




Verification of Multi-agent Systems with Timeouts for Migration and Communication

Bogdan Aman^{1,2}  and Gabriel Ciobanu^{1,2} 

¹ Faculty of Computer Science,

Alexandru Ioan Cuza University, Iasi, Romania

² Institute of Computer Science, Romanian Academy, Iasi, Romania

{bogdan.aman,gabriel}@info.uaic.ro

Abstract. A prototyping high-level language is used to describe multi-agent systems using timeouts for migration between explicit locations and local communication in a distributed system. We translate such a high-level specification into the real-time Maude rewriting language. We prove that this translation is correct, and provide an operational correspondence between the evolutions of the mobile agents with timeouts and their rewriting translations. These results allow to analyze the multi-agent systems with timeouts for migration and communication by using the real-time Maude tools. A running example is used to illustrate the whole approach.

1 Introduction

Multi-agent systems are composed of a large number of agents that behave according to their timed actions. The mobility of agents and the communication between agents may lead to unexpected behaviours. Components can be highly heterogeneous, having individual objectives and using different temporal scales to achieve them. As multi-agent systems are getting more complex, automated verification of such systems is needed. Actually, the specification and analysis of multi-agent systems represent an active research direction in the last years. It is important to have modelling techniques able to describe easily such systems, as well as tools to simulate and verify some complex (qualitative and quantitative) properties of their behaviours. We take a step in this direction by developing a high-level specification language for specifying the mobile agents with timeouts, and providing a way to perform automated verification of some complex systems involving explicit locations and timeouts for migration and local communication in distributed networks.

There exist already some approaches to formalize timed systems, for instance timed automata [1]. Software platforms as UPPAAL [3] represent model checking tools used for the simulation and verification of real-time systems modelled as timed automata [10]. Logic-based models complement the timed automata models as they are able to capture other aspects of real-time systems: e.g., mobility of agents and the communication between agents. Rewriting logic is appropriate for

providing a general semantic framework for various languages and models of concurrency [11]. Maude is a system that supports computations based on rewriting and equational logic, while real-time Maude [14] provides a specification formalism with several decidability results for many system properties. Also, in real-time Maude different types of communication used in process calculi can be modelled. The real-time Maude tool is developed using an extension of rewriting logic, and seem to be an appropriate tool for specification, validation and verification of real-time systems using features as migration of agents and communication between agents. This tool is useful in applications that use features not yet implemented in several existing model checkers for real-time systems [13].

For the specification of the multi-agent systems with timeouts for migration and local communication we use a real-time version of an existing high-level framework called `TIMO`, a framework able to describe easily interacting mobile agents in distributed systems. Then we translate this high-level specification into real-time Maude. There are some problems to overcome in order to obtain a fully executable specification in real-time Maude. Firstly, the transitions of a system need sometimes to use fresh names (to overcome binding problems); this is due to the fact that the communication of values takes place eventually after alpha-converting (to avoid clashes) in the high-level specification. Secondly, since infinite computations are not supported by real-time Maude, implementing an unbounded recursion operator is not possible; a solution to this problem is to consider a bounded recursion in which a process can be unfolded only a finite number of times during an execution. This restriction does not influence the results because we model real systems in which the recursive processes need to be unfolded only for a finite number of times.

2 Syntax and Semantics of the High-Level Specification

In the high-level specification language, the processes are allowed to migrate between explicit distributed locations and to communicate locally with other processes. The coordination of the processes in time and space is done by using timed migration and timed communication. The timeouts added to a migration action enforce the process to migrate to the target location after a period of time equal to the timeout constraint. Two processes are allowed to communicate only if they are both available into the same location at the same unit of time, and if the timeout restrictions of the active communication actions are non-negative. If a communication action cannot be executed before its timeout restriction expires, then the action is removed and the actions of an alternative process are executed as a continuation. The transitions involving either processes migration between locations, processes communication or unfolding of recursive processes are executed in a maximal parallel manner. This means that if a process can migrate, communicate or unfold, it has to do it. The transitions with timeouts are alternated with transitions involving the passage of time over all the processes; a global clock is used to model the passage of time. The operational semantics of the high-level specification is provided by using these two types of transitions:

a transition relation for timed migration and communication actions executed in the maximal parallel manner, and a transition relation used to model the passage of time.

A timeout restriction assigned to a migration action is given as a natural number t , while a timeout restriction assigned to a communication action is given as Δt , where $t \in \mathbb{N}$. The t notation means that migration action can be consumed exactly after t units of time, while the Δt notation means that the communication can be consumed at any moment in the next t units of time.

The syntax of the high-level language is given in Table 1, where the following notations are used:

- we use the set Loc of locations, set $Chan$ of communication channels, and set Id of process identifiers;
- for each process identifier $id \in Id$ there exists a unique process definition $id(u_1, \dots, u_{m_{id}}) \stackrel{def}{=} P_{id}$ in which the m_{id} parameters are identified by the distinct variables u_i ;
- a, l, t denote a communication channel, a location or a location variable, and an action timeout, respectively; u and v denote a tuple of variables and a tuple of expressions built from values (e.g., strings, integers, bools), variables and allowed operations.

Table 1. The syntax of the high-level language

<i>Processes</i>	$P, Q ::= a^{\Delta t}!\langle v \rangle \text{ then } P \text{ else } Q \mid$	(output)
	$a^{\Delta t}?(u) \text{ then } P \text{ else } Q \mid$	(input)
	$go^t l \text{ then } P \mid$	(move)
	$0 \mid$	(termination)
	$id(v) \mid$	(recursion)
	$P \mid Q$	(parallel)
<i>Located Processes</i>	$L ::= l[[P]]$	
<i>Multi-Agent Systems</i>	$N ::= L \mid N \parallel N \mid 0$	

An output communication process $a^{\Delta t}!\langle z \rangle \text{ then } P \text{ else } Q$ describes the fact that for t time units the process is available for sending on channel a the value z . Whenever the process succeeds in sending the value before the deadline, it continues its evolution according to process P ; otherwise, it continues its evolution by the alternative process Q . The input communication process $a^{\Delta t}?(x) \text{ then } P \text{ else } Q$ describes the fact that for t time units the process is available to receive on channel a a value to instantiate the variable x . In a similar manner as for the output communication process, the continuation of an input communication process depends on the success of the communication.

