THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A,
Volume 15, Number 1/2014, pp. 95–104

EXPRESSING MOBILE AMBIENTS IN TEMPORAL LOGIC OF ACTIONS

Bogdan AMAN, Gabriel CIOBANU

Institute of Computer Science, Romanian Academy, Iaşi Branch
Corresponding author: Gabriel CIOBANU, E-mail: gabriel@info.uaic.ro

Abstract text Temporal logic of actions is a logic for specifying and reasoning about concurrent systems, developed mainly for verification. Ambient calculus is a formalism for describing mobility and distributed computation. We express mobile ambients in temporal logic of actions, provide some results and illustrate the description through examples. Finally we give an implementation of mobile ambients in temporal logic of actions that can be used for the verification of their behaviour.

Key words: mobile ambients, temporal logic of actions.

1. INTRODUCTION

Temporal Logic of Actions (TLA) is a logic for specifying and reasoning about concurrent systems [7]; it was developed for the verification of complex concurrent systems and algorithms. A TLA specification defines a state machine. A TLA verification consists of taking two TLA specifications: one describing a simple state machine (“specification”), another describing a more detailed machine (“implementation”). Then, it is checked whether the implementation (machine) simulates the specification (machine).

Ambient calculus is a formalism introduced in [2] for describing mobility in distributed computation. In contrast to the π-calculus [10] whose computational model is based on the notion of *communication*, the ambient calculus is based on the notion of *movement*. A mobile ambient represents both a location and a unit of movement. Ambient mobility is controlled by capabilities: *in*, *out* and *open*. Mobile ambients can be adequately described by their behaviours, that is their possible sequences of movements/capabilities. Intuitively, a safety property asserts that something bad does not happen, and a liveness property asserts that something good does eventually happen. In specifying mobile ambients, a safety property might assert that a movement can take place only if some preconditions are fulfilled, and a liveness property might assert that a movement eventually takes place if its preconditions are fulfilled.

A rather specific logic for mobile ambients was defined in [3], in which the models are given by mobile ambients. In this logic, two process formulas can distinguish between two mobile ambients based on their ambient structure. In this paper we follow a different approach and present a logic, without spatial modalities, that will enable us to specify and analyse the behaviour of mobile systems as a program acting over a memory representation. In the specification of a mobile ambient, the program describes the sequences of actions by which a modification of the system takes place, ensuring the safety property such that the correct actions take place. Additional fairness constraints assert that certain actions might eventually occur, ensuring the liveness property such that actions which should take place, do eventually take place.

A system is specified in TLA by a formula describing a set of permitted behaviours; the behaviours permitted by a specification are described as sequences of states satisfying a safety or liveness property [1]. We are dealing with specifications in which the safety property is described by an “abstract” nondeterministic program, and we say that a behaviour satisfies the safety property if it can be generated by that program. We represent a program by a state machine which can have an infinite set of states, together with additional fairness constraints.

The structure of the paper is as follows. Section 2 briefly presents TLA and its semantics. Section 3 gives a brief presentation of mobile ambients. A description of mobile ambients by using TLA is presented in Section 4, followed by an example, while an implementation in TLA is presented in Section 5. Conclusion and references end the paper.

2. TEMPORAL LOGIC OF ACTIONS

The temporal logic of actions is built on both a logic of actions and a logic for reasoning about actions. The logic of actions is used for writing predicates (boolean expressions containing constants and variables), state functions (non-boolean expressions containing constants and variables) and actions (boolean expressions containing constants, variables, and primed variables). The syntax and semantics of TLA, along with additional notations used to write TLA formulas are summarised in what follows; more details in [7].

Syntax

$\langle formula \rangle$	=	$\langle predicate \rangle \mid \Box \left[\langle action \rangle \right]_{\langle state\ \ function \rangle}$
		$\mid \langle formula \rangle \wedge \langle formula \rangle \mid \neg \langle formula \rangle \mid \Box \langle formula \rangle$
$\langle action \rangle$	=	boolean-valued expression containing constant symbols, variables and primed variables
$\langle predicate \rangle$	=	$\langle action \rangle$ with no primed variables | $Enabled \langle action \rangle$
$\langle state\ function \rangle$	=	non-boolean expressions containing constant symbols and variables

An action represents a relation between old and new states, where the unprimed variables (e.g., v) refer to the old state, and the primed variables (e.g., v') to the new state. A behaviour is an infinite sequence of states. The set of all behaviours is denoted by S^{∞}. If σ is a behaviour $s_0, s_1, \ldots$, then σ_i denotes the ith state s_i and σ^{+i} denotes the behaviour $\sigma_i, \sigma_{i+1}, \ldots$ obtained by cutting off the first i states of the sequence σ. A formula $\Box F$ asserts that F holds now and always in the future; for any formula F and behaviour σ by $[\![\Box F]\!](\sigma) = \forall i \geq 0 \mathbf{:} [\![F]\!](\sigma^{+i})$ is denoted the fact that formula F holds at a certain time if and only if F holds for an infinite behaviour starting at that time.

