Mobile Agent Modelling Notation

Yih-Jiun Lee
Department of Information Management,
ChienKuo Technology University, Taiwan
yjlee@cc.ctu.edu.tw

ABSTRACT

Mobile Agent Modelling Notation (MAMN) is a modelling language, designed for distributed execution environment. MAMN uses the idea of ambients, which allows an executable environment to move around over networks. Processes or agents can be carried to particular places and continue their tasks. Thus the mobile computing can be both stateful, and stateless. Another benefit to use MAMN is that part of security processes is included in MAMN. Therefore, the user does not need to define the processes by him. MAMN is based on a pratical notation Java Ambients. A MAMN model can be converted into an executable java program to run and test its correctness. It is a straightforward modelling notation, which has wide usage for most of the distributed scenarios.

1: INTRODUCTIONS

Distributed computing has become an important benchmark in computer science. Many novel technologies have been proposed and implemented, such as web-services and grid computing, to achieve the goals of distributed. Most of them try to share resources (including physical resources, such as computing power or data sets) among components. However, when the number of distributed computation projects raises up, the most important thing is to secure the computation both for service providers and service requestors.

This paper provides a modelling language to study the problems over distributed computing and its evolution. The modelling language is called Mobile Agent Modelling Notation (MAMN), which is derived from Java Ambient and has similar but easier syntax. A MAMN model can be implied to a run-able Java code, so the correctness of model can be examined. The reductions in MAMN are "in", "out", "be", "open" and "repeat". The research is based on the idea of a movable and executable environment, which is called ambients. Ambients can carry one or several processes (sometimes called gremlins) in them. Processes may be loaded or unloaded when they reach their destination. The mobile ambient idea was firstly proposed by Dr. Luca Cardelli.

This paper is structured as below. Section 2 briefs some related works regarding modelling languages. Section 3 introduces Mobile Agent Modelling Notation (MAMN) and the reductions in MAMN by demonstrating a public transportation scenario. Then, section4 shows an electronic cash system, which can be separated into three phases: withdraw, shopping and deposit. A discussion will also be given. Finally, a conclusion is made in Section5.

2: REVIEW OF RELATED WORK ON MODELLING LANGUAGES

Model checking is an automatic technique for verifying finite state concurrent systems [4]. Earlier research tried to reason infinite state systems, but can only be provided by well-educated experts. Model checking provides a new technology to verify finite state concurrent systems, and the verification can be performed automatically. The procedure usually includes a search of system states to find the correctness of systems.

Models are used to describe the behaviour of complicated systems. A model should define the properties of a system or an abstraction of a system. Modelling concurrent systems usually includes the communication occurring between inter-processes. With property and behaviour descriptions, the model should be able to predict the execution of system. However, modelling concurrent systems is complicated, because processes in concurrent systems are parallel and asynchronous. According to [5], models usually capture "the essence of a subject" from different observations, such as in the traditional shared variables model and the exchange functions model. The shared variables model has two kinds of objects, processes and shared variables. Each process can perform asynchronously and independently. Processes communicate to each other by accessing the shared variables. The exchange functions model describes another mechanism for distributed and embedded systems. It assumes that communications occur in the model through calling functions to exchange values. Communication in function is "bidirectional, simultaneous and symmetric [5]". Modern modelling languages usually focus on mobility, and they will be explained later.

The evolution of modelling languages in concurrent computing started with the modelling of static connectivity among processes. Usually, the number of processes in the model has to be a fixed number. CSP [7]

and CCS [10] were the earliest efforts on static connectivity. However, with the growth of the Internet, communication appears and disappears unexpectedly. Connection frequently changes. Therefore, the second generation of process calculi was developed for dynamic connectivity. pi-calculus and asynchronous pi-calculus can model channel mobility. However, the model still lacks the capability to model the distributed locations of processes [13]. It is important to model the location of the processes, because different administrative domains have different restrictions. Thus, Ambient Calculus can model mobility in active mobile computation. In the following sections, CSP and Ambient Calculus will be addressed to represent two generations of modeling languages.