A migration process $go^t l \text{ then } P$ indicates a location change after t time units, namely after t units of time the process continues its execution as P at location l (and not at the current location). Since variables are instantiated through communication, this means that the location variables can be instantiated; this feature allows a flexible behaviour as processes can adapt their migration based

on received information. The process 0 models an inactive process, while the process $P \mid Q$ models the parallel composition of the process P and Q that might also interact through communication. A process P currently located in location l is denoted by $l[[P]]$, while a system is composed of located processes composed by using the parallel operator.

There is only one binding operator in our calculus: in the input process $a^{\Delta t?}(u)$ then P else Q , the variable u is bound in process P . However, as process Q is an alternative process executed when the input action is not consumed, this means that variable u is not bound in process Q . Given a process P , we denote by $fv(P)$ its set of free variables. In case u_i are the m_{id} parameters of the process P_{id} , then the assumption $fv(P_{id}) \subseteq \{u_1, \dots, u_{m_{id}}\}$ holds. As usually assumed in process calculi, we consider that processes are defined up to an alpha-conversion. Also, $P\{v/u, \dots\}$ denotes a process P in which v replaces all the free occurrences of the variable u , possible after using alpha-conversion inside P to remove possible clashes. A system N is said to be well-formed if $fv(N) = \emptyset$.

Operational Semantics. The structural equivalence relation \equiv represents an ingredient of the operational semantics; it is defined as the smallest congruence relation satisfying the equations of Table 2. The purpose of this relation \equiv is to provide a way of rearranging the processes in a system such that they can evolve by using the operational semantics rules from Table 3.

Table 2. Structural congruence in high-level specification

(PNULL)	$P \mid 0 \equiv P$
(LNULL)	$N \parallel \mathbf{0} \equiv N$
(LCOMM)	$N \parallel N' \equiv N' \parallel N$
(LASSOC)	$(N \parallel N') \parallel N'' \equiv N \parallel (N' \parallel N'')$
(LSPLIT)	$l[[P \mid Q]] \equiv l[[P]] \parallel l[[Q]]$

The equalities of Table 2 are useful for transforming a system N into the system $l_1[[P_1]] \parallel \dots \parallel l_n[[P_n]]$ composed of located process $l[[P_i]]$ such that there do not exist Q_i and R_i such that $P_i \equiv Q_i \mid R_i$. A located process that cannot be split into parallel located processes by using the rule (LSPLIT) is called a component of N , while the component decomposition of a system N is the system $l_1[[P_1]] \parallel \dots \parallel l_n[[P_n]]$, where all $l_i[[P_i]]$ are components.

Table 3 presents the operational semantics rules. The transitions of the form $N \rightarrow N'$ indicate either processes migrating between locations, processes communicating locally or unfolding of processes, all these executed in parallel in one step. The passing of t time units is given by transitions of the form $N \xrightarrow{t} N'$.

In rule (COM), two process $a^{\Delta t!}\langle v \rangle$ then P else Q and $a^{\Delta t?}(u)$ then P' else Q' , both located at location l , are using channel a to communicate a tuple of values v to be used for the instantiation of the variable u . Applying the rule (COM) does not lead to a location change for any of the involved processes, but to a consumption of the output and input action. Upon a successful communication, the processes

Table 3. The operational semantics of the high-level language

(STOP)	$l[[0]] \not\rightarrow$	(DSTOP)	$l[[0]] \xrightarrow{t} l[[0]]$
(COM)	$l[[a^{\Delta t}! \langle v \rangle \text{ then } P \text{ else } Q]] \parallel l[[a^{\Delta t'}? \langle u \rangle \text{ then } P' \text{ else } Q']] \rightarrow l[[P]] \parallel l[[P'\{v/u\}]]$		$t \geq t' > 0$
(DPUT)	$l[[a^{\Delta t}! \langle v \rangle \text{ then } P \text{ else } Q]] \xrightarrow{t'} l[[a^{\Delta t-t'}! \langle v \rangle \text{ then } P \text{ else } Q]]$		
(PUT0)	$l[[a^{\Delta 0}! \langle v \rangle \text{ then } P \text{ else } Q]] \rightarrow l[[Q]]$		$t \geq t' > 0$
(DGET)	$l[[a^{\Delta t}?\langle u \rangle \text{ then } P \text{ else } Q]] \xrightarrow{t'} l[[a^{\Delta t-t'}?\langle u \rangle \text{ then } P \text{ else } Q]]$		
(GET0)	$l[[a^{\Delta 0}?\langle u \rangle \text{ then } P \text{ else } Q]] \rightarrow l[[Q]]$		$t \geq t'$
(DMOVE)	$l[[go^t l' \text{ then } P]] \xrightarrow{t'} l[[go^{t-t'} l' \text{ then } P]]$		
(MOVE0)	$l[[go^0 l' \text{ then } P]] \rightarrow l'[[P]]$		
(DCALL)	$l[[P_{id}\{v/x\}]] \xrightarrow{t} l[[P'_{id}]] \quad id(v) \stackrel{def}{=} P_{id}$		$l[[id(v)]] \xrightarrow{t} l[[P'_{id}]]$
(CALL)	$l[[P_{id}\{v/x\}]] \rightarrow l[[P'_{id}]] \quad id(v) \stackrel{def}{=} P_{id}$		$l[[id(v)]] \rightarrow l[[P'_{id}]]$
(DPAR)	$N_1 \xrightarrow{t} N'_1 \quad N_2 \xrightarrow{t} N'_2 \quad N_1 \parallel N_2 \not\rightarrow$		$N_1 \parallel N_2 \xrightarrow{t} N'_1 \parallel N'_2$
(PAR)	$N_1 \rightarrow N'_1 \quad N_2 \rightarrow N'_2$		$N_1 \parallel N_2 \rightarrow N'_1 \parallel N'_2$
(DEQUIV)	$N_1 \equiv N'_1 \quad N'_1 \xrightarrow{t} N'_2 \quad N'_2 \equiv N_2$		$N_1 \xrightarrow{t} N_2$
(EQUIV)	$N_1 \equiv N'_1 \quad N'_1 \rightarrow N'_2 \quad N'_2 \equiv N_2$		$N_1 \rightarrow N_2$

$a^{\Delta t}! \langle v \rangle \text{ then } P \text{ else } Q$ and $a^{\Delta t'}? \langle u \rangle \text{ then } P' \text{ else } Q'$ continue their executions as processes P and $P'\{v/u\}$, respectively. If the process $a^{\Delta 0}! \langle v \rangle \text{ then } P \text{ else } Q$ exists in the system, then the communication action is discarded by using the rule (PUT0), and the execution continues as the alternative process Q . In a similar manner, by using the rule (GET0), the process $a^{\Delta 0}?\langle u \rangle \text{ then } P' \text{ else } Q'$ continues its execution as the alternative process Q . In rule (MOVE0), a process $go^0 l' \text{ then } P$ is able to change its location by migrating from the current location l to the given location l' where it continues its execution as process P . The unfolding of recursive processes is performed by using the rule (CALL). In order to use the structural equivalence relation \equiv to rearrange a system such that its components can interact for communication or migration, the rule (EQUIV) is used. Composing larger systems from smaller systems is done by using the rule (PAR) for the parallel composition operator.