We denote a state functions by f, variables by v, and use $(\forall v \mathbf{:} x / v)$ to denote a substitution of a variable v by x. As can be noticed from the additional notations given below, the state function can be replaced by formulas. ***St*** denotes the set of states, and A denotes the actions. F and G denote formulas, p is either a state function or a predicate, and $\mathbb{N}$ denotes the set of natural numbers.

Semantics

$s[\![f]\!]$	=	$f\left(\forall v \mathbf{:} s[\![v]\!] / v\right)$
$s[\![A]\!]t$	=	$A\left(\forall v \mathbf{:} s[\![v]\!] / v, t[\![v]\!] / v'\right)$
$\Vert\!= A$	=	$\forall s,t \in \mathbf{St:} s[\![A]\!]t$
$\sigma[\![F \wedge G]\!]$	=	$\sigma[\![F]\!] \wedge \sigma[\![G]\!]$
$\sigma[\![\neg F]\!]$	=	$\neg \sigma[\![F]\!]$
$\Vert\!= F$	=	$\forall \sigma \in S^{\infty} \mathbf{:} \sigma[\![F]\!]$
$s[\![Enabled\ A]\!]$	=	$\exists t \in \mathbf{St:} s[\![A]\!]t$
$\langle s_0, s_1, \ldots \rangle [\![A]\!]$	=	$s_0 [\![A]\!] s_1$
$\langle s_0, s_1, \ldots \rangle [\![\Box F]\!]$	=	$\forall n \in \mathbb{N} : \langle s_n, s_{n+1}, \ldots \rangle [\![F]\!]$

Additional notation

p'	=	$p(\forall v \mathbf{:} v' / v)$
$[A]_f$	=	$A \vee (f' = f)$
$\langle A \rangle_f$	=	$A \wedge (f' = f)$
$Unchanged\ f$	=	$f' = f$
$\Diamond F$	=	$\neg \Box \neg F$
$F \mapsto G$	=	$\Box(F \Rightarrow \Diamond G)$
$WF_f(A)$	=	$\Box Enabled \langle A \rangle_f \mapsto \langle A \rangle_f$
$SF_f(A)$	=	$\Box \Diamond Enabled \langle A \rangle_f \mapsto \langle A \rangle_f$

For a state function f, the mapping from the set ***St*** of states to the set ***Val*** of values is denoted by $[\![f]\!]\mathbf{:}\boldsymbol{St} \rightarrow \boldsymbol{Val}$. If $s[\![f]\!]$ denotes the value that $[\![f]\!]$ assigns to a state s, then the values obtained from f by substituting $s[\![v]\!]$ for all variables v are obtained by $f\left(\forall v\mathbf{:}s[\![v]\!]/v\right)$. For an action A, the mapping from a pair of states to boolean values is denoted by $[\![A]\!]$. If s is the old state and t is the new state, then an action $s[\![A]\!]t$ is obtained from A by replacing each unprimed variable v by $s[\![v]\!]$, and each primed variable v' by $t[\![v]\!]$. An action A is valid (written $||= A$) if and only if every step is an A step. By $\sigma[\![F]\!]$ we denote the boolean value that formula F assigns to a behaviour σ. Since a formula is built up from elementary formulas, it is easy to write $\sigma[\![F]\!]$ in terms of elementary formulas obtained from F. A formula F is valid (written $||= F$) if and only if it is satisfied by all possible behaviours.

$Enabled\ A$ is a predicate that is true for a state if and only if it is possible to take a step A starting in that state. The meaning of $\langle s_0, s_1, \ldots\rangle[\![A]\!]$ is that $[\![A]\!]$ is true for a behaviour if and only if the first pair of states in the behaviour is an A step; $\langle s_0, s_1, \ldots\rangle[\![\Box F]\!]$ denotes the fact that a behaviour satisfies $\Box F$ if and only if every step of the behaviour is an F step. The formula $\Diamond F$ asserts that F holds now or eventually in the future.

The operators $\Box$ and $\Diamond$ can be nested and combined with logical operators to provide more complicated temporal modalities. For example, $\Box\Diamond F$ asserts that at all times, F must be true then or at some future time. In other words, $\Box\Diamond F$ asserts that F is true infinitely often. $F \mapsto G$ states that whenever F is true, G will eventually become true.

$WF_f(A)$ (weak fairness of A) asserts that if A becomes forever enabled, then a non-stuttering A transition must eventually occur. $SF_f(A)$ (strong fairness of A) asserts that whenever A is enabled infinitely often, a non-stuttering A transition must eventually occur. Since $\Box F$ implies $\Box\Diamond F$ for any F, we see that $SF_f(A)$ implies $WF_f(A)$, for any action A.