2.1: Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) is a modeling language for concurrent and distributed computation. CSP was defined in a paper by C. A. R. Hoare in 1978 [8] and in a book in 1985 [7]. It can be used to validate program correctness and operating systems description. In order to model operating systems, dynamic process creation and recursion are prohibited. In addition, because the processes are static, the possible communications can be defined at system creation.

A CSP program defines a set of processes and the number of processes has to be fixed. The CSP program also defines the communication happening between two processes, such as one message coming from one process and sending to another process. Thus, communication has to be synchronous, with unidirectional information flow [5]. Moreover, CSP can name the channel used in the communication. Therefore, a process can block when it is executing a communication action (either input or output). It continues execution when the corresponding action happens. More specifically, CSP allows users to "describe systems as a number of (parallel) components (processes) that operate independently and communicate with each other solely over well-defined channels" [1].

The object of CSP is to validate the correctness of programs and operating systems. Its semantics become popular in describing concurrent programs. Although many extensions have been added to CSP to support different situations, the communication in CSP or revised version of CSP has to be synchronous. Thus, the communication statements must appear as a pair. One process outputs a message to an indicated process, which contains a statement waiting for an incoming message from the former process. Moreover, both parties have to be predefined at system initiation. CSP does not support message buffering and output guard, but they can be still achieved by means of user defined macros or processes.

2.2: pi-Calculus

The pi-Calculus is a model of concurrent computation based on the notion of naming [12, 11]. The pi-Calculus provides a solution in describing and analyzing agent composed systems and members of which can interact with each other. Since mobile agents may be linked dynamically during the system execution and the information carried in the communication may also change the linkage states. pi-Calculus provides a "naming" method to achieve the link mobility.

A pi-Calculus model defines a sequence of communication by means of names and processes. Using "naming", the namer and named are concurrent entities [11]. Therefore, the communication in the system must be synchronous, (although many modified asynchronous pi-Calculus have been proposed in the recent year, such as [6]). The reasons that naming is involved in the calculus are: 1. naming may provide better independence; 2. naming shows the co-existence of sender and receiver; and 3. naming treats name and location as the same prospection. Although this language is used to model mobile agents, one thing to mention is that names are naming of channels but not agents.

pi-Calculus achieves the goal of link mobility by allowing link dynamically changed. However, in the original pi-Calculus, passing a process through links is possible but forbidden for several reasons. First, passing a process may cause a replication of process along with its state. The replication might become a site-effect. Second, to pass process P1 to P2 might give P2 the right to access the whole P1, but in the design, passing a value, a name or a link only gives the receiver partial access to another. (For example, Q, in the earlier example, can only communicate to R via channel "a" and R can still use another channel to talk to other processes.) Finally, sending a link was very popular in the communication and computation. Although sending a process is not recommended, users can still define an input channel as a trigger of process to achieve the goal of process activation.

2.3: Ambient Calculus

Ambient Calculus [2, 3] was proposed by Dr.Luca Cardelli. An ambient might be defined as a place which is delimited by a boundary and where computation happens. An ambient has a name, a collection of local processes and a collection of "sub-ambients". Cardelli also refers to an ambient informally as a folder which is a moveable thing. A folder can move from one host to another and be executed on the server as a local object. The things which are stored in an ambient are gremlins, such as executed code and sub-folders. The gremlins can be moved from one folder to another if two folders are neighbours. These activities which are defined as four reductions: enter, exit, open and copy and other related information will be detailed in the later session. Because an ambient can move over the Nets, Cardelli provides a naming method to secure the system. All objects in the

system are named by obeying several rules and they can verify and be verified by each other through its name. The processes, called gremlins in Ambient Calculus, are free until they want to cross another boundary. It is called "capability-based model" of security which can be used with "cryptography–based model" and an "access-control-based model" security system. Access-control is used to implement RPC-like invocations that have to cross boundary and authenticate every time. The cryptography-based model is achieved by interpreting encryption keys as ambient names. These issues will be discussed in the following sections

3: MOBILE AGENT MODELLING NOTATION (MAMN)

Mobile Agent Modelling Notation (MAMN) are proposed in this section. This notation is partially derived from Ambients Calculus and from a practical notation called Java Ambients [9]. MAMN uses a similar notation to Java Ambients. A MAMN model can be implied to a runnable JAVA code to examine its correctness. Here, we use a public transportation scenario (which is from [3]), as Figure 1 to introduce the reductions used in MAMN.