The passage of time is described by the rules having their names starting with the capital letter D . The hypothesis $N_1 \parallel N_2 \not\rightarrow$ from the rule (DPAR) indicates the fact that placing the two systems N_1 and N_2 in parallel does

not trigger the application of a rule (COM) that would modify these systems. The negative premises are essential to separate the steps based on the execution of actions by those based on time passing (i.e., time cannot pass when an action is executed).

A transition of the form $N \rightarrow N_1$ followed by a time passing transition of the form $N_1 \xrightarrow{t} N'$ describe a complete step that can be written as:

$$N \rightarrow N_1 \xrightarrow{t} N'.$$

Thus, a complete step indicates that a parallel execution of processes migrating between locations, processes communicating or unfolding is necessarily followed by a time step. We say that the system N' is directly reachable from N if a complete computational step $N \xrightarrow{\Delta} N_1 \xrightarrow{t} N'$ exists. If $N \not\rightarrow$, then only a time step $N \xrightarrow{t} N'$ can be performed in the system N .

Theorem 1. *For all the systems N , N_1 and N_2 ,*

$$\text{if } N \xrightarrow{t} N_1 \text{ and } N \xrightarrow{t} N_2, \text{ then } N_1 \equiv N_2.$$

Theorem 1 claims that nondeterminism cannot be introduced upon executing a time transition in a system, namely the obtained system is unique up to structural congruence.

Theorem 2. *For all the systems N , N_1 , N_2 and $0 < t' < t$, we have $N \xrightarrow{t} N_2$ if and only if there is a N_1 such that $N \xrightarrow{t'} N_1$ and $N_1 \xrightarrow{t-t'} N_2$.*

Theorem 2 claims that whenever a time transition of length t can be performed in a system N leading to a system N_2 , then always a time transition of length t' with $0 < t' < t$ can be performed in the same system N leading to a system N_1 followed by another time transition of length $t - t'$ in the systems N_1 leading to N_2 , and vice versa. This result ensures that the passage of time in a system is continuous (no jumps).

Example 1. Let us consider an example in which a client wants to buy, at a good price, a flight ticket to a given location. The scenario is depicted in Fig. 1, where the names and values have the meanings given below (we explain the names and values in the order they appear, from left to right).

- The process *client* initially resides at location *home*. It has access to 130 cash units to be used for purchasing a flight ticket. Once the *client* reaches the *travelshop* location, an *agent* communicates to it the location of a *standard* offer. The *client* process goes to this location to receive the *standard* offer details. Here it also receives the location for a *special* offer. After receiving the information about the *special* offer, it goes to the bank for paying the cheaper offer between the *standard* and the *special* offers, and returns *home* (its initial location).
- The process *update* is able to migrate to the *special* location by starting from its initial location *travelshop* in order to communicate locally a reduction for the price *special* from 90 to 60 cash units.

- The process *agent* resides at the *travelshop* location, and has access to 100 cash units available in the cash register. Once a *client* reaches the *travelshop* location and the *agent* is available for communication, the client receives the location where the details of the standard offer are available. The *agent* has also the possibility to go to the *bank* to withdraw the available money from the *till*. Regardless of the amount of money taken from the *bank*, the *agent* always returns to *travelshop*, its initial location.
- The process *flightinfo* process residing at the *standard* location is able to do only local communications in order to provide to any interested client the details about the *standard* offer: the price of 110 cash units, and the location where the *special* offer resides.
- The process *saleinfo* process residing at the *special* location is able to do only local communications in order to provide (to any interested client) the details about the *standard* offer: the price of 90 cash units, and the location of the *bank* for the payment. The *saleinfo* process can also interact locally with the *update* process in order to modify the price of the *special* offer.
- The process *till* process owning 10 cash units and residing at the *bank* location is able to do only local communications: it can interact with a *client* to receive the payment for a flight ticket, and can interact with the *agent* in order to transfer the accumulated cash to it.

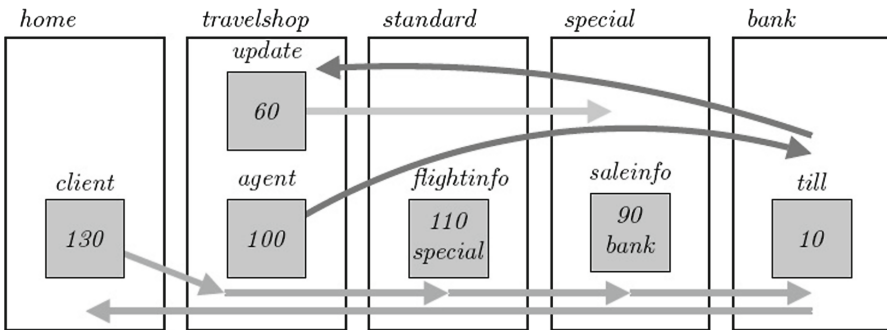


Fig. 1. Initial scenario

After all the interactions described in Fig. 1, the system looks like in Fig. 2.