A program is a temporal logic formula Π defined by

$$\Pi = Init_\Pi \wedge \Box[N_\Pi] \wedge L_\Pi ,$$

where:

- a state predicate $Init_\Pi$ is specifying the initial state;
- an action N_Π is specifying the state transitions allowed by the program Π; N_Π is the disjunction of the actions representing the program's atomic operations;
- a temporal formula L_Π specifies the program's progress condition; the program action N_Π specifies what the program must do, while the progress condition L_Π describes what the program eventually may do.

Viewed as an assertion about a behaviour σ, the first conjunct in Π states that $Init_\Pi$ should hold in the initial state σ_0, the second conjunct states that every step of σ is a stuttering step or a transition allowed by N_Π, and the third one states that σ satisfies the progress condition.

A progress condition is usually expressed in terms of fairness conditions for actions. The weakest progress condition which occurs in practice asserts that the program never halts if some step is possible; this condition is expressed by the formula $WF(N_\Pi)$.

3. MOBILE AMBIENTS

Since we intend to express mobile ambients in the temporal logic of actions, we need to consider a subset of mobile ambients, where there is no communication, restriction or replication. We follow [2] to provide a brief description of mobile ambients (called also MA-processes). Given an infinite set of names

Names (ranged over by *m,n, ...*), we define the set **A** of mobile ambients (denoted by ***A,A′,B,...***) together with their capabilities (denoted by *C,C′,...*) as follows:

$$C ::= in\ n \mid out\ n \mid open\ n$$
$$\boldsymbol{A} ::= \boldsymbol{0} \mid \boldsymbol{A}|\boldsymbol{B} \mid C.\boldsymbol{A} \mid n[\boldsymbol{A}]$$

Process ***0*** is an inactive mobile ambient. A movement *C.**A*** is provided by the capability *C*, followed by the execution of ***A***. An ambient *n[**A**]* represents a bounded place labelled by *n* in which a MA-process ***A*** is executed. ***A*** | ***B*** is a parallel composition of mobile ambients ***A*** and ***B***. The *structural congruence* $\equiv_a$ over ambients is the least congruence such that (**A**, |, ***0***) is a commutative monoid.

The operational semantics of mobile ambients is defined in terms of a reduction relation $\to_a$ by the following axioms and rules:

Axioms:

(In) $n[in\ m.\ \boldsymbol{A} \mid \boldsymbol{A'}] \mid m[\boldsymbol{B}] \to_a m[n[\boldsymbol{A} \mid \boldsymbol{A'}] \mid \boldsymbol{B}]$

(Out) $m[n[out\ m.\ \boldsymbol{A} \mid \boldsymbol{A'}] \mid \boldsymbol{B}] \to_a n[\boldsymbol{A} \mid \boldsymbol{A'}] \mid m[\boldsymbol{B}]$

(Open) $open\ n.\ \boldsymbol{A} \mid n[\ \boldsymbol{B}\ ] \to_a \boldsymbol{A} \mid \boldsymbol{B}$

Rules:

(Comp) $\dfrac{\boldsymbol{A} \to_a \boldsymbol{A'}}{\boldsymbol{A} \mid \boldsymbol{B} \to_a \boldsymbol{A'} \mid \boldsymbol{B}}$

(Amb) $\dfrac{\boldsymbol{A} \to_a \boldsymbol{A'}}{n[\boldsymbol{A}] \to_a n[\boldsymbol{A'}]}$

(Struct) $\dfrac{\boldsymbol{A} \equiv_a \boldsymbol{A'}, \boldsymbol{A} \to_a \boldsymbol{B'}, \boldsymbol{B} \equiv_a \boldsymbol{B'}}{\boldsymbol{A} \to_a \boldsymbol{B}}$

By $\to_a^*$ we denote the reflexive and transitive closure of the binary relation $\to_a$.

4. DESCRIBING MOBILE AMBIENTS BY USING TLA

A specific feature of ambient calculus is that an MA-process has a spatial tree-like structure, and so, the mobile ambients are acted upon by the tree structure.

In order to write specifications for mobile ambients in TLA, we first have to find a way in which the ambients can be treated as programs represented in the memory of a computer. We denote such a representation by *mem*, and it consists of a matrix with five columns and a number of lines equal with the number of ambients and capabilities present in the translated MA-process. Inspired from [5], we represent ambients and capabilities in the memory of a computer as tuples *(name,type, target, parent_label,label)* where:

- *name* - it is either the name of the ambient, or a capability *{in, out,open}*;
- *type* - it indicates that we have either an ambient (“amb”), or a capability (“cap”);
- *target* - when *name* is a capability, *target* indicates the ambient it intends to move;
- *parent_label* - the label of the parent (we use natural numbers for labels);
- *label* - a unique label given to *name* (which is ambient or capability).