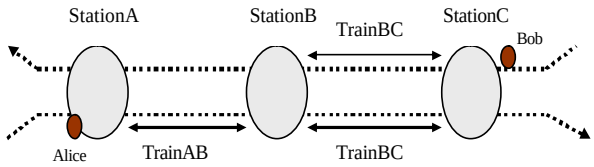

Figure 1. Train and Station Scenario

In this scenario, there are three train stations: stationA, stationB and stationC which are unmoveable ambient. Three trains which are moveable ambients are named as trainXY that means this train routes between stationX and stationY. Two passengers, Alice and Bob, are also moveable ambients. Alice will travel from stationA to stationC and Bob wants to go the other direction. The passengers do not concern about which route they will travel and which train they should board. In this chapter, we will introduce the syntax of MAMN by using this example. Suppose, StationX and StationY are two unmovable ambients and train is movable ambient.

3.1: Basic Reductions

3.1.1: Trains' Behaviour : "enter", "exit" and "repeat"

in

"Enter" is an action which allows one folder to enter another folder. Assumption, train is outside of StationX and has a gremlin which is "in StationX".

```
ambient StationX{
}
! ambient train{
   in StationX;
}
```

After execution, it is shown that train moves into StationX.

```
ambient StationX{
   ambient train{
   }
}
```

However, the target ambient of "in" has to be the neighbour of the current folder. Thus, if train is currently in StationY then "in StationX" should be blocked until train and StationX are neighbours.

```
ambient StationX{
}
| ambient StationY{
   ambient train{
     in StationX;
   }
}
```

In the last example, the only thing that can unlock the "in" reduction is the train leaves StationY and StationX becomes reachable.

Out

The executed condition of "out StationY" which is in folder "StationY" is train' s parent. So the earlier example can be modified as:

```
ambient StationX{
}
| ambient StationY{
   ambient train{
     out StationY;
     in StationX;
   }
}
```

Then after execution those two reductions, the current state is:

```
ambient StationX{
   ambient train{
   }
}
| ambient StationY{
}
```

"out" reduction has only one precondition, which is "the train must be inside the object which the train wants to leave".

repeat

Reduction "repeat" is a simple "copy" process, which allows the system to duplicate the specified process on demand. For instance, "repeatP" can also be written as !P or P;!P, meaning process P is followed by process P. It is a recursion.

Combination

Therefore, the train finally shuttles between StationX and StationY by defined its behaviour as :

```
ambient train{
  repeat{   in StationX; out StationX;
            in StationY; out StationY;   }
}
```

3.1.2 : Passenger' s behaviour

In this scenario, there are two passengers, Nancy and Alice. Nancy wants to travel from StationX to StationY by train. Nancy can be simply defined as

```
ambient Nancy{
  in StationX; in train;
  out train; out StationY;
}
```

However, there is a problem. In the definition, Nancy embarks "train" and leaves "train". That means, Nancy can leave the train at any time, even when train is moving. It does not fit the requirement of Nancy arriving StationY. Thus, Nancy, the passenger, must notice the train' s traveling status to decide whether to disembark. To satisfy the requirement, reduction "be" is designed.

"be" is the name changing reduction. It is used for that a process may wait for an ambient to change its name to a given name [3]. To distinguish the train is moving or stopping at any station, the train can be given different names for different circumstance. For instance, the train can be named as "moving" when it is moving; the train is named "XYatX" if it shuttles between stationX and stationY and is currently at stationX; the train is named "XYatY" if it shuttles between stationX and stationY and is currently calling at stationY. Then the train, in the scenario, must be modified as:

```
ambient train{
  repeat{
    be moving;    in StationX;
    be XYatX;     wait triptime;
    be moving;    out StationX;
    in StationY;  be XYatY;
    wait triptime;
    be moving;    out StationY;
  }
}
```

Correspondingly, Nancy now is defined as

```
ambient Nancy{
  in StationX; in XYatX;
  out XYatY; out StationY;
}
```

Thus, Nancy cannot leave the train unless the train' s current name is XYatY.