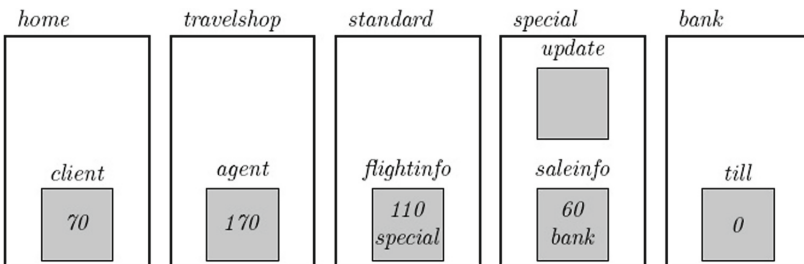


Fig. 2. A possible outcome

In the above example we have: (i) agents migrating in a distributed network with explicit locations; (ii) local communication of these agents (to get specific results); (iii) both migration and communication require certain time indicated by timeouts.

We show how this example can be easily described in our high-level language. First of all, in order to simplify the syntax, we consider that:

$a^{\Delta\infty}!\langle v \rangle$ then P else Q can be written as $a!\langle v \rangle \rightarrow P$,
 $a^{\Delta\infty}?(u)$ then P else Q can be written as $a?(u) \rightarrow P$, and
 $\text{go}^t l$ then P can be written as $\text{go}^t l \rightarrow P$.

This is because branch Q is ignored as it can never be executed.

The system presented in Fig. 1 is described in the high-level language as:

$$\begin{aligned} \text{TravelShop} = & \text{home} [[\text{client} (130)]] \parallel \text{travelshop} [[\text{update} (60) \mid \text{agent} (100)]] \\ & \parallel \text{standard} [[\text{flightinfo} (110, \text{special})]] \parallel \text{special} [[\text{saleinfo} (90, \text{bank})]] \\ & \parallel \text{bank} [[\text{till} (10)]], \end{aligned}$$

where:

$$\begin{aligned} \text{client} (\text{init}) = & \text{go}^5 \text{travelshop} \rightarrow \text{flight} ?(\text{standardoffer}) \\ & \rightarrow \text{go}^4 \text{standardoffer} \rightarrow \text{finfo2a} ?(p1) \rightarrow \text{finfo2b} ?(\text{specialoffer}) \\ & \rightarrow \text{go}^3 \text{specialoffer} \rightarrow \text{sinfo2a} ?(p2) \rightarrow \text{sinfo2b} ?(\text{paying}) \\ & \rightarrow \text{go}^6 \text{paying} \rightarrow \text{payc} !(\min\{p1, p2\}) \\ & \rightarrow \text{go}^4 \text{home} \rightarrow \text{client} (\text{init} - \min\{p1, p2\}) ; \\ \text{update} (\text{saleprice}) = & \text{go}^1 \text{special} \rightarrow \text{info1} !(\text{saleprice}) ; \\ \text{agent} (\text{balance}) = & \text{flight} !(\text{standard}) \\ & \rightarrow \text{go}^{20} \text{bank} \rightarrow \text{paya} ?(\text{profit}) \\ & \rightarrow \text{go}^{12} \text{travelshop} \rightarrow \text{agent} (\text{balance} + \text{profit}) ; \\ \text{flightinfo} (\text{price}, \text{next}) = & \text{finfo2a} !(\text{price}) \rightarrow \text{finfo2b} !(\text{next}) \\ & \rightarrow \text{flightinfo} (\text{price}, \text{next}) ; \\ \text{saleinfo} (\text{price}, \text{next}) = & \text{info1}^{\Delta 2} ?(\text{newprice}) \\ & \text{then } \text{sinfo2a} !(\text{newprice}) \rightarrow \text{sinfo2a} !(\text{next}) \rightarrow \text{saleinfo} (\text{newprice}, \text{next}) \\ & \text{else } \text{sinfo2a} !(\text{newprice}) \rightarrow \text{sinfo2a} !(\text{next}) \rightarrow \text{saleinfo} (\text{price}, \text{next}) ; \\ \text{till} (\text{cash}) = & \text{payc}^{\Delta 22} ?(\text{newpayment}) \\ & \text{then } \text{paya}^{10} !(\text{cash} + \text{newpayment}) \text{ then } \text{till} (0) \\ & \text{else } \text{till} (\text{cash} + \text{newpayment}) \\ & \text{else } \text{paya}^{10} !(\text{cash}) \text{ then } \text{till} (0) \\ & \text{else } \text{till} (\text{cash}) . \end{aligned}$$

3 Translating the High-Level Specification into Maude

In what follows we define a rewriting theory corresponding to the semantics of our high-level language defined in Table 3. The syntax used to give the rewriting theory is that of *real-time Maude*. A *rewrite theory* \mathcal{R} is defined as a triple (Σ, E, R) , where Σ stands for signature of function symbols, E and R are sets of Σ -equations and Σ -rewrite rules, respectively. The Σ -equations and Σ -rewrite rules can contain side conditions; for example, the conditions appearing in a rewrite rule can contain equations or other rewrite rules. Just like in [9], we use

a typed setting given as an order-sorted equational logic (Σ, E) including sorts and an inclusion relation *subsort* between sorts. Given a rewrite theory \mathcal{R} , we write $\mathcal{R} \vdash t \Rightarrow t'$ if $t \Rightarrow t'$ is provable in \mathcal{R} by using the rewrite rules of R . Rewriting logic is basically a computational logic that combines term rewriting with equational logic.

Let us discuss first the high-level recursion operator that is not directly encodable into real-time Maude (because infinite computations are not supported into this tool). Our solution is to use the construction $id(v, n)$ that is an extension of the constructions $id(v)$ of our language with a number n that limits the number of recursive calls to be executed during the evolution of the system.

In order to translate the high-level language (whose syntax is given in Table 1), we consider sorts corresponding to sets from our language: e.g., for the set *Chan* of channels, the sort **Channel** is created. Certain new aspects are provided by the sorts **AGuard** and **MGuard**. The sort **AGuard** contains the action parts $a^{\Delta t}!(v)$ and $a^{\Delta t}?(u)$ of the communication processes $a^{\Delta t}!(v)$ then P else Q and $a^{\Delta t}?(u)$ then P else Q , while the sort **MGuard** contains the action part $go^t l$ of the migration processes of the form $go^t l$ then P . The elements of the sorts **AGuard** and **MGuard** are essential in constructing the sequential processes of our language. Among the subsorting relations between the given sorts, we explain **subsorts Var < Location Channel Value** that illustrates the fact that location names, channel names or values can be used to instantiate variables. To work with multisets of values, we use the sort **MValue**.

```

sorts Location Channel Value MValue Var Process
      AGuard MGuard System .
subsorts Var < Location Channel Value < MValue .
subsort Location < Value .
subsorts System < GlobalSystem .

