A Language for Description and Verification of Mobile Agent Algorithms

Xuhui Li
The State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China
lixuhui@whu.edu.cn

Jiannong Cao
Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong
csjcao@comp.polyu.edu.hk

Yanxiang He
The State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China
yxhe@whu.edu.cn

Abstract

Mobile agent technology has been widely adopted in network computing, whereas it remains a problem to design and verify the mobile agent algorithms in a platform independent way. In this paper, we propose a script language called SMAL to design the mobile agent algorithm. The semantics of SMAL is briefly introduced with its execution model. Furthermore, to verify the agent program in SMAL, a transformation function for converting SMAL program to Mobile UNITY specification is presented, which would facilitate making use of UNITY-logic to prove the correctness properties of the program.

1. Introduction

Mobile agent, a software entity itinerating in the network to accomplish its scheduled tasks, has attracted researchers in distributed computing for its features such as reactivity, autonomy and mobility [1]. However, the concept has different meaning for different people. Some people simply refer it as a mobile software entity in essays, some think it essentially as a combination of mobile processes in theory, and some treat it as a program running in and migrating across certain platforms called "hosts".

The variance in understanding the concept leads to the variance in describing function and, further, the semantics of the mobile agents. Therefore, there is not a standard definition for mobile agent and its semantics. The platform-specific agent programs cannot describe an algorithm based on mobile agent running in other platforms, let alone its verification. Whereas theorists proposed some theoretic models to describe the semantics of mobile processes and the verification of the algorithms, such as π-Calculus [2] and Mobile UNITY [3,4], the models often present a fine-grained architecture where a concrete algorithm would exhibit a fairly trivial and lengthy form.

The lack of standard description makes it difficult to unambiguously design and understand the mobile agent algorithm and naturally makes the correctness verification of mobile computation a complicated problem. Although some formal approaches, among which Mobile UNITY is the most notable one, have been proposed, it is not easy to directly apply the formal tools to agent algorithm.

The fact that the researches on designing, implementing and verifying the mobile agent algorithms are isolated has actually embarrassed the improvement of mobile agent technology. In this paper, we try to find a solution by proposing a script language to overcome the embarrassment. The language named SMAL is based on a simple execution model defined to describe the semantics of mobile agent's behaviors. The standard and explicit semantics enables the transformation from an algorithm's specification in SMAL to its representation in formal tools such as Mobile UNITY for verification. By presenting a transformation approach, the gap between algorithm's design and verification can be filled.

The rest of the paper is organized as follows. In Section 2, the SMAL language is formally presented and its semantics is briefly introduced with the execution model. In Section 3, the approach of transforming the SMAL programs to its Mobile UNITY representation is introduced, and a simple example is given. Finally, Section 4 concludes the paper.

This research is partially supported by following grants: 863 Foundation of China (No.2002AA4Z3450) and Research Fund for the Doctoral Program of Higher Education (20010486029), University Grant Council of Hong Kong under the CERG Grant B-Q518 and the Polytechnic University under HK PolyU Research Grant A-PD54.

Proceedings of the Fourth International Conference on Computer and Information Technology (CIT'04)
0-7695-2216-5/04 $20.00 © 2004 IEEE