The idea is that the tree representation of an ambient coincides with the representation derived from a memory representation. In this way, an ambient n is represented as *(n, “amb”, n, parent_label, label)*, where *target=n* because the ambient does not act upon other ambients as capabilities do. A capability *cap n* with *cap* ∈ *{in, out,open}* is represented as *(cap, ”cap”', n, parent_label, label)*.

In order to justify the use of unique labels to identify ambients and capability, as well as their parents, suppose we have the mobile ambient *k[open n | n[n[m[ ]]]]* that evolves to *k[n[m[ ]]]* by using the *(Open)* axiom of mobile ambients.

Given a memory representation of these ambients, we should be able to simulate (execute) the *(Open)* axiom. This means that all the ambients and capabilities which had as parent the dissolved ambient *n*, should have their parent changed to the ambient *k* which is the parent of the dissolved ambient *n*. By adding unique labels to each ambient and capability, we are able to identify which of the two *n* ambients is dissolved, and thus the steps simulating the *(Open)* axiom are:

- search all ambient and capability representations which have *parent_label*= l_1 , where l_1 is the unique label attached to the dissolved ambient *n*;
- replace l_1 by l_2 in all these representations, where l_2 is the unique label associated to ambient *k*, the parent ambient of *n*.

Using this representation, we can automatically translate a mobile ambient into a memory representation by using a function $T: \mathbf{A} \times \mathbb{N} \times \mathbb{N} \rightarrow Memory$ defined by

$$T(\boldsymbol{A}, l_1, l_2) = \begin{cases} (cap, "cap", n, l_1, l_2) + T(\boldsymbol{B}, l_2, l_2 + 1) & \text{if } \boldsymbol{A} = cap\ n.\boldsymbol{B} \\ (n, "amb", n, l_1, l_2) + T(\boldsymbol{B}, l_2, l_2 + 1) & \text{if } \boldsymbol{A} = n[\boldsymbol{B}] \\ T(\boldsymbol{B}, l_1, l_2) + T(\boldsymbol{C}, l_1, l_2 + count(\boldsymbol{B})) & \text{if } \boldsymbol{A} = \boldsymbol{B} \mid \boldsymbol{C} \\ \varnothing & \text{if } A = \boldsymbol{0} \end{cases}$$

In this translation:

- ***A*** is a mobile ambient;
- l_1 is the label of the parent of ***A***;
- l_2 is the (unique) label which is used as a reference (starting position) for the next steps;
- + adds a new entry in the matrix *mem*;
- $\varnothing$ represents an empty entry that can be ignored from the matrix *mem*.

We use an additional function $count : \mathbf{A} \rightarrow \mathbb{N}$ which provides the number of ambients and capabilities in a mobile ambient:

$$count(\boldsymbol{A}) = \begin{cases} 1 + count(\boldsymbol{B}) & \text{if } \boldsymbol{A} = cap\ n.\boldsymbol{B} \text{ or } \boldsymbol{A} = n[\boldsymbol{B}] \\ count(\boldsymbol{B}) + count(\boldsymbol{C}) & \text{if } \boldsymbol{A} = \boldsymbol{B} \mid \boldsymbol{C} \\ \boldsymbol{0} & \text{if } \boldsymbol{A} = \boldsymbol{0} \end{cases}$$

Example 1. We consider a simple example to illustrate how this translation works. Suppose we have a mobile ambient *m[ ] | n[in m]*. Then, by applying the translation function T for this mobile ambient having a parent *k* with label *1*, we have:

mem= T *(m[ ] | n[in m], 1, 2)* = T *(m[ ], 1, 2)* + T *(n[in m], 1, 3)*
= *(m,"amb", m, 1, 2)+(n,"amb", n, 1, 3)* + T *(in m, 3, 4)*
= *(m,"amb", m, 1, 2) + (n,"amb", n, 1, 3) +(in,"cap", m, 3, 4)*

We can simulate the evolution of a mobile ambient by using this representation, where *Loc* is the set of all possible memory locations. In order to non-deterministically select a capability of an ambient, we define

$$ProcAction\ \ GetCap(x) = \left(\oplus_{l \in Loc} \left((\boldsymbol{cap} = mem[l][\boldsymbol{2}]) \wedge (x' = mem[l]) \right) \uparrow \right)^*$$

The right hand side of a **ProcAction** statement is an expression formed by combining TLA actions with the following additional operators defined in [8] ; $\otimes$ $\oplus$ $(\ldots)$ $\uparrow$, operators that are used to formally add process algebra to TLA. These operators have the following intuitive interpretation:

- *A;B* - do *A* then *B*;
- *A* $\otimes$ *B* - do *A* or *B*;
- $\oplus_{v \in S}$ *A(v)* - do *A(v)* for some $v \in S$;
- $(A)^*$ -- keep doing *A* actions forever, or until the loop is exited;
- $A \uparrow$ -- do *A*, then exit from the innermost part containing $(\ldots)^*$.