```

To each operator used in the syntax of Table 1 we attach the attribute **ctor** marking the fact that this operator is used to construct the system, and attribute **prec** followed by a number marking its applicability precedence with respect to other operators. Moreover, in order to encode properly into real-time Maude the parallel operators $|$ and $||$ from Table 1, we add to them the attributes **comm** and **assoc** to illustrate that they are commutative and associative constructors that respect the rules of Table 2.

```

op _^!<_> : Channel TimeInf Value -> AGuard [ctor prec 2] .
op _^?<_> : Channel TimeInf Var -> AGuard [ctor prec 2] .
op go^__ : TimeInf Location -> MGuard [ctor prec 2] .
op _then<_>else<_> : AGuard Process Process -> Process
  [ctor prec 1] .
op _then<_> : MGuard Process -> Process [ctor prec 1] .
op |_| : Process Process -> Process [ctor prec 4 comm assoc] .
op stop : -> Process [ctor] .
op _'[_]' : Location Process -> System [ctor prec 3] .
op _||_ : System System -> System [ctor prec 5 comm assoc] .
op void : -> System [ctor] .

```

As already stated, most of the rules of the structural congruence (Table 2) are encoded by using the attributes `comm` and `assoc` when defining the previous operators. For the rest of the rules we provide the following equations:

```

eq P | stop = P .
eq M || void = M .
eq k[[P | Q]] = (k[[P]]) || (k[[Q]]) .

```

As communication between two processes by using rule (COM)) leads to a substitution of variables by the communicated values, we need to define this operation explicitly in real-time Maude. Such an operator acts only upon the free occurrences of a name, while leaving bound names as they are.

```

op _'[_/_' : Process Value Var -> Process [prec 8] .

eq ((c ^ t ! < b + a >) then (P) else (Q)) { V / b } =
  ((c ^ t ! < V + a >) then (P { V / b }) else (Q { V / b })) .
eq ((c ^ t ! < min(b , a) >) then (P) else (Q)) { V / b } =
  ((c ^ t ! < min(V , a) >) then (P { V / b })
   else (Q { V / b })) .

eq ((c ^ t ! < X >) then (P) else (Q)) { V / X } =
  ((c ^ t ! < V >) then (P { V / X }) else (Q { V / X })) .
ceq ((c ^ t ! < W >) then (P) else (Q)) { V / X } =
  ((c ^ t ! < W >) then (P { V / X })
   else (Q { V / X })) if V /= W .
eq ((c ^ t ? ( X )) then (P) else (Q)) {V / X} =
  ((c ^ t ? ( X )) then (P) else (Q)) .
ceq ((c ^ t ? ( Y )) then (P) else (Q)) {V / X} =
  ((c ^ t ? ( Y )) then (P { V / X })
   else (Q { V / X })) if X /= Y .
eq ((go ^ t X) then (P)) {V / X} = ((go ^ t V) then (P {V / X})) .
ceq ((go ^ t l) then (P)) {V / X} =
  ((go ^ t l) then (P {V / X})) if X /= l .
eq (P | Q) {V / X} = ((P {V / X}) | (Q {V / X})) .
eq stop {V / X} = stop .
eq (P) { V / X } = (P) [owise] .

```

However, the above operator does not take into account the need for alpha-conversion in order to avoid name clashes once substitution takes place. To illustrate this issue, let us consider the process $P = a^t(b) \text{ then } (go^t l \text{ then } X) \text{ else } stop$ in which the name b is bound inside the input prefix. If the substitution $\{b/X\}$ needs to be performed over this process, the obtained process would be $P\{b/X\} = a^t(b) \text{ then } (go^t l \text{ then } b) \text{ else } stop$. This means that once variable X is replaced by the name b , name b would become bound not only in the input action. To avoid this, we define an operator able to perform alpha-conversion by using terms of the form $[X]$ that contain fresh names:

```

op '['[_]' : Var -> System [ctor] .

```

The terms of the form $[X]$ containing fresh names are composed with the system by using the parallel operator $||$. Using the given fresh names, the renaming is done (when necessary) before substitution. This is provided by the operator:

```

op _'[_/_' : Process Value Var -> Process [prec 8] .

```

This operator has a definition similar with the substitution operator, except the case when we deal with bound names.

$$\text{eq } ((c \hat{=} t ? (X)) \text{ then } (P) \text{ else } (Q)) (V / X) = \\ ((c \hat{=} t ? (V)) \text{ then } (P \{ V / X \}) \text{ else } (Q \{ V / X \})) .$$

It is worth noting that this is different from the substitution operator that does not allow the change of the bound name:

$$\text{eq } ((c \hat{=} t ? (X)) \text{ then } (P) \text{ else } (Q)) \{V / X\} = \\ ((c \hat{=} t ? (X)) \text{ then } (P) \text{ else } (Q)) .$$

As most of the rules in Table 3 contain hypotheses, translating these rules in real-time Maude requires the use of conditional rewrite rules in which the conditions are the hypotheses of rules of Table 3. Notice that in what follows we do not directly implement the rules (PAR), (DEQUIV) and (EQUIV) as rewrite rules into real-time Maude, due to the fact that the commutativity, associativity and the congruence rewriting of the parallel operators $|$ and $||$ are already encoded into the matching mechanism of Maude. In order to identify for each of the below rewrite rule which rule from Table 3 it models, we consider simple intuitive names for these rewrite rules. More complicated names could be considered by using rewriting rules similar with the ones given for the executable specification of the π -calculus in Maude [15].

```

crl [Comm] : (k[[ (c ^ t ! < V >) then (P) else (Q) ]])
  || (k[[ (c ^ t' ? (X)) then (P') else (Q') ]])
  => (k [[ P ]]) || (k [[ P' {V / X} ]]) if notin(V , bnP(P')) .
crl [Comm'] : (([Z]) || (k[[ (c ^ t ! < V >) then (P) else (Q) ]]))
  || (k[[ (c ^ t' ? (X)) then (P') else (Q') ]])
  => (([X]) || (k [[ P ]])) || (k [[ (P' (Z / V)) { V / X} ]])
  if in(V , bnP(P')) /\ (notin(Z , bnP(P'))) .
crl [Input0] : (k[[ (c ^ t ! < V >) then (P) else (Q) ]]) => k[[Q]]
  if t == 0 .
crl [Output0] : (k[[ (c ^ t ? (X)) then (P) else (Q) ]]) => k[[Q]]
  if t == 0 .
crl [Move] : k[[ (go ^ t l) then (P) ]]) => l[[P]] if t == 0 .

```

It is also worth noting that there are two instances for the rule [Comm]. This is a consequence of the fact that after communication, before a substitution takes place, one may need to perform alpha-conversion to avoid name clashes. Rule [Comm] is applicable if the variable V is not bound inside process P (modelled by the condition $\text{notin}(V, \text{bnP}(L'))$), and so only a simple substitution is enough to complete the replacement of the variable X by name V . On the other hand, rule [Comm'] is applicable if the variable V is bound inside process P (modelled by the condition $\text{in}(V, \text{bnP}(L'))$); in this case an alpha-conversion is needed to avoid the clash of name V . To be able to perform the alpha-conversion we also check before applying the rule [Comm'] if a fresh name $[Z]$ exists in the system, name not present in process P' (modelled by the condition $\text{notin}(Z, \text{bnP}(Q))$).

The conditions of the rules [Comm] and [Comm'] make use of the functions `in`, `notin` and `bnP` for checking the membership of a name to the set of bound names for a given process.

A tick rewriting rule is used to model the passing of time in the encoded system by a positive amount of time that is at most equal with the maximal times that can elapse in the system. Such a tick rule has the form:

```
crl [tick] : {M} => {delta(M, t)} in time t if t <= mte(M) [nonexec] .
```

The [tick] rule uses the function `delta` to decrease all time constraints in a system by the same positive value. In order to correctly model the steps needed to obtain complete computational steps, namely the time cannot elapse if rewrite rules are applicable, we use the `frozen` attribute for the function `delta`. The attribute (1) marks the argument to be frozen (first one in this case).

```
op delta : System TimeInf -> System [frozen (1)] .
eq delta (k[[c ^ t ! < V >] then (P) else (Q) ]) , t' =
  k[[c ^ (t monus t') ! < V >] then (P) else (Q) ]) .
eq delta (k[[c ^ t ? ( X )] then (P) else (Q) ]) , t' =
  k[[c ^ (t monus t') ? ( X )] then (P) else (Q) ]) .
eq delta (k[[go ^ t 1] then (P)]) , t' =
  k[[go ^ (t monus t') 1] then (P)] .
eq delta (k[[P | Q]] , t') = delta (k[[P]] , t')
  || delta (k[[Q]] , t') .
eq delta (M || N , t') = delta(M , t') || delta(N , t') .
eq delta (void , t') = void .
eq delta (1[[stop]] , t') = 1[[stop]] .
eq delta (M , t') = M [owise] .
```

The function `mte` from the condition of rule [tick] is used to compute the maximal time that can be elapsed in a system, a time that is equal with the minimum time constraint of the applicable actions in the system.

```
op mte : System -> TimeInf [frozen (1)] .
eq mte (k[[c ^ t ! < V >] then (P) else (Q) ]) = t .
eq mte (k[[c ^ t ? ( X )] then (P) else (Q) ]) = t .
eq mte (k[[go ^ t 1] then (P)]) = t .
eq mte (k[[P | Q]] ) = min(mte (k[[P]]), mte (k[[Q]])) .
eq mte (M || N) = min(mte(M) , mte(N)) .
eq mte (void) = INF .
eq mte (1[[stop]]) = INF .
eq mte (M) = INF [owise] .
```

The full description of the translation into real-time Maude is available at <https://profs.info.uaic.ro/~bogdan.aman/RTMaude/TiMoSpec.rtmaude> .

In order to study the correspondence between the operational semantics of our high-level specification language and that of the real-time Maude, we inductively define a mapping $\psi : \text{TtMO} \rightarrow \text{System}$ as

$$\psi(M) = \begin{cases} l[[\varphi(P)]] & \text{if } M = l[[P]] \\ \psi(N_1) \parallel \psi(N_2) & \text{if } M = N_1 \parallel N_2; \\ \text{void} & \text{if } M = \mathbf{0} \end{cases}$$

$$\varphi(P) = \begin{cases} a^{\Delta t}!\langle v \rangle \text{ then } \varphi(R) \text{ else } \varphi(Q) & \text{if } P = a^{\Delta t}!\langle v \rangle \text{ then } R \text{ else } Q \\ a^{\Delta t}?(X) \text{ then } \varphi(R) \text{ else } \varphi(Q) & \text{if } P = a^{\Delta t}?(X) \text{ then } R \text{ else } Q \\ (go^t l) . \varphi(R) & \text{if } P = go^t l \text{ then } R \\ \text{stop} & \text{if } P = \mathbf{0} \\ \varphi(Q) \mid \varphi(R) & \text{if } P = Q \mid R \\ \varphi(R)\{v/u\} & \text{if } P = R\{v/u\} \text{ and } v \notin bn(R) \\ \varphi(R)(Z/v)\{v/u\} & \text{if } P = R\{v/u\} \text{ and } v \in bn(R) \\ & \text{and } Z \notin bn(R). \end{cases}$$

By \mathcal{R}_D we denote the rewrite theory defined previously in this section by the rewrite rules `[Comm]`, `[Comm']`, `[Input0]`, `[Output0]`, `[Move]` and `[tick]`, and also by the additional operators and equations appearing in these rewrite rules.

The next result relates the structural congruence of the high-level specification language with the equational equality of the rewrite theory.

Lemma 1. $M \equiv N$ if and only if $\mathcal{R}_D \vdash \psi(M) = \psi(N)$.

Proof. \Rightarrow : By induction on the congruence rules of our high-level language.

\Leftarrow : By induction on the equations of the rewrite theory \mathcal{R}_D .

The next result emphasizes the operational correspondence between the high-level systems M , N and their translations into a rewriting theory. We denote by $M \rightarrow N$ any rule of Table 3.

Theorem 3. $M \rightarrow N$ if and only if $\mathcal{R}_D \vdash \psi(M) \Rightarrow \psi(N)$.

Proof. \Rightarrow : By induction on the derivation $M \rightarrow N$.