2. SMAL Language: A Language Describing Mobile Agents

2.1 SMAL 's Syntax

As mentioned above, mobile agent systems always support a programming language for users to write their own agent programs. However, the difference between the implementation languages of agent systems would result in different execution models of agents. Additionally, the difference between concrete agent programs and programs written in Mobile UNITY is distinct too. Based on deep exploration of well-rounded mobile agent systems such as IBM Aglets[5] and Mole[6], we propose a script language called SMAL (Simple Mobile Agent Language) to formulate mobile agent algorithms. With a formal syntax and explicitly description on its execution model, programmers can easily and unambiguously understand algorithms written in SMAL.

```
   <AGENT> ::= Agent Agent_Declaration_Name
Named Agent_Name At Host_Name <BODY>
   <BODY> ::= Body <DECLARATION> <INIT>
<MIGRATION> <MESSAGE>+ BodyEnd
   <DECLARATION> ::= Declaration < ADECL_ITEM>+
   <ADECL_ITEM> ::= Lock_Variable_Name : <LTYPE>;
| <DECL_ITEM>
   <DECL_ITEM> ::= Variable_Name : <DTYPE>;
   <DTYPE> ::= <VTYPE> | Record_Type_Name
   <VTYPE> ::= Int | Bool
   <LTYPE> ::= Lock
   <RTYPEDEF> ::= Record Record_Type_Name
RecordBegin <RECORDFIELD>+ RecordEnd
   <RECORDFILED> ::= FiledName : <VTYPE>
   <INIT> ::= Initilization <STATEMENTBLOCK>
   <MIGRATION> ::= MigrationRecover
<STATEMENTBLOCK>
   <MESSAGE> ::= On Message Message_Name With
Message_Argument : <VTYPE> Do <STATEBLOCK>;
   <STATEMENTBLOCK> ::= [ BlockDeclaration
<DECL_ITEM>+ ] BlockBegin <STATEMENT>+ BlockEnd
   <STATEMENT> ::=  <Int_Variable> = <INTEXP> |
<Bool_Variable> = <BOOLEXP> |
   if ( <BOOLEXP> ) Begin <STATEMENT>+ End
   else Begin <STATEMENT> + End |
   while ( <BOOLEXP> ) Begin <STATEMENT>+ End |
   lock( Lock_Name) Begin <STATEMENT>+ End
unlock( Lock_Name) |<PRIMITIVE>
   <PRIMITIVE> ::= sendmessage( Agent_Name,
Message_Name, <VEXP> ) | migrateto( Host_Name )
|createagent( Agent_Name,
Agent_Declaration_Name, Host_Name ) |
regainmessage() | removeMessage() |
blockmessage() | unblockmessage() | dispose()
   <VEXP> ::= <BOOLEXP> | <INTEXP>
   <BOOLEXP> ::= true | false | <Bool_Variable>
| (Not <BOOLEXP>) | (<BOOLEXP> And <BOOLEXP>) |
(<BOOLEXP> Or <BOOLEXP>) | (<INTEXP> > <INTEXP>)
| (<INTEXP> == <INTEXP>)
   <INTEXP> ::= integers | <Int_Variable> |
<INTEXP> + <INTEXP> | <INTEXP> - <INTEXP> |
<INTEXP> * <INTEXP> | <INTEXP> / <INTEXP> |
<INTEXP> % <INTEXP>
   <Int_Variable> ::= Int_Variable_Name |
Record_Variable_Name . Int_Field_Name
   <Bool_Variable> ::= Bool_Variable_Name |
Record_Variable_Name . Bool_Field_Name
```

Figure 1. Syntax of SMAL

The BNF syntax of SMAL is listed in Figure 1. It describes the composition rules of agent programs. For simplicity, the definitions of some trivial items such as variables are not presented, which would not incur misunderstanding. Obviously SMAL is a simple program script language and has the same expressive power as normal programming languages.

As Figure 1 depicts, an agent is presented as a program with four sections: *declaration, initialization*, *migration recovery* and *message handling*. The variables are declared in the *declaration* section, and the other three sections are all composed of the statement blocks that define the behaviors of the agent in each state and are executed by the threads scheduled properly. Statements in SMAL programs are composed of the assignment statements, conditional statements, loop statements and primitives. The former three kinds of statements make it convenient to describe the algorithm with common pseudocode programs. The primitives are deeply related with the threads' execution and interaction that is the essential part of SMAL. The details of the language and its semantics would be introduced in the following subsection.

2.2 Execution Model of SMAL

As a script language, SMAL has a formal semantics explicitly presented with a set of π-Calculus processes denoting each syntactical element in the program [7]. However, the semantical rules for SMAL is too complicated and trivial to be stated here, therefore we use the execution model instead of formal process algebra to explain the semantics of SMAL.

Generally, in SMAL agent program three kinds of variables can be declared: the first one is the agent state variables which can be used to maintain the states of the agents and would not be changed during the migration; the second is lock variables used to indicate the exclusive critical region in the statement blocks of the agent; the third one is the local variables declared in statement blocks. Since SMAL follows the weak migration scheme, the execution state of the agents such as the call stack and the local variables in statement blocks would not be preserved during the migration, while the value of the agent state variables would remain.

SMAL agent drives a set of threads to execute the statement blocks. When the agent is created, an initialization thread executes the statement block defined in the *initialization* section, so the variables can be initialized there. When agent's migration is

accomplished, a thread executes the block in the *migration recovery* section too. Once the initialization thread or the recovery thread accomplishes its work, the agent would be in working state and ready to handle the incoming messages. Generally, two kinds of threads, manager thread and worker thread, are involved in message handling. Manager thread runs in background and schedules tasks undertaken by worker threads. The tasks are defined in the *message handling* section that consists of a set of statement blocks each of which defines how to handle a certain message and is executed by a worker thread. Thus, a single thread under the control of the manager thread handles a message.