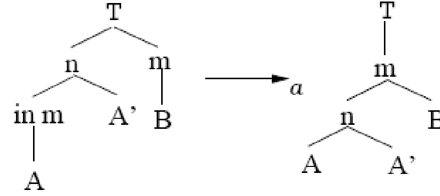
The procedure above describes the following process: we search the memory until we find a location *l* where the element *l[2]* is equal to *cap*, and we save this entry in *x*.

In a similar manner we can non-deterministically choose two ambients, one at a time:

$$ProcAction\ \ GetAmb(y) = \left(\oplus_{l \in Loc} \left((\boldsymbol{amb} = mem[l][\boldsymbol{2}]) \wedge (y' = mem[l]) \right) \uparrow \right)^*$$

In order to describe the consumption of capabilities of a mobile ambient, we study three cases, one for each type of capability. Let us suppose that after using *GetCap(x)* we have a capability in the entry *x*, and similarly, using $GetAmb(y_1)$ and $GetAmb(y_2)$ we have two ambients in the entries y_1 and y_2.

In what follows we present the reduction when a capability *in* is consumed. This means that we start with a mobile ambient *n[in m.**A** | **A'**] | m[**B**]* which is reduced in one step to the mobile ambient *m[n[**A** |**A'**] | **B**]*. In a tree representation, this reduction can be viewed as:



The conditions which should be satisfied in order to use a capability *in* are:

- [1]="*in*",
- x[3]= y_1 [3] (the capability is *in* y_1 [3]),
- y_1 [4]= y_2 [4] (two ambients having the same parent),
- x[4]= y_2 [5] (the capability *in* y_1 [3] is placed inside the ambient with the label y_2 [5]).

The consumption of a capability *in* can be expressed in TLA as:

$$ProcAction \quad In(x, y_1, y_2) = (GetCap(x) \wedge GetAmb(y_1) \wedge GetAmb(y_2)$$
$$\wedge (x[1] = in) \wedge (x[3] = y_1[3]) \wedge (y_1[4] = y_2[4]) \wedge (x[4] = y_2[5]) \uparrow)^* \wedge (y'_2[4] = y_1[5])$$
$$\wedge \oplus_{l \in Loc}((l[4] = x[5]) \wedge (mem' = [mem \quad EXCEPT \quad ![l][4] = x[4], \quad ![x] = '' \, '']))$$

The effect of this procedure is given by the following steps:

- we nondeterministically select a capability through *GetCap(x)*, and two ambients through $GetAmb(y_1)$ and $GetAmb(y_2)$;
- we check if the conditions to consume a capability *in* are satisfied; namely $(x[1] = in) \wedge (x[3] = y_1[3]) \wedge (y_1[4] = y_2[4]) \wedge (x[4] = y_2[5])$;
- if step 2 is satisfied, then we go to step 4; if not, then we go to step 1 and search for a new triple (a capability and two ambients);
- we change the parent of the ambient y_2 with y_1 as an effect of consuming a capability *in*, namely $(y'_2[4] = y_1[5])$;
- we read all the memory entries ($\oplus_{l \in Loc}$) searching for all the capabilities or ambients which have as parent the consumed capability *(l[4]=x[5])*. If we find such entries, then we change their parent to the parent of the consumed capability (*l[4]=x[4]*). Also, in order to simulate the consumption of the capability, we remove the corresponding data from the memory (*x=" "*). The rest of the memory remains the same ($mem' = [mem \quad EXCEPT \quad ![l][4] = x[4], \quad ![x] = '' \, '']$); the TLA notation *[z EXCEPT ![i]=u]* means that the array $\hat{z}$ has the same values as *z* excepting index *i* where we have $\hat{z}[i] = u$.

Based on such a description in TLA, we can prove the following result:

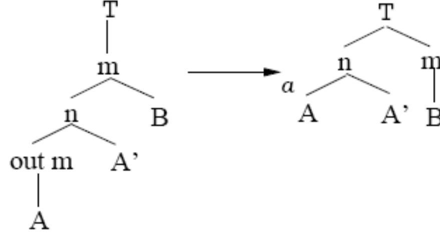
THEOREM 1. *If* $mem = T(\boldsymbol{A}, 1, 2)$ and it evolves to *mem'* *by executing an* $In(x, y_1, y_2)$, *then there exists an ambient* ***B*** *such that* $\boldsymbol{A} \rightarrow_a \boldsymbol{B}$, *with* $mem' = \rho(T(\boldsymbol{B}, 1, 2))$.