- (Com): We have $M = l[[a^{\Delta t}!\langle v \rangle \text{ then } P \text{ else } Q]] \parallel l[[a^{\Delta t}?(u) \text{ then } P' \text{ else } Q']]$ and $N = l[[P]] \parallel l[[P'\{v/u\}]]$. By definition of ψ , we obtain $\psi(M) = l[[a^{\Delta t}!\langle v \rangle \text{ then } \varphi(P) \text{ else } \varphi(Q)]] \parallel l[[a^{\Delta t}?(u) \text{ then } \varphi(P') \text{ else } \varphi(Q')]]$. Depending on the fact that v appears or not as a bound name in P' , we have two cases:
 - if $v \notin bn(P')$: By applying `[Comm]`, we have $\mathcal{R}_D \vdash \psi(M) \Rightarrow l[[\varphi(P)]] \parallel l[[\varphi(P')\{v/u\}]] = N'$, and by the definition of ψ , we have $\psi(N) = N'$.
 - if $v \in bn(P')$: We should apply first an alpha-conversion before the value is communicated. This is done by using a fresh name $[Z]$ such that by applying the rule `[Comm']` we get $\mathcal{R}_D \vdash [Z] \parallel \psi(M) \Rightarrow [[v]] \parallel l[[\varphi(P)]] \parallel l[[\varphi(P')(Z/v)\{v/u\}]] = N'$. By the definition of ψ , we have $\psi(N) = N'$.

- (Move0), (Put0) and (Get0): These cases are similar to the previous one, by using the rules [Move], [Input0] and [Output0], respectively.
- (DMove): We have that $M = l[[go^t l' \text{ then } P]]$ and $N = l[[go^{t-t'} l' \text{ then } P]]$. By definition of ψ , we obtain $\psi(M) = l[[go^t l' . \varphi(P)]]$. By applying the rule [tick] we get $\mathcal{R}_D \vdash \psi(M) \Rightarrow l[[go^{t-t'} l' . \varphi(P)]] = N'$. By definition of ψ , we have $\psi(N) = N'$.
- (DStop), (DPut) and (DGet): These cases are similar to the previous one, by using also the rule [tick].
- The rest of the rules are simulated using the implicit constructors of Maude.

\Leftarrow : By induction on the derivation $\mathcal{R}_D \vdash \psi(M) \Rightarrow \psi(N)$.

- [Comm]: We have $\psi(M) = l[[a^{\Delta t}! \langle v \rangle \text{ then } P \text{ else } Q]] \parallel l[[a^{\Delta t'} ? \langle u \rangle \text{ then } P' \text{ else } Q']]$ and $\psi(N) = l[[P]] \parallel l[[P'\{v/u\}]]$. According to the definition of ψ , we get $M = l[[a^{\Delta t}! \langle v \rangle \text{ then } P_1 \text{ else } Q_1]] \parallel l[[a^{\Delta t'} ? \langle u \rangle \text{ then } P'_1 \text{ else } Q'_1]]$, where $P = \varphi(P_1)$ and $Q = \varphi(Q_1)$. By applying (Com), we get $M \rightarrow l[[P_1]] \parallel l[[Q_1\{v/u\}]] = N'$. By definition of ψ , we have $N = N'$.
- The other rules are treated in a similar manner.

4 Analyzing Timed Mobile Agents by Using Maude Tools

We have the translation of the high-level specification of the multi-agent systems into real-time Maude rewriting system, and have also the operational correspondence between their semantics. The *TravelShop* system presented in Example 1 can now be described in real-time Maude. The entire system looks like this:

```

eq TravelShop = home[[client(130 , 1)]]
                || travelshop[[agent(100 , 1) | update(60 , 1)]]
                || standard[[flightinfo(110 , special , 1)]]
                || special[[saleinfo(90 , bank , 1)]]
                || bank[[till(10 , 1)]] .

```

where, e.g., the *client* syntax in real-time Maude is:

```

ceq client(init , applyC)=
((go ^ 5 travelshop)
 then ((flight ^ INF ? ( standardoffer ))
  then ((go ^ 4 standardoffer)
   then ((finfo2a ^ INF ? ( p1 ))
    then ((finfo2b ^ INF ? ( specialoffer ))
     then ((go ^ 3 specialoffer)
      then ((sinfo2a ^ INF ? ( p2 ))
       then ((sinfo2b ^ INF ? ( paying ))
        then ((go ^ 6 paying)
         then ((payc ^ INF ! < min(p1 , p2) >)
          then ((go ^ 4 home)
           then (client(sd(init,min(p1,p2)),applyC minus 1))
          else (stop) ) )
        )
       )
      )
     )
    )
   )
  )
 )
)

```

```

        else (stop) )
      else (stop) ) )
    else (stop) )
  else (stop) ) )
if applyC >= 1 .

```

Since the recursion operator cannot be directly encoded into real-time Maude, we include for each recursion process appearing in *TravelShop* system a second parameter saying how many times the process can be unfolded. For our example this is 1 (but it could be any finite value).

Before doing any verification, we have the possibility in real-time Maude to define the length of the time units performed by the whole system. For our example we choose a time unit of length 1 by using the following command:

```
(set tick def 1 .)
```

When using the rewrite command (`frew {TravelShop} in time < 38 .`), the Maude platform executes *TravelShop* by using the equations and rewrite rules of \mathcal{R}_D as given in the previous section, and outputs the following result:

```
Timed fair rewrite {TravelShop} in Example with mode default time
  increase 1 in time < 38
```

```
Result ClockedSystem :
  {bank[[till(0,0)]|| home[[client(70,0)]|| special[[stop]]
  || special[[saleinfo(60,bank,0)]
  || standard[[flightinfo(110,special,0)]
  || travelshop[[agent(170,0)]]} in time 37
```

```
rewrites: 786514 in 404ms cpu (406ms real) (1946816 rewrites/second)
```

We use the real-time Maude platform to perform timed reachability tests, namely if starting from the initial configuration of a system one can reach a given configurations of the system before a time threshold. The real-time Maude is able to provide answers to such inquires by searching into the state space obtained into the given time framework for the given configuration. As we are interested in searching the appearance of the given configuration within a time-framework, the fact that multiple computational steps can be performed is marked by the use of the `=>*`. Also, the annotation `[n]` bounds the number of performed computational steps to n , thus reducing the possible state space.

```
(tsearch [2] {TravelShop} =>* {bank[[till(0,0)]|| home[[client(70,0)]
  || special[[stop]]||special[[saleinfo(60,bank,0)]
  || standard[[flightinfo(110,special,0)]
  || travelshop[[agent(170,0)]]} in time < 40 . )
```