3.2: Messaging

The other important instructions in MAMN is messaging. There are one pairs of instructions, output and input, to send and to get the messages. The precondition of messaging is that only the receiver who is "holding" the sender gets the message. For instance,

```
ambient A{
  ambient B{ output M }
  | input N
}
```

Message M is sent by ambient B, whose parent is ambient A, so the receiver of the message is ambient A. In this scenario, ambient A is using a variable N to take the incoming message. Therefore, after transmission, N is equal to M. The usage of sending message is actually designed for sending instruction(s). Suppose, ambient Process is moving to station A with a message "open Process" to output.

When station A is taking the message by using "input N; N;", N becomes an instruction "open Process" to proceed. Then the gremlin(s) in ambient Process become the local processes in station A.

4: ELECTRONIC CASH SYSTEM IN MAMN

Authentication, authorization and delegation which are useful in the e-business scenario have been proved that it is available in MAMN, which can be found in [9]. In this section, a practical electronic cash mechanism is introduced.

An e-cash system should have some features: acceptability, guaranteed payment, no transaction charges and anonymous. In this example, we will show that the emoney can be withdrawn from the bank, verified by the user, paid to the shop to exchange good and deposit into bank account. To be easily description, this system can be divided as three phases: withdraw, shopping, and deposit. In withdraw, the user sends a request to a bank, and get a money bank. The bank has to verify the request which is valid. In shopping, the user sends the money to the shop to swap the goods back. Finally, the shop can save the money to the bank account and exchange the receipt back. Because many processes in the three phases are similar, we can define some macros to be reused.

4.1: Macro definition

In this section, several reusable macros are defined. Some of the macros creates a new ambient and other creates a new gremlin. The reason that macros are designed is that if the user is familiar with the function of a macro, a model can be easier to read and written.

```
define ActiveBehaviour
       (current,Sender,Receiver,OpenTarget,SB)=
   ambient current {
      open OpenTarget; out Sender;
      in Receiver; open OpenTarget;
      out Receiver; in Sender;
      SendBack(current, SB);
   }

define SendBack(target, amb)=
   ambient amb{ out target ; }

define BeGO(otarget, btarget)=
   ambient temp{
      out otarget; be btarget;
   }

define check(target1,target2,.,targetN)=
   { open target1; open target2;
     . . open targetN; }

define OutBeGO(current, parent)=
   ambient current{
      out parent; BeGO(current, GO);
   }
```

4.2: Withdraw

User send a "withdraw" ambient to the Bank. Bank creates a Money ambient which is an electronic money and packs it into an envelope named BankKey. User can open the BankKey to verify that the sub-ambient, money, was created by the real Bank.

```
ambient User{
   ActiveBehaviour(Withdraw, User, Bank,
           GO, Withdraw, BackFromBank);
   |{
      check(BackFromBank,Withdraw, BankKey);
   }
}
ambient Bank{
   ambient BWithdraw{
      in Withdraw ;
      ambient BankKey{
         out BWithdraw ;
         BeGO(BankKey, GO);
         ambient MONEY{
            in Pay ; BeGO ( MONEY, GO );
            in SPay ; open GO ;
            in Deposit ; BeGO ( MONEY, GO );
            in BDeposit ;
         };
      };
      out Withdraw ;
   }
}
```

4.3: Shopping

User sends a "Pay" ambient which has the money to the Merchant. Merchant creates a Goods ambient which packs the electronic goods bought by User. User can open the Goods to verify that the ambient is legal. Goods and Pay should be two keys negotiated by User and merchant.

```
ambient User{
   ActiveBehaviour(Pay, User, Shop,
                   GO, Pay, BackFromShop);
   |{ check(BackFromShop, Pay); }
}
ambient Shop{
   ambient SPay{
      in Pay;
      OutBeGO(GOOD, SPay);
      out Pay;
      ambient GO{
         in MONEY;
         out SPay;
      }
   }
}
```