In SMAL, messages are sent in an asynchronous way, so the agent would not receive the messages explicitly. Incoming messages are transferred by the agent platforms and forwarded to the message manager thread. The message manager thread uses two message queues, a *processing* queue and a *waiting* queue, to accommodate the messages being handled or not currently. To control the message management in the user-defined statements, four primitives, *blockmessage*, *unblockmessage*, *regainmessage* and *removemessage,* are provided in the statement blocks. The *blockmessage* and *unblockmessage* primitives are used to set the *blocked* flag in an agent. Once a message is received it would be stored in the *waiting* queue. If the *blocked* flag were *false*, the message would automatically be put into the *processing* queue and then a message thread would be started to execute the statement block specified to handle this message. Otherwise, the message would stay in the *waiting* queue until the *blocked* flag is changed into *false*. On default the *blocked* flag is set to true so that the incoming messages would be stored in the *waiting* queue waiting for the end of initialization. The *regainmessage* primitive moves the messages in the *waiting* queue to the *processing* queue, while the *removemessage* primitive removes the current message from the *waiting* queue. These two primitives are used to ensure that the messages are correctly handled even when the work would be interrupted due to migration.

In the statement blocks, some primitives are provided for common agent behaviors. They are *sendmessage*, *createagent*, *migrateto* and *dispose.* The semantics of the primitives are apparent. It should be noted that once *migrate* primitive is executed the threads handling the messages are all interrupted. In the former case, the agent would wait for the time out of suspension. When the agent migrates to the target host, the *migration recovery* statements are executed and then the messages in the *waiting* queue are handled.

Since the threads are running concurrently, it is vital to guarantee that no conflict among the threads such as deadlock would occur. Therefore, the *lock* and *unlock* primitives are used to make the threads execute in the order as the user prefers to. Once a lock statement, e.g., *lock(r),* is executed by one thread, representing it would require the control of a lock variable *r* declared in the *declaration* section, the other threads would be suspended if they encounter a *lock(r)* until the statement *unlock(r)* is executed by the thread who have acquired the control of *r*. Each lock variable, say, *r*, is associated with a thread queue containing the threads waiting for the control of *r*. Thus, the worker threads of an agent can run appropriately following the *lock* and *unlock* primitives in the statement blocks. Providing the *lock* and *unlock* primitives, SMAL gives the users full control over thread's execution whereas leaves the problem of concurrently accessing the agent's shared area to the users. It is all users' duty to make sure his program would not incur concurrency problem.

A simple example of SMAL agent is listed as Figure 2:

```
Agent a Named Agent_Name at Host_Name
Body
Declaration
  aint: Int;
Initialization
  BlockBegin
   createagent(b1, b, h2);
   aint := 0;
   sendmessage(self, run, aint);
   unblockmessage();
  BlockEnd
On Message run with init: Int Do
  BlockBegin
   sendmessage(b1, update, init);
  BlockEnd
BodyEnd

Agent b Named Agent_Name at Host_Name
Body
Declaration
  bint: Int;
Initialization
  BlockBegin
    unblockmessage();
  BlockEnd
On Message update with value: Int Do
  BlockBegin
   bint := value;
  BlockEnd
BodyEnd
```

Figure 2. An Example of SMAL Agent

In the example, once an instance of agent program *a* is created, it would create an agent named *b1* in host *h2*, and send a *run* message to itself to launch the work. Then the *unblockmessage* primitive notify the agent to begin handling the messages, which would start a worker thread dealing with the *run* message and

sending an integer to *b1*. As to the agent *b1*, it would be ready to handle the incoming messages if the initialization is finished. As the message from *a*'s instance comes, the value of *bint* in *b1* is set to the one of message's argument. This example would be further discussed in later section.

SMAL presents a simple scheme to specify the mobile agent algorithm based on asynchronous message passing. Different from other formal language describing mobile computation such as π-Calculus and Mobile UNITY, SMAL adopts a high-level language form to present mobile agent algorithms, which undoubtedly facilitates the design of the algorithms. However, the theoretic achievements in mobile computation, i.e., the verification rules in Mobile UNITY, cannot be directly used in SMAL, which pushes us looking for an approach to transform SMAL programs to other formal representation so as to take advantage of those existed formal tools.

3. SMAL-Mobile UNITY Transformation

Mobile agent algorithm is easy to present in SMAL language, but it is difficult to verify its correctness. Fortunately, Mobile UNITY has set out a complete set of rules for verification. Therefore, a shortcut to verify the algorithm in SMAL is to translate it into an equivalent program in Mobile UNITY, and then utilize the UNITY-logic [8] to prove the properties indicating the correctness of the algorithm.