The result of performing the above inquiry is:

```
Timed search [2] in Example
  {TravelShop} =>* {bank[[till(0,0)]|| home[[client(70,0)]]
  || special[[stop]]|| special[[saleinfo(60,bank,0)]]
  || standard[[flightinfo(110,special,0)]]
  || travelshop[[agent(170,0)]]}
in time < 40 and with mode default time increase 1 :
```

```
Solution 1
TIME_ELAPSED:Time --> 37
```

```
Solution 2
TIME_ELAPSED:Time --> 38
```

```
rewrites: 3684 in 24ms cpu (25ms real) (153500 rewrites/second)
```

Instead of searching for the entire reachable system, we can also search only for certain parts of it: for instance, to check when the *client* remains with 70 cash units in a given interval of time. This can be done by using the command:

```
(tsearch {TravelShop} =>* {home[[client(70,0)]] || X:System}
in time-interval between >= 22 and < 40 . )
```

The answer returns that there exists such a situation at time 22.

```
Timed search [1] in Example
{TravelShop} =>* {home[[client(70,0)]]|| X:System}
in time between >= 22 and < 40 and with mode default time increase 1 :
```

```
Solution 1
TIME_ELAPSED:Time --> 22 ; X:System --> bank[[paya ^ 6 ! < 70 >) then
(till(0,0))else(till(70,0))]]|| special[[stop]]|| special[[saleinfo(60,
bank,0)]]|| standard[[flightinfo(110,special,0)]]|| travelshop[[go ^
3 bank) then ((paya ^ INF ?(profit)) then ((go ^ 12 travelshop) then
(agent(profit + 100,0))else(stop))]]
```

The real-time Maude tool allows also the following command to find the shortest time to reach a desired configuration:

```
(find earliest {TravelShop} =>* {home[[client(70,0)]] || X:System} . )
```

It returns the same solution as the previous one, and tells that it was reached in time 22.

If time is not relevant for such a search, we can use the *untimed* search command:

```
(utsearch [1] {TravelShop} =>* {home[[client(70,0)]] || X:System} . )
```


5 Conclusion and Related Work

In the current paper we translated our high-level specifications of the multi-agent systems with timeouts for migration and communication into an existing rewriting engine able to execute and analyze timed systems. This translation satisfies an operational correspondence result. Thus, such a translation is suitable for analyzing complex multi-agent systems with timeouts in order to be sure that they have the expected behaviours and properties. We analyze the multi-agent systems with timeouts by using the real-time Maude software platform. The approach is illustrated by an example.

The used high-level specification is given in a language forthcoming realistic programming systems for multi-agent systems, a language with explicit locations and timeouts for migration and communication. It is essentially a simplified version of the timed distributed π -calculus [7]. It can be viewed as a prototyping language of the TiMO family for multi-agent systems in which the agents can migrate between explicit locations in order to perform local communications with other agents. The initial version of TiMO presented in [5] led to some extensions; e.g., with access permissions in perTiMO [6], with real-time in rTiMO [2]. In [4] it was presented a Java-based software in which the agents are able to perform timed migration just like in TiMO. Using the model checker Process Analysis Toolkit (PAT), the tool TiMO@PAT [8] was created to verify timed systems. In [16], the authors consider an UTP semantics for rTiMO in order to provide a different understanding of this formalism. Maude is used in [17] to define a rewrite theory for the BigTiMO calculus, a calculus for structure-aware mobile systems combining TiMO and the bigraphs [12]. However, the authors of [17] do not tackle the fresh names and recursion problems presented in our current approach.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theoret. Comput. Sci.* **126**, 183–235 (1994)
2. Aman, B., Ciobanu, G.: Real-time migration properties of rTiMO verified in UPPAAL. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 31–45. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40561-7_3
3. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
4. Ciobanu, G., Juravle, C.: Flexible software architecture and language for mobile agents. *Concurrency Comput. Pract. Experience* **24**, 559–571 (2012)
5. Ciobanu, G., Koutny, M.: Timed mobility in process algebra and Petri nets. *J. Logic Algebraic Program.* **80**, 377–391 (2011)
6. Ciobanu, G., Koutny, M.: Timed migration and interaction with access permissions. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 293–307. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_23

7. Ciobanu, G., Prisacariu, C.: Timers for distributed systems. *Electron. Not. Theor. Comput. Sci.* **164**, 81–99 (2006)
8. Ciobanu, G., Zheng, M.: Automatic analysis of TiMo systems in PAT. In: IEEE Computer Society Proceedings 18th Engineering of Complex Computer Systems (ICECCS), pp. 121–124 (2013)
9. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoret. Comput. Sci.* **105**, 217–273 (1992)
10. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: testing real-time systems using UPPAAL. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) *Formal Methods and Testing*. LNCS, vol. 4949, pp. 77–117. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_3
11. Meseguer, J.: Twenty years of rewriting logic. In: Ölveczky, P.C. (ed.) *WRLA 2010*. LNCS, vol. 6381, pp. 15–17. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16310-4_2
12. Milner, R.: *The Space and Motion of Communicating Agents*. Cambridge University Press, Cambridge (2009)
13. Ölveczky, P.C.: Real-time Maude and its applications. In: Escobar, S. (ed.) *WRLA 2014*. LNCS, vol. 8663, pp. 42–79. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12904-4_3
14. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of real-time Maude. *Higher-Order Symbolic Comput.* **20**, 161–196 (2007)
15. Thati, P., Sen, K., Martí-Oliet, N.: An executable specification of asynchronous π -calculus semantics and may testing in Maude 2.0. *Electron. Not. Theor. Comput. Sci.* **71**, 261–281 (2004)
16. Xie, W., Xiang, S.: UTP semantics for rTiMo. In: Bowen, J.P., Zhu, H. (eds.) *UTP 2016*. LNCS, vol. 10134, pp. 176–196. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52228-9_9
17. Xie, W., Zhu, H., Zhang, M., Lu, G., Fang, Y.: Formalization and verification of mobile systems calculus using the rewriting engine Maude. In: *IEEE 42nd Annual Computer Software and Applications Conference*, pp. 213–218 (2018)