4.4: Deposit

Merchant sends a "Deposit" ambient which has "money" inside to the Bank. Bank tries to open money to verify the money is created by itself. Then Bank sends a Receipt to Merchant to complete this transaction.

```
ambient Shop{
   ActiveBehaviour(Deposit, Shop,
                   Bank, GO, Deposit, BackFromB);
   |{ check(BackFromB,Deposit,Receipt); }
}
ambient Bank{
   ambient BDeposit {
      in Deposit;
      open Money;
      OutBeGO(Receipt, BDeposit);
      out Deposit;
   }
}
```

4.5: Discussion

In this scenario, several MAMN features are shown.

- ✓ Code mobility is approved.
- ✓ Repeatedly behaviours can be defined as macro, even though they belong to different servers or owners Macro is used to shorten the model. When the MAMN model is converted into a Java Ambient model to execute, the source code of macro is expanded at where the macros are used.
- ✓ "Naming" method can be seen as a unique identity. For instance, ambient "Money" can be verified by "Bank" during deposit, because "Money" is

generated by “Bank”, which keeps the stamp of money. Thus, “open Money” is such as process to verify.

- ✓ Ambient “BankKey” can be seen as a signature of Bank. Thus, what macro “check” does is to verify (“open”) locked ambient to satisfy the legality of the message. It also satisfies the requirement of nonrepudiation.

5: CONCLUSION

MAMN is a modelling language which can be used to model most distributed applications. An ambient in MAMN notation can be seen as a mobile process or an unmovable node which might be a server, a computer or a mobile device, even an endpoint of a web service. Thus, MAMNcan also be used in web-service based applications. The benefit of MAMN is that security is self-contained. An implementation using Java Remote Method Invocation (Java RMI) has been designed to prove that MAMN can also be practical. However, there is a potential inadequacy. As observed in earlier sections, MAMN components define the actions in terms of active mobility. This is complicated when the scenario is complex, such as in the e-payment systems or grid virtual organization solutions. Thus, a proper modification might have to be made to adapt MAMN to the new technology, such as passive message transmission. However, MAMN is still a simple and straightforward modelling language for distributed environment and it is suitable for students to know distributed systems.

References

[1] Wikipedia, on-line encyclopedia.

[2] L. Cardelli. Mobile ambient synchronization. Src technical note, July 1997.

[3] L. Cardelli. Abstractions for Mobile Computation. Technical report, Microsoft Research Technical Report MSR-TR- 98-34, July 1998.

[4] E. Clarke. Model Checking. The MIT Press, 1999.

[5] R. E. Filman and D. P. Friedman. Coordinated Computing : tools and techniques for distributed software. New York: McGraw-Hill Book Company, 1984.

[6] O. M. Herescu and C. Palamidessi. Probabilistic asynchronous pi-calculus. In Foundations of Software Science and Computation Structure, pages 146–160, 2000.

[7] C. Hoare. Communicating Sequential Processes. Prentice Hall International, 1985.

[8] C. A. R. Hoare. Communicating sequential processes. Communications of the ACM, 21(8), Aug. 1978.

[9] Y. Lee. MODELS OF WORKFLOW IN GRID SYSTEMS - WITH APPLICATIONS TO SECURITY AND MOBILE CODE. PhD thesis, Southampton, UK, June 2006.

[10] R. Milner. A Calculus of Communicating Systems. Springer Verlag.

[11] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, Logic and Algebra of Specification, pages 203–246. Springer-Verlag, 1993.

[12] R. Milner. Communicating and Mobile Systems: the pi-Calculus. Cambridge University Press, 1999.

[13] Raja and Shyamasundar. Mobile Computation: Calculus and Languages (a tutorial). In CSC: Asian Computing Science Conference, LNCS, 1998.