A Mobile UNITY system comprises a set of ***components*** that are activated ***programs*** and the ***interactions*** among the ***components***. The specification of a program consists of four sections, ***declare***, ***always***, ***initially*** and ***assign***, acting as the common elements of a normal programming language do. The variables are declared in the ***declare*** section and the initial values are assigned to them in the ***initially*** section, the constants and macro notations are defined in the ***always*** section for simplification of the program. In the ***assign*** section, the major part of the program, the statements representing the actions of the program are listed. The statements adopting a guard-statement style can be executed concurrently as soon as the predicate in its guard is satisfied. However, which statement is chosen to run is not definite when there are several statements available. Readers can refer to [3] for the details of Mobile UNITY.

As a reactive formal tool describing mobile computing, Mobile UNITY differs a lot from SMAL either in syntax or in semantics. For example, Mobile UNITY enables the strong migration because the program would maintain its states while it changes the location variable λ. Additionally, every statement in a Mobile UNITY program has a chance to run once the guard condition is satisfied and, regarding the rules called "strong fairness" in reactive models, each statement would be executed eventually unless its guard condition is no longer satisfied. On the contrary, the statements in SMAL are usually executed sequentially. Generally, the elements of SMAL can be translated into Mobile UNITY statements, but there are still some features of SMAL such as multithreading cannot be identically interpreted, which make us have to find an alternative to simulate them.

3.1. Transformation of Variables

Variables declared in SMAL program are composed of agent's global variables and local variables in statement blocks. Agents would only maintain the states of global variables. These variables can be correspondingly transformed into the variables in Mobile UNITY; however, the transformation method varies with the variables' data types and locations. SMAL has a type system supporting two simple value types, user-defined record types and a *Lock* type, whereas Mobile UNITY just adopts some data types on default. However, Mobile UNITY permits the free definition and use of record types, queue types and array types, which makes the data types in UNITY powerful enough to represent the ones in SMAL.

The conversion of global variables is fairly simple. For the variables with value types, *Integer* and *Boolean*, are directly supported by Mobile UNITY, all the effort needed is to add corresponding variables in **declare** section. Usually, to facilitate differentiating the variables, a naming policy is used in converting the names. For example, under the guidance of a certain naming policy, the declaration of a global *Int* variable named *count* in agent program *task1* might be changed into *task1_global_count: integer* in the corresponding program specification in Mobile UNITY. Since the *Lock* variables in SMAL always concern the allocation of resources, a *Bool* variable indicating the occupation of the resource and a variable queue of Thread type (a record type representing a thread) indicating the threads waiting for the resources are required to represent the variables with the *Lock* type in Mobile UNITY.

Variables declared in statement blocks of *initialize* and *message recovery* can follow the similar conversion rules above because they can be treated as static variables too. However, it is more complicated to convert the ones in message handling sections. As a notation language whose major purpose is to verify the properties, Mobile UNITY didn't enable the dynamical creation of variables, which is the basis of SMAL's multithreading mechanism. As each message

handling thread owns a replication of local variables, we have to declare a set of variables for each thread in the Mobile UNITY representation. For simplicity, we can declare an array for each variable in a message handling section, e.g., the array *Msg_A_b[N]:integer* represents the conversion of a variable *b: Integer* in *On Message A* section. The range of the array is the maximal count allowed for concurrently running message-handling threads. Further, to make sure the variables would not interfere with each other, some statements are required in the **assign** section to assist handling the elements in variable arrays.

3.2. Transformation of Statements

The reactive model in Mobile UNITY permits the statements executed in a concurrently way, while the statements in a block in SMAL can only run sequentially. To represent the statement blocks in Mobile UNITY, some mechanisms are conceived to make the concurrent statements running in a sequential way. The steps to transform the statements can be discussed briefly as follows.

Firstly, each SMAL statement would be assigned a unique label, and we denote the set of the labels as L with a function next: L→L. Next (l_1) = l_2 indicates that once the statement labeled l_1 has been executed, the statement labeled l_2 in SMAL would be the next one to execute. Therefore, in **declare** section of Mobile UNITY specification, we can declare a label variable for each statement in *initialize* and *migration recovery* due to their uniqueness, and declare an array of label variables for each statement in message handling section for simulating multithreading.

Secondly, the statements in SMAL are all transformed into the reactive assignment statements with the assistance of label variables. For example, a statement $I = I + 1$ with the label l can be transformed to be I_A, $l_A := I_A + 1$, *next(l)* ***if*** $l_A = l$, here l_A is a variable performing the function of a program counter. Similarly, conditional statements and loop statements can also be transformed with some preprocessing, e.g., loop statements can be converted to conditional statements by introducing a *goto* statement, and the Boolean condition *b* in a *if(b)* can be directly put into statements' guard.

Thirdly, the primitives and associated mechanism in SMAL would be described in Mobile UNITY. The message queues and the management primitives would be translated into UNITY's variable declarations and statements. Meanwhile, there are still some primitives such as creation and communication that would not be transformed. Although it is feasible to transform them, we would used these primitives directly as default functions and add some logic rules accordingly, which would make the specification compact.

Obviously, the transformation is to interpret the SMAL program in Mobile UNITY. In other words, it is the semantics specification of SMAL program in Mobile UNITY. To explicitly explain the transformation approach, we would use a function to formally describe the transformation of each element of the program.

3.3. MU: Transformation Function for SMAL to Mobile UNITY

Under the guidance above, we defined a transformation function MU to translate a SMAL program to a semantically equivalent Mobile UNITY program. MU defines all the rules needed to transform the syntax elements of SMAL, however, here we can only introduce the major parts of the definition and ignore the trivial ones. In MU, some auxiliary functions, *declare*, *always*, *initially* and *assign*, and an operator ⊕ are introduced for facilitating the description. The purpose of the notations is to combine the contents in corresponding sections, e.g., A ⊕ declare[B] means add B into A's declare section.

3.3.1. Agent framework. It is fundamental to create a framework of Mobile UNITY program when we want to transform the concrete elements. The framework includes the agent program structure and the implementation of message management.

```
MU[Agent Agent_Declaration_Name Named
Agent_Name At Host_Name <BODY>] ≡
  ( program Agent_Declaration_Name(Agent_Name,
Host_Name) at λ declare… initially… always…
assign… end ) ⊕
  MU[default] ⊕ MU[<BODY>]
```

MU[default] defines the variables and statements involving message queues, message management, migration and dispose in generating the framework.

```
MU[default] ≡
declare[ … <variables>… ] ⊕
always[Message ≡ (type, arg) ] ⊕
initially[ … <variables_initialization>…] ⊕
assign[ waitingqueue, messagebuffer :=
waitingqueue.head(messagebuffer),
tail(messagebuffer)  reacts-to messagebuffer ≠ ε
∧ lively = true
 processingqueue, waitingqueue, n, thread_msg
[n], l_msg[n] :=
processingqueue.head(waitingqueue),
tail(waitingqueue), pick(msg), true, startl(msg)
if  waitingqueue≠ε ∧ head(waitingqueue).type =
msg ∧ messageblocked = false ∧ lively = true
 waitingqueue, processingqueue :=
waitingqueue.head(processingqueue),
tail(processingqueue)  reacts-to
processingqueue ≠ ε ∧ regainbegin = true ∧ lively
= true
```

□ regainbegin := false reacts-to processingqueue = ε ∧ regainbegin = true ∧ lively = true

□ processingqueue, removebegin := ε, false reacts-to removebegin = true ∧ lively = true

□ < 0≤I<N, msg∈MSG:: □ $thread_{msg}$[i] := false reacts-to migrating = true ∧ $thread_{msg}$[i]=true ∧ lively = true >

□ < lock_name∈Lock :: □ $LockFlag_{lock\_name}$, $LockQueue_{lock\_name}$:= false, ε reacts-to migrating = true ∧ $LockFlag_{lock\_name}$ = true ∧ lively = true >

□ migrating, $thread_{migraterecover}$, $l_{migraterecover}$:= false, true, start(migratereover) if migrating = false ∧ $thread_{migraterecover}$ = false ∧ lively = true]

3.3.2. Variables declaration. Based on the methods previously mentioned, MU renames the variables and adds them to the **declare** section.

MU[Declaration < ADECL_ITEM>+] ≡ ⊕ $MU_{declare}$[< ADECL_ITEM>]

For agent variables with value types, the original names are transformed by a function *tranfromname*, and the value types would remain. For Lock variables, a flag variable with Boolean type and a thread queue variable are declared.

$MU_{declare}$[Int_Variable: Int] ≡
declare[transformname(Int_Variable, Agent) : integer]

$MU_{declare}$[Bool_Variable: Bool] ≡
declare[transfromname(Bool_Variable, Agent) : boolean]

$MU_{declare}$[Record_Variable: RecordTypeName] ≡
declare[transfromname(Record_Variable, Agent) : RecordTypeName]

$MU_{declare}$[Lock_variable: Lock] ≡
declare[$LockFlag_{Lock\_variable}$: Boolean □ $LockQueue_{Lock\_variable}$: queue of Thread]

The variables in statement blocks of message handling sections are often defined as arrays, i.e.,

MU[BlockDeclaration < DECL_ITEM>+, BlockName] ≡ ⊕ $MU_{BlockDeclare}$[<DECL_ITEM>, BlockName]

$MU_{BlockDeclare}$[Int_Variable: Int, Msg_Name] ≡ declare[transformname(Int_Variable, Msg_Name) : array[N] of integer]

Other definition on variables can be easily inferred according to the examples above.

3.3.3. Statement blocks. Transformation of the statement blocks in SMAL is the key point of MU. Here we present the details of transforming message-handling sections as the representation that involve all the techniques required. Generally, transformation of a message-handling section includes a variables declaration and statements transformation, as follow shows.

MU[On Message Msg_Name With Msg_Arg: <MTYPE> Do <STATEMENTBLOCK>] ≡
$MU_{BlockDeclare}$[Msg_Arg:<MTYPE>, Msg_Name] ⊕
declare[l_{Msg_Name}:array[N] of Label □ $thread_{Msg\_Name}$: array[N] of boolean] ⊕
initially[<0 ≤ I < N :: $thread_{msg\_Name}$[I] := false] ⊕ $MU_{message}$[<STATEMENTBLOCK>, Msg_Name]
$MU_{message}$[BlockDeclaration < DECL_ITEM>+ BlockBegin<STATEMENT>+ BlockEnd, Msg_Name] ≡ MU[BlockDeclaration < DECL_ITEM>+, Msg_Name] ⊕
MU_{label}[label(BlockBegin<STATEMENT>+BlockEnd, Msg_Name), Msg_Name]

Where MU_{label} would transform the statements labeled by the *label* function. A common assignment statement can be directly converted into a UNITY statement, i.e.,

MU_{label}[l_1: *Int_Variable* = <INTEXP>, Msg_Name] ≡ assign[< 0 ≤ I < N ::
□ transformname(*Int_Variable,* Msg_Name)[I], l_{Msg_Name}[I] := transformexp(<INTEXP>, Msg_Name, i), next(l_1) if l_{Msg_Name}[I] = l_1∧ $thread_{Msg\_Name}$[I] = true ∧ lively = true >]

The guard conditions are utilized to specify the direction of the next statement in the conditional statements and loop statements.

MU_{label}[l_1: if (<BEXP>) l_2:Begin…End l_3:Else l_4:Begin…End, Msg_Name] ≡ assign[< 0 ≤ I < N ::
□ l_{Msg_Name}[I] := l_2 if l_{Msg_Name}[I] = l_1 ∧ transform(<BEXP>, Msg_Name, I) ∧ $thread_{Msg\_Name}$[I] = true ∧ lively = true
□ l_{Msg_Name}[I] := l_4 if l_{Msg_Name}[I] = l_1 ∧ not(transform(<BEXP>, Msg_Name, I)) ∧ $thread_{Msg\_Name}$[I] = true] ∧ lively = true

MU_{label}[l_1: while(<BEXP>) l_2:Begin… l_3:End, Msg_Name] ≡ assign[< 0 ≤ I < N ::
□ l_{Msg_Name}[I] := l_2 if l_{Msg_Name}[I] = l_1 ∧ transform(<BEXP>, Msg_Name, i) ∧ $thread_{Msg\_Name}$[I] = true ∧ lively = true
□ l_{Msg_Name}[I] := next(l_3) if l_{Msg_Name}[I] = l_1 ∧ not(transform(<BEXP>, Msg_Name, i)) ∧ $thread_{Msg\_Name}$[I] = true ∧ lively = true
□ l_{Msg_Name}[I] := l_1 if l_{Msg_Name}[I] = l_3 ∧ $thread_{Msg\_Name}$[I] = true ∧ livel=true >]

Some primitives such as lock, migration, dispose and message management primitives can be simulated in Mobile UNITY. In *lock* and *unlock* statements current status of the Lock is required to determine the guard conditions. In migration statements, the *migrating* flag is set to be true and the value of the location variable λ is reset, indicating the migration's taking place. Dispose is similarly handled. Blockmessage and unblockmessage primitives can be easily transformed, while removemessage and regainmessage primitives would incur more works.

When the execution of a message-handling block is over, the status of the thread is set to false.

MU_{label}[l_1:lock(Lock_Name), Msg_Name] ≡
assign[< 0 ≤ I < N ::
□ l_{Msg_Name}[I], $LockFlag_{Lock\_Name}$:= next(l_1), true if l_{Msg_Name}[I] = l_1 ∧ $LockFlag_{Lock\_Name}$ = false ∧ $thread_{Msg\_Name}$[I] = true ∧ lively = true
□ $thread_{Msg\_Name}$[I], $LockQueue_{Lock\_Name}$:= false, $LockQueue_{Lock\_Name}$• (Msg_Name, I) if l_{Msg_Name}[I] = l_1∧ $LockFlag_{Lock\_Name}$ = true ∧ $thread_{Msg\_Name}$[I] = true ∧

```
lively = true]
   MU_lable [l1:unlock(Lock_Name), Msg_Name] ≡
     assign[< 0 ≤ I < N ::
     □ l_Msg_Name[I], LockFlag_Lock_Name := next(l1),
false  if l_Msg_Name[I] = l1 ∧ LockQueue_Lock_Name = ε ∧
thread_Msg_Name[I] = true ∧ lively = true
     □ l_Msg_Name[I], thread_Msg_Name1[j],
LockQueue_Lock_Name := next(l1), true,
tail(LockQueue_Lock_Name)  if l_Msg_Name[I] = l1 ∧
LockQueue_Lock_Name = ε ∧ head(LockQueue_Lock_Name ) =
(Msg_Name1, j) ∧ thread_Msg_Name[I] = true ∧ lively
= true ]
   MU_label[l1:MigrateTo(Host_Name), Msg_Name] ≡
     assign[< 0 ≤ I < N ::
     □ λ, migrating := λ_Host_Name , true if l_Msg_Name[I]
= l1∧ thread_Msg_Name[I] = true ∧ lively = true > ]
   MU_label[l1:Dispose()] ≡
     assign[< 0 ≤ I < N ::
     □ lively := false if l_Msg_Name[I] = l1∧
thread_Msg_Name[I] = true ∧ lively = true > ]
   MU_label[l: blockmessage(), Msg_Name] ≡
     assign[< 0 ≤ I < N ::
     □ l_Msg_Name[I], messageblocked := next(l1),
true  if l_Msg_Name[I] = l1 ∧ thread_Msg_Name[I] = true
∧ lively = true]
   MU_label[l: regainmessage(), Msg_Name] ≡
     assign[< 0 ≤ I < N ::
     □ l_Msg_Name[I], regainbegin := next(l1), true
if l_Msg_Name[I] = l1 ∧ thread_Msg_Name[I] = true ∧ lively
= true >]
   MU_label[l: removemessage(), Msg_Name] ≡
     assign[< 0 ≤ I < N ::
     □ l_Msg_Name[I], removebegin := next(l1), true
if l_Msg_Name[I] = l1 ∧ thread_Msg_Name[I] = true ∧ lively
= true >]
   MU_label[l1: BlockEnd, Msg_Name] ≡
   assign[< 0 ≤ I < N ::
     □ thread_Msg_Name[I] := false if l_Msg_Name[I] = l1∧
thread_Msg_Name[I] = true ∧ lively = true > ]
```

The partial definition of MU depicted above shows almost of the techniques required for presenting a SMAL agent program in Mobile UNITY. Although there are still some statements not listed, it is not difficult to present them following the examples. However, it is still not enough for the verification in that we ignored the definition of some crucial primitives, i.e., creation and communication. The transformation of them is trivial and would cause redundant inference steps during the verification. Therefore, we introduce some inference rules with programs instead of transformation rules concerning the primitives to make the verification feasible.

3. 4. Auxiliary Programs and Inference Rules

In MU function the transformation methods for migration, lock and message management are specified, meanwhile we leave the creation and communication to the inference system rather than MU function. The difficulty in specifying the creation is similar to the embarrassment in the multithreading mechanism, that is, Mobile UNITY is lack of the means of presenting agents' dynamical creation. On the contrary, it is feasible to transform *sendmessage* as other primitives did, but the trivial specification would make the verification rather tedious. Therefore, we can simplify the procedures by directly introducing a simple auxiliary program and some inference rules that play as axiomatic semantics of the statements in the style of Hoare's triples.

Creation of an agent can be represented by the variation of the properties involving the *createagent* primitive. To facilitate the denotation, we use a notation *Initially(Agent_Program, Agent_Name, Host_Name)* to indicate the properties holding when an agent is just initialized, i.e., *Initially(agent, an, hn)* represents the disjunction of all the equations holding in the ***initially*** section of the program *agent* with the parameters *an* and *hn*. Thus, the inference rule about the creation would be

$$\frac{\neg(p \Rightarrow agent(an,hn).lively)}{\{p\} < createagent(agent, an, hn) > \{p \vee Initailly(agent, an, hn)\}}$$

The inference rule concerning the communication needs the assistance from a program called *network* implementing the communication transparent to the agents. And, in the ***interactions*** section of the system, the *in* queue in the *network* is directly coupled with the agents' waiting queue.

```
   program  network at λ
     declare
       in, out : array [ Nagents ] of queue of
message
     assign
     < □ i, j : 0 ≤ i, j < Nagents ::
         □ in[i], out[j] := in[i] • head(out[j]),
tail(out[j]) if antoint(out[j] ≠ ε ∧
head(out[j]).targetname) = i  >
   end
   interactions
   < □ i: 0 ≤ i < Nagents ::
     in[i] ≈ a.waitingqueue when antoint(a)=I ∧
a.lively=true >
```

And, the corresponding inference rule is:

$$\frac{p \Rightarrow a.lively = true}{\{p\} < a.sendmessage(msg) > \{p \vee rhead(out[anto\,\mathrm{int}(a)]) = msg\}}$$

In the programs and inference rule above, the function *rhead(q)* returns the element locating in the tail of the queue *q*, the function *antoint(a)* transform the agent into an integer which uniquely identifies an agent in the system. From the communication programs and rules, it is easy to deduce that the following rule holds:

```
   {p}a.sendmessage(b, mn, ma){q} ⇒
   q leads-to rhead(b.waitingqueue)=(b, mn, ma)∨
b.lively=false
```

Up to now, the framework of transformation for SMAL program has been established, and we would

like to exemplify the usage with the example listed above.

3. 5. Verification Example

According to the programs in Figure 2., it is easy to see that when an instance of agent *a* is created, there would be an agent named *b1* whose ariable *bint* would have the value 0. Assume that *a*'s instance is created by a statement *createagent(a1, a, h1)*, we can interpret the property as

Initially(a, a1, h1) **leads-to** b1.lively ∧ b1.bint = 0.

Once the agent programs' specifications in Mobile UNITY are figured out with the definition of MU function, it is not difficult to deduce the auxiliary properties step by step, which would finally prove that the property above holds. These properties are:

(1) Initially(a,a1,h1) **leads-to** a1.lively ∧ ∃i∈Int (thread$_{run}$[i] ∧ l$_{run}$[i] = start(run)) ∧ b1.lively

(2) a1.lively ∧ ∃ i∈Int(thread$_{run}$[i] ∧l$_{run}$[i]= start(run)) ∧ b1.lively **leads-to** head(b1.waitingqueue) =(b1, update, 0) ∧ b1.lively

(3) head(b1.waitingqueue) =(b1, update, 0) ∧ b1.lively **leads-to** b1.lively ∧ b1.bint = 0

The example shows the feasibility and practicality of the design and verification of mobile agent algorithm using our approach. In fact, we have successfully tried some complicated algorithms to test the approach and got the anticipated results.

4. Conclusion

In this paper we proposed a language called SMAL for designing mobile agent algorithms. Adopting a high-level language style, SMAL enables the users easily figure out the algorithms for mobile computation. Further, an approach for converting the SMAL agent program to a semantically equivalent specification in Mobile UNITY is presented, which facilitates the correctness verification of SMAL program.

Now we are engaged in improving the SMAL language and the transformation approaches. Currently SMAL can only support asynchronous communication mode, which is not efficient in some circumstances where the synchronous communication is required. Therefore, in the future work we would add the features of synchronous communication into SMAL and present the transformation rules accordingly. Besides these, the formal proof of soundness and completeness of the transformation approach is also an important topic to consider for developing a robust theoretical model to verify the algorithms.

References

[1] G. Cabri and L. Leonardi and F. Zambonelli, "Mobile Agent Technology: Current Trends And Perspectives", *Mobile Object Systems*, *Lecture Notes in Computer Science, No. 1222*, Springer Verlag (D), February 1997.

[2] R.Milner, "The Polyadic π-calculus: a Tutorial", In *Logic and Algebra of Specification,* Springer-Verlag, Berlin, pp.1-49.

[3] Peter J. McCann, Gruia-Catalin Roman, "Compositional Programming Abstractions for Mobile Computing", *IEEE Trans. On Software Engineering,* 24(2), 1998, pp.97-110.

[4] A.Fuggetta, G.P.Picco and G.Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering,* 24(5), 1998, pp.342-361.

[5] Ariv Aridor and Mitsuru Oshima. Infrastructure for Mobile Agents: Requirements and Design. In *Proceedings of 2nd International Workshop on Mobile Agents (MA '98)*, Springer Verlag, September 1998.

[6] J. Baumann, F. Hohl, K. Rothermel and M. Straßer, "Mole - Concepts of a Mobile Agent System", *World Wide Web*, 1(3), 1998, pp.123-137.

[7] Xuhui Li, *Research on the Problems of the Description and the Simulation of Mobile Agents*, Ph.D. Thesis, Wuhan University, 2003.

[8] Jayadev Misra, "A logic of Current Programming", *Journal of Computer and Software Engineering*, 3(2), 1995, pp. 239-300.

[9] L.Cardelli, A.D.Gordon. Mobile Ambients. In *Proceedings of Foundations of Software Science and Computation Structures*, ETAPS, Lisbon. 1998.

[10] Xuhui Li, Jiannong Cao, Yanxiang He. A Direct Execution Approach to Simulating Mobile Agent Algorithms. *The Journal of Supercomputing,* Vol.29, No.2, 2004.