Proof [Sketch] . ρ is a label renaming function that is necessary since ***A*** and ***B*** have different structure, and it is possible that by applying the T function, the same ambient or capability from the ***A*** and ***B*** has different labels. The proof is by induction on the structure of ***A***. In what follows, we present a simple

case, the others being treated in a similar manner. Consider the ambient ***A***=*m[ ] | n[in m]* from *Example 1* that can evolve to ***B***=*m[n[ ]]*. In this case *mem= (m, "amb", m, 1, 2) + (n, "amb", n, 1, 3) + (in, "cap", m, 3, 4)*. By executing an $In(x, y_{1,} y_2)$, we obtain the memory: *mem'= (m, "amb", m, 1, 2) + (n, "amb", n, 2, 3)*. It can be noticed that $mem' = T(\boldsymbol{B}, 1, 2)$, so in this case no renaming is necessary and the ρ function is in fact the identity function.

We illustrate in Section 5 this result with an example implemented and executed in TLA+.

The consumption of the capabilities *out* and *open* can be defined in a similar manner. For instance, when we have a capability *out*, a mobile ambient *m[n[out m.**A** | **A'**] | **B**]* reduces to the mobile ambient *n[**A** | **A'**] | m[**B**]*. In a tree representation, this reduction can be viewed as:



The conditions which should be satisfied in order to use a capability *out* are:

- x[1]="*out*",
- x[3]= y_1 [3] (the capability is *out* y_1 [3]),
- y_2 [4]= y_1 [5] (the parent of the ambient y_2 is the ambient with the label y_1 [5]),
- x[4]= y_2 [5] (the capability out y_1 [3] is placed inside ambient y_2).

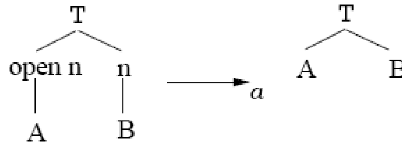
The consumption of a capability *out* can be expressed in TLA as:

$$ProcAction \quad Out(x, y_{1,} y_2) = (GetCap(x) \wedge GetAmb(y_1) \wedge GetAmb(y_2)$$
$$(x[\boldsymbol{1}] = out) \wedge (x[\boldsymbol{3}] = y_1[\boldsymbol{3}]) \wedge (y_2[\boldsymbol{4}] = y_1[\boldsymbol{5}]) \wedge (x[\boldsymbol{4}] = y_2[\boldsymbol{5}]) \uparrow)^* \wedge (y'_2[\boldsymbol{4}] = y_1[\boldsymbol{4}])$$
$$\wedge \oplus_{l \in Loc} ((l[\boldsymbol{4}] = x[\boldsymbol{5}]) \wedge (mem' = [mem \quad EXCEPT \quad ![l][\boldsymbol{4}] = x[\boldsymbol{4}], \quad ![x] = ''\,'']))$$

Based on this description, we get the following result:

THEOREM 2. *If* $mem = T(\boldsymbol{A}, 1, 2)$ *and it evolves to mem' by executing an* $Out(x, y_{1,} y_2)$, *then there exists an ambient **B** such that* $\boldsymbol{A} \rightarrow_a \boldsymbol{B}$, *with* $mem' = \rho(T(\boldsymbol{B}, 1, 2))$.

When a capability *open* is consumed, we have a mobile ambient *open n.**A** | n[**B**]* which reduces in one step to the mobile ambient ***A** | **B***. In a tree representation, this reduction can be viewed as:



The conditions which should be satisfied in order to use a capability *open* are:

- *x*[1]="*open*"',
- *x*[3]= y_1 [3] (the capability is *open* y_1 [3]), y_1
- [4]=*x*[4] (y_1 [3] and *open* y_1 [3] have the same parent),
- we ignore the second chosen ambient.

The consumption of a capability *open* can be expressed in TLA as:

$$ProcAction \quad Open(x, y_{1,} y_2) = (GetCap(x) \wedge GetAmb(y_1) \wedge GetAmb(y_2)$$
$$(x[\mathbf{1}] = open) \wedge (x[\mathbf{3}] = y_1[\mathbf{3}]) \wedge (y_1[\mathbf{4}] = x[\mathbf{4}]) \uparrow)^*$$
$$\wedge \oplus_{l \in Loc} ((l[\mathbf{4}] = x[\mathbf{5}]) \wedge (mem' = [mem \quad EXCEPT \quad ![l][\mathbf{4}] = x[\mathbf{4}], \quad ![x] = ''\,''\,]))$$
$$\wedge \oplus_{l \in Loc} ((l[\mathbf{4}] = y_1[\mathbf{5}]) \wedge (mem' = [mem \quad EXCEPT \quad ![l][\mathbf{4}] = y_1[\mathbf{4}], \quad ![x] = ''\,''\,]))$$

Based on this description, we get the following result:

THEOREM 3. *If* $mem = T(\boldsymbol{A}, 1, 2)$ *and it evolves to mem' by executing an* $Open(x, y_{1,} y_2)$*, then there exists an ambient* ***B*** *such that* $\boldsymbol{A} \rightarrow_a \boldsymbol{B}$*, with* $mem' = \rho(T(\boldsymbol{B}, 1, 2))$.

