Multiparty Session Types in Distributed Systems With Migration

Bogdan Aman and Gabriel Ciobanu

Romanian Academy, Institute of Computer Science, Iaşi, Romania
"A.I.Cuza" University, Blvd. Carol I no.11, 700506 Iaşi, Romania
bogdan.aman@gmail.com, gabriel@info.uaic.ro

Abstract. We propose a multiparty session type system for distributed systems involving timed migration between explicit locations and local communications. We use TiMo (Timed Mobility) as the process calculus over which we define the session types, and investigate certain scenarios in which processes are required to behave according to these types. The typing system ensures some properties including type soundness, communication and migration safety.

1 Introduction

In the current distributed systems there exist a high number of mobile entities moving between different locations and interacting with other entities to exchange data through communication. It is necessary to go beyond the sequential computation of the λ-calculus, even beyond the concurrent processes of the π-calculus. It is also worth to consider timed processes and explicit locations for migration. This is why we use a process calculus named TiMo able to describe migrating processes in a unified framework working with timers, explicit locations and local clocks in distributed systems [10]. We can see TiMo as a prototyping language for multi-agent systems, featuring mobility and local interaction. Multi-agent systems typically consist of a large number of agents which exhibit autonomic behaviour depending on their timeouts and actions. The mobility of agents and interaction between the agents through communication may introduce new and sometimes unexpected behaviours. Components can be highly heterogeneous, each operating at different temporal scales and having different objectives. Analysing these systems is becoming more and more necessary because they are really complex and increasingly used in various critical application domains such as e-commerce and distributed collaborative systems. Thus, it is important to have some modelling techniques which are able to describe such systems, and to reason about their behaviour in both qualitative and quantitative terms. To move towards this goal, we consider that it is important to develop a specific typing system.

In distributed systems, the behavioural types were introduced to secure the compatibility of interaction patterns among processes [23]. The behavioural type of a process specifies its expected interactions by using a simple type language,

and so determining a correct evolution. Building secure mobile environments requires solutions for a lot of problems due to the inherent timed migration and communication of the computing entities. We extend TiMo with session types able to reason over migrating and communicating processes and their timed behaviour. The idea is to use these session types to coordinate the communicating processes inside sessions (represented by the locations of our calculus) by representing the trace of the channels usage as a structured sequences of types. The migration capabilities represent the change of the current session for a user, when different sessions are active and each session is independently characterised by corresponding global types.

Global types [24] are specifications of the interactions between the processes that participate in a session. For each session represented in TiMo by the involved distributed locations, there exist a global type specifying its evolution. Also, for each participant uniquely identified by a natural number, there exist a global type that illustrates the movement of the participant between different sessions during the evolution of the whole system. Each global type associated to a session can be projected onto a set of local types describing the session from the perspective of each single participant. As done in [4], we use timed global types and local session types, but with some differences explained in what follows.

In this paper we consider systems in which processes may be engaged in different sessions during their evolutions, where each session is independently characterised by corresponding global types. Also, in our approach an arbitrary number of participants can dynamically join or leave sessions (represented as locations in our calculus) during their evolutions, and so we do not require session creation as is usually done in session types.

2 TiMo : Syntax and Semantics

In TiMo the processes can migrate between different locations of a distributed environment consisting of a number of explicit distinct locations. Timing constraints over migration and communication actions are used to coordinate processes in time and space. The passage of time in TiMo is described with respect to a global clock, while migration and communication actions are performed in a maximal parallel manner. Timing constraints for migration allow one to specify a temporal timeout after which a mobile process must move to another location. Two processes may communicate only if they are present at the same location. In TiMo, the transitions caused by performing actions with timeouts are alternated with continuous transitions. The semantics of TiMo is provided by multiset labelled transitions in which multisets of actions are executed in parallel (in one step).

Timing constraints applied to mobile processes allow us to specify how many time units are required by a process to move from one location to another. A timer in TiMo is denoted by $^{\Delta t}$, where $t \in \mathbb{N}$. Such a timer is associated with a migration action such as $go^{t}bridge$ then P indicating that process P moves to location *bridge* after t time units. A timer $^{\Delta 5}$ associated with an output

communication process $a^{\Delta 5}!\langle z\rangle$ then P else Q makes the channel a available for communication (namely it can send z) for a period of 5 time units. It is also possible to restrict the waiting time for an input communication process $a^{\Delta 4}?(x)$ then P else Q along a channel a; if the interaction does not happen before the timeout 4, the process gives up and continues as the alternative process Q.

The syntax of TiMo is given in Table 1, where the following are assumed:

- a set *Loc* of locations, a set *Chan* of communication channels, and a set *Id* of process identifiers (each $id \in Id$ has its arity m_{id});
- for each $id \in Id$ there is a unique process definition $id(u_1, \ldots, u_{m_{id}}) \stackrel{def}{=} P_{id}$, where the distinct variables u_i are parameters;
- $a \in Chan$ is a communication channel; l is a location or a location variable;
- $t \in \mathbb{N}$ is a *timeout* of an action; u is a tuple of variables;
- v is a tuple of expressions built from values, variables and allowed operations.

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

Table 1. TiMo Syntax

Shorthand notation:

$a\,!\,\langle \boldsymbol{v}\rangle \ \rightarrow P$ will be used to denote $a^{\Delta\infty}\,!\,\langle \boldsymbol{v}\rangle$ `then` P `else stop`
$a\,?\,(\boldsymbol{u}) \ \rightarrow P$ will be used to denote $a^{\Delta\infty}\,?\,(\boldsymbol{u})$ `then` P `else stop`

The only binding constructor is $a^{\Delta t}?(u)$ then P else Q that binds the variable u within P (but *not* within Q). $fv(P)$ is used to denote the free variables of a process P (and similarly for systems); for a process definition, it is assumed that $fv(P_{id}) \subseteq \{u_1, \ldots, u_{m_{id}}\}$, where u_i are the process parameters. Processes are defined up-to an alpha-conversion, and $P\{v/u, \ldots\}$ denotes P in which all free occurrences of the variable u are replaced by v, eventually after alpha-converting P in order to avoid clashes.

Mobility is provided by a process $go^t l$ then P that describes the migration from the current location to the location indicated by l after t time units. Since l can be a variable, and so its value is assigned dynamically through communication with other processes, this form of migration supports a flexible scheme for the movement of processes from one location to another. Thus, the behaviour can adapt to various changes of the distributed environment. Processes are further constructed from the (terminated) process 0, and parallel composition $P \mid Q$. A located process $l[[P]]$ specifies a process P running at location l, and a system is built from its components $L \mid N$. A system N is well-formed if there are no free variables in N.

2.1 Operational Semantics of TiMo

The first component of the operational semantics of TiMo is the structural equivalence $\equiv$ over systems. The structural equivalence is the smallest congruence such that the equalities in Table 2 hold.

(PNULL)	$P \mid 0 \equiv P$
(NNULL)	$N \mid \mathbf{0} \equiv N$
(NCOMM)	$N \mid N' \equiv N' \mid N$
(NASSOC)	$(N \mid N') \mid N'' \equiv N \mid (N' \mid N'')$
(NSPLIT)	$l[[P \mid Q]] \equiv l[[P]] \mid l[[Q]]$

Table 2. TiMo Structural Congruence

Essentially, