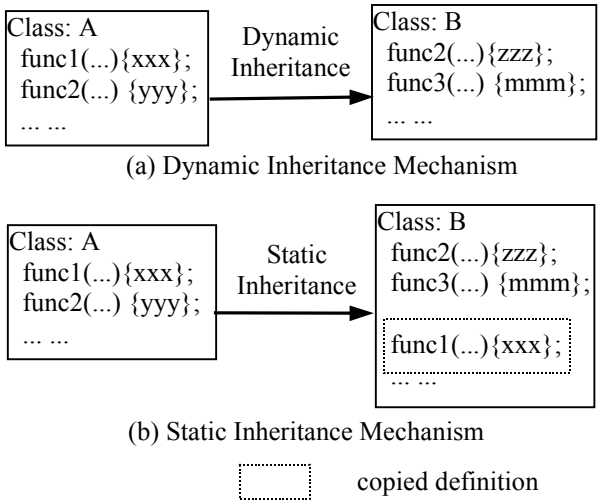

Fig. 5: Dynamic Inheritance Mechanism and Static Inheritance Mechanism

show examples of the dynamic inheritance mechanism and the static inheritance mechanism.

In A++, two instantiation mechanisms, a dynamic instantiation mechanism and a static instantiation mechanism, are also provided. A dynamic instantiation relation between a class and its instance is defined by

```
dynamic new <class>;
```

The dynamic instantiation mechanism is the same as the traditional instantiation mechanism used in C++, Java, and C#. Due to this nature, the keyword dynamic is optional.

A static instantiation relation between a class and its instance is defined by

```
static new <class>;
```

When static instantiation mechanism is used, all the instance variables and instance methods defined in the class are instantiated permanently to this new instance. Changes of the descriptions in the class are not propagated to the new instance. Fig. 6 (a) and (b) show examples of the dynamic instantiation mechanism and the static instantiation mechanism.

5.3. Concurrency Functions in A++

In A++, many computing tasks are created and executed simultaneously. Coordination of the execution of these tasks is conducted by a multiple-task control mechanism, which was developed based upon concurrent programming principles. Tasks are also called processes or threads in this work.

(1) Thread

A thread is created by executing an instance method, as shown in the following example:

```
thread object1.func3(24,30);
```

The thread terminates when the execution of the method is completed.

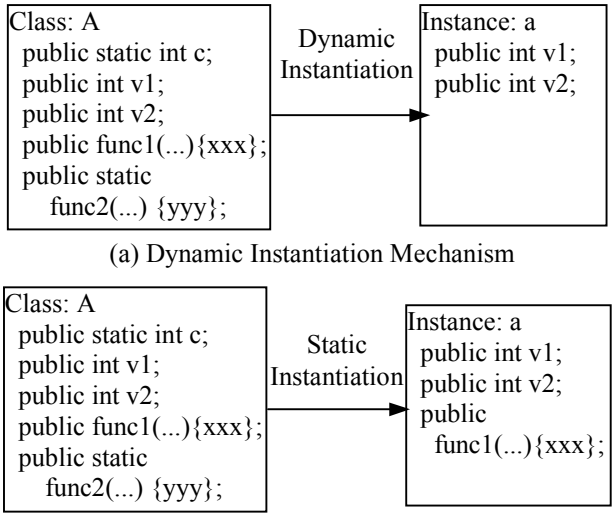

Fig. 6: Dynamic Instantiation Mechanism and Static Instantiation Mechanism

(2) Priority

Tasks are associated with priorities. Each priority is described by an integer. Execution of a task with lower priority is conducted only when all tasks with higher priorities are completed. Default priority is 0.

A++ Environment

The task of A++ environment always has the highest priority, thus being able to interrupt activities of other tasks.

Agents

Each agent can be associated with a priority. When the execution of an instant agent with higher priority is not required, the execution of an instance agent with lower priority is then considered. For example, the following two expressions

```
priority 2 new MyAgent(20, “Tom”);
priority 3 new YourAgent(30, “Peter”);
```

are used for creating 2 instance agents with different priorities.

Methods of Agents

Methods of agents can be executed in different tasks by defining these methods using the keyword thread. For example, the following three expressions are used for defining priorities of three methods.

```
priority 2 thread public int func1(...) {...};
priority 0 thread public int func2(...) {...};
priority MAX thread public int func3(...) {...};
```

Internet Nodes

Tasks are also created by the requests from the remote Internet nodes. The priorities of these tasks are determined by the

priorities of these nodes. For example, the following expression is used for defining the priority of an Internet node.

```
priority 2 InternetNode @node1;
```

Threads

Threads are tasks created by executing instance methods. The following expression is used for defining the priority of a thread.

```
priority 2 thread object1.func3(25, "Peter");
```

(3) Synchronization

Synchronization mechanism prevents activities in several tasks from being conducted simultaneously. Examples of task synchronization are shown as:

Agents

```
synchronized new MyAgent(20, "Tom");
synchronized new YourAgent(30, "Peter");
```

Methods of Agents

```
synchronized public int func1(...) {...};
synchronized public int func2(...) {...};
synchronized public int func3(...) {...};
```

Internet Nodes

```
synchronized InternetNode @node1;
synchronized InternetNode @node2;
```

Threads

```
synchronized thread object1.func3(25, "Peter");
synchronized thread object2.func3(30, "Tom");
```

(4) Timer

Timer is used to create a special task with the highest priority at a predefined time. An example of timer task is shown as:

```
MyTimer t = new MyTimer();
t.SetUp(03/02/01/15/06/32);
```

The actual program to be executed is defined in a method called run() of the class MyTimer, such as

```
public void run() {
   object1.func2(25,"Peter");
};
```

(5) Semaphore

Semaphore mechanism uses variables to control the execution processes of different tasks. Two methods are used for implementing the semaphore mechanism.

- Signal(variable)
  If the value of the variable is 0, the value of this variable will be changed to 1. If the value of the variable is 1, the value of this variable will be kept as 1.

- Wait(variable)
  If the value of the variable is 0, the system will do nothing, but waiting for the variable value change. When the variable is 1 or changed to 1, the expressions following Wait() method will be executed, and the variable's value is changed back to 0.

(6) Messages

Messages are used for controlling the execution processes of different tasks. Two methods are used in this mechanism.

- SendMessage(agent, message)
  A message, described by a string, is sent to an agent.

- ReceiveMessage(agent, message)
  The current process waits for a message from an agent. When a message is received from this agent, the expressions following the ReceiveMessage() method call will be executed.

(7) Preemption

Preemption is a mechanism to prevent an expression or a collection of expressions from being interrupted by other tasks. This mechanism is effective for the expressions involving the access of computer devices such as keyboard, monitor, printer, file, etc. An example of preemption is shown as:

```
preemption {
        a.print(1,3);
        @n1.o2.show(2,5);
};
```

5.4. Mobility Functions in A++

The mobility of agents is the capability that the programs implemented at one site can be used at another site.

(1) By defining a class using a remote class as the super-class, the descriptions of the remote super-class are inherited to the new sub-class automatically.

The new sub-class can be defined as a dynamic class or a static sub-class, such as

```
class B1: @A.A1 {...};
static class B2: @A.A2 {...};
```

When the new class is defined as a dynamic class, changes of the descriptions in the remote super-class influence the behaviors of the sub-class. When the new class is defined as a static class, changes of the descriptions in the remote super-class don't influence the behaviors of the sub-class.

(2) By creating an instance using a remote class as the template, the descriptions of the remote class are instantiated to the new instance automatically.

The new instance can be defined as a dynamic instance or a static instance, such as

```
A1 a1 = new @A.A1();
A2 a2 = static new @A.A2();
```

When the new instance is created as a dynamic instance, the instance variables are copied to the new instance. The class variables and instance methods defined in the remote class can be used by this instance. Changes of the descriptions in the remote class influence the behaviors of the instance. When the new instance is created as a static instance, both the instance variables and the instance methods are copied to the new instance. Therefore changes of the descriptions in the remote class don't influence the behaviors of the instance.

(3) The remote classes and their class variables/class methods can be accessed directly from the local site.

The remote class variables and class methods are accessed by the expressions such as

```
int var1 = @A.YourClass.ClassVariable1;
int var2 = @A.YourClass.ClassMethod1();
```

The remote method call mechanism is implemented using the client-server communication architecture. When a remote method at node B is called from node A, A is then the client and B the server. Since program at each Internet node can call methods located at other Internet nodes, and be executed by programs at different Internet nodes, a node is therefore a client of many servers and a server of many clients.

(4) The remote instances and their instance variables/instance methods can be accessed directly from the local site.

The remote instance variables and instance methods are accessed by the expressions such as

```
int var1 = @instance1.InstanceVariable1;
int var2 = @instance1.InstanceMethod1();
```

In A++, classes are organized by Internet nodes. Different classes with the same class name can be defined in different Internet nodes. Therefore the polymorphism is realized at two different levels, Internet node level and class level. For the execution of a method, both its class and its Internet node should be identified.

6. CONCLUSIONS

A new computer language A++ is introduced in this research for developing Agent-Oriented Programming (AOP) systems. The A++ provides four characteristics of the Agent-Oriented Programming: distribution, autonomy, concurrency, and mobility. Both agents and objects can be modeled in A++. The characteristics of A++ are summarized as follows:

(1) Distribution
A new class can be defined using a remote class as the super-class. A new instance can be created using a remote class as the template. An instance at a remote site can be treated the same as an instance at the local site.

(2) Autonomy
Both the dynamic relationships and the static relationships between a super-class and its sub-class and between a class and its instance can be defined.

(3) Concurrency
Many concurrent programming functions, including priority, synchronization, timer, semaphore, message, preemption, are provided in A++ to coordinate the tasks of agents, methods of agents, and Internet nodes.

(4) Mobility
The A++ allows the programs defined at remote locations to be used at the local site by selecting the remote classes as the super-classes to define sub-classes at the local site, using the remote classes to create instances at the local site, treating the remote instances the same as the local instances, and executing the methods of the remote classes and instances directly.

7. REFERENCES

[1] W. Shen, D. H. Norrie and J. P. Barthes, **Multiagent Systems for Concurrent Design and Manufacturing**, Taylor and Francis, 2001.

[2] Y. Shoham, "Agent-oriented Programming", **Artificial Intelligence**, Vol. 60, No. 1, 1993, pp. 51-92.

[3] F. Zhang and D. Xue, "Distributed Database and Knowledge Base Modeling for Concurrent Design", **Computer-Aided Design**, Vol. 34, No. 1, 2002, pp. 27-40.

[4] D. Xue and Y. Xu, "Web-based Distributed System and Database Modeling for Concurrent Design", **Computer-Aided Design**, Vol. 35, No. 5, 2003, pp. 433-452.

[5] D. Xue, "An Experience of Representing Knowledge and Data in Mechanical Design Using Smalltalk-80", **OOPS Messenger**, Vol. 5, No. 3, 1994, pp. 37-46.

[6] D. Xue, "Developing a Superset of Smalltalk for Modeling Mechanical Systems", **Journal of Object-oriented Programming**, Vol. 13, No. 5, 2000, pp. 12-17.