After defining the operations which simulate the consumption of capabilities $In(x, y_{1,} y_2)$, $Out(x, y_{1,} y_2)$ and $Open(x, y_{1,} y_2)$ in mobile ambients, we could define the specification for any MA-process $\boldsymbol{A}$ as follows: $Init = T(\boldsymbol{A}, 1, 2)$, $N = In(x, y_{1,} y_2) \vee Out(x, y_{1,} y_2) \vee Open(x, y_{1,} y_2)$ and $L = WF(N)$ where:

- *Init* represents the initial configuration of the memory describing process ***A***;
- *N* specifies the possible operations corresponding to capabilities *in*, *out* and *open*;
- *L* represents a temporal formula describing the weak fairness condition that asserts that the program never halts if some step is possible.

$\Pi = Init \wedge \Box[N] \wedge L$ represents a uniform description of mobile ambients in TLA. Starting from a mobile ambient ***A*** which is translated into *mem*, by executing Π we reach a memory configuration that corresponds to a mobile ambient ***B*** such that $\boldsymbol{A} \rightarrow_a \boldsymbol{B}$.

COROLLARY 1. *If* $mem = T(\boldsymbol{A}, 1, 2)$ *and it evolves to mem' by executing an* Π*, then there exists an ambient* ***B*** *such that* $\boldsymbol{A} \rightarrow_a \boldsymbol{B}$*, with* $mem' = \rho(T(\boldsymbol{B}, 1, 2))$.

This result proves that we can simulate the evolution of a MA-process ***A*** by executing the TLA description of ***A*** with the procedures *In*, *Out* and *Open*.

5. IMPLEMENTING MOBILE AMBIENTS IN TLA+

TLA+ [9], a formal language based on TLA, is able to specify the behaviour of very complex systems. In order to analyse and verify several properties of complex systems, we use TLC, a tool for automatic verification of finite-state TLA+ specifications by model checking. In order to present our implementation of mobile ambients in TLA+, let us consider the following ambient structure *m[s[ ]] | n[in m | in s]*, a structure which reduces in two steps to *m[s[n[ ]]]* by consuming the capabilities *in m* and *in s*. This means that some changes occur in memory. The steps and the fact that the modifications take place can be expressed by the next MODULE written in TLA+, where the extended modules are standard in TLA+:

MODULE SimpleEx

EXTENDS Naturals, TLC, Sequences

The mobile ambients are kept in the variable *memory*.

VARIABLES memory

Initially the variable memory contains the mobile ambient *m[s[ ]] | n[in m | in s]*. The translation is performed manually, since an implementation of the T function represents further work.

Init == memory \in {⟨⟨⟨⟨ "m", "amb", "m", "1", "2" ⟩⟩ ,

⟨⟨ "s", "amb", "s", "2", "3"⟩⟩ , ⟨⟨ "n", "amb", "n", "1","4"⟩⟩ ,

⟨⟨ "in", "cap" ,"s", "4","6"⟩⟩ , ⟨⟨ "in", "cap" ,"m", "4","5"⟩⟩⟩⟩ }

In order to execute $In(x, y_{1,} y_2)$, the fact that a capability *in* can be consumed should be verified as described before Theorem 1. This means

$InTest(x, y_{1,} y_2)$ **== (memory[x1][2]="cap")**
/\ (memory[y1][2] = ''amb'') /\ (memory[y2][2] = "amb")
/\(memory[x1][3]=memory[y1][3])
/\(memory[y1][4]=memory[y2][4])
/\(memory[x1][4]=memory[y2][5])

If the above preconditions are satisfied for a capability and some ambients, then a step *in* is simulated, and the memory is changed.

InNext == /\ \E x1 \in 1..5, y1\in 1..5, y2 \in 1..5: InTest(x1,y1,y2)
/\ memory' = [memory EXCEPT
![y2]=[@ EXCEPT ![4]= memory[y1][5]],
![x1]=" "," "," "," "," "]
In == Init /\ [][InNext] memory /\ WF memory(InNext)

Using this program, we can check if sometime in the future the ambient *n* is inside ambient *s*:

Test == (memory[3][4] = memory[2][5])
THEOREM In => Test

The execution generates three states (see Figure 1) corresponding to the three mobile ambients reached during execution, namely *m[s[ ]]|n[in m|in s]*, *m[s[ ]|n[in s]]* and *m[s[n[ ]]]*. The generated error "deadlock reached", means that the execution produced all reachable states and that the system cannot evolve any more, and checks if ambient n ever enters ambient s by checking if the parent ambient of n is ever going to be s (i.e., *memory[3][4] = memory[2][3]*).

```
TLC Version 2.0 of August 23, 2007
Model-checking
Parsing file SimpleEx.tla
Parsing file C:\tla\tlasany\StandardModules\Naturals.tla
Parsing file C:\tla\tlasany\StandardModules\TLC.tla
Parsing file C:\tla\tlasany\StandardModules\Sequences.tla
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module TLC
Semantic processing of module SimpleEx
Finished computing initial states: 1 distinct state generated.
Error: deadlock reached. The behavior up to this point is:
STATE 1: <Initial predicate>
memory = << <<"m", "amb", "m", "1", "2">>,
   <<"s", "amb", "s", "2", "3">>,
   <<"n", "amb", "n", "1", "4">>,
   <<"in", "cap", "m", "4", "5">>,
   <<"in", "cap", "s", "4", "6">> >>
STATE 2: <Action line 22, col 1 to line 25, col 88 of module SimpleEx>
memory = << <<"m", "amb", "m", "1", "2">>,
   <<"s", "amb", "s", "2", "3">>,
   <<"n", "amb", "n", "2", "4">>,
   <<"", "", "", "", "">>,
   <<"in", "cap", "s", "4", "6">> >>
STATE 3: <Action line 22, col 1 to line 25, col 88 of module SimpleEx>
memory = << <<"m", "amb", "m", "1", "2">>,
   <<"s", "amb", "s", "2", "3">>,
   <<"n", "amb", "n", "3", "4">>,
   <<"", "", "", "", "">>,
   <<"", "", "", "", "">> >>
3 states generated, 3 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 3.
```

Fig. 1 – The output of a TLA+ program.

6. CONCLUSION AND RELATED WORK

In this paper we use the temporal logic of actions to write formal specifications of mobile ambients. Then we execute these formal descriptions by using TLA+, and prove that such an execution corresponds operationally to the formal reduction steps in ambient calculus. Being interested in providing verification tools based on existing tools, we show that TLA can be used to specify mobile ambients, and TLA+ to

simulate and analyse them. In this way, we can consider temporal logic of actions as a specification and verification framework for mobile ambients.

A related approach is presented in [4], where ambient logic is used for specification and verification of mobile ambients. As a drawback of that approach, the dynamics of the systems cannot be described in detail in the logic, and a software executing the formal description does not exist.

We can also mention [11] which introduced a spatial-temporal logic called MTLA whose temporal part is based on TLA. In addition to the temporal operators, some spatial modalities can be used to describe the structure of the system and its modifications. In this paper we did not take this approach since the TLA+ software was not extended to incorporate this spatial extension of TLA. That is why, inspired by [5], we used a memory representation of mobile ambients such that no spatial modalities are needed for the simulation of mobile ambients evolution.

A different approach of using TLA over mobile ambients is given in [6], where a process-algebraic approach for specifying and checking connectivity was proposed. A process is connected to another one if a message from the first one reaches the other one by using communication and movement actions.

ACKNOWLEDGEMENTS

The work was supported by a grant of the Romanian National Authority for Scientific Research, project number PN-II-ID-PCE-2011-3-0919.

REFERENCES

1. B. ALPERN, F.B. SCHNEIDER, *Recognizing Safety and Liveness*. Distributed Computing, **2**, pp. 117–126, 1987.
2. L. CARDELLI, A. GORDON, *Mobile Ambients*. Lecture Notes in Computer Science, **1378**, pp. 140–155, Springer, 1998.
3. L. CARDELLI, A. GORDON, A*nytime, Anywhere: Modal Logics for Mobile* Ambients, Proceedings POPL, 2000, pp. 365–377.
4. W. CHARATONIK, S. DAL ZILIO, A.D. GORDON, S. MUKHOPADHYAY, J-M. TALBOT, *Model Checking Mobile Ambients*, Theoretical Computer Science, **308**, pp. 277–331, 2003.
5. G. CIOBANU, V. ZAKHAROV, *Encoding Mobile Ambients into pi-calculus*, Lecture Notes in Computer Science, **4378**, pp. 148–161, Springer, 2006.
6. T. KAPUS, *Checking Connectivity in Mobile System Ambients with the Temporal Logic of Actions*, IEICE Transactions, **E89**, **A**, *11*, pp. 3333–3340, 2006.
7. L. LAMPORT, *The Temporal Logic of Actions*, ACM Transactions on Programming Languages and Systems, **16**, *3*, pp. 872–923, 1994.
8. L. LAMPORT, *Adding Process Algebra to TLA*, January 1995.
9. L. LAMPORT, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley Professional, 2002.
10. R. MILNER, *Communicating and Mobile Systems: the pi-calculus*, Cambridge University Press, 1999.
11. J. ZAPPE, *Towards a Mobile Temporal Logic of Actions*, Ph.D. Thesis, Institut fur Informatik, Ludwig Maximilians Universitat Munchen, 2005.

Received July 12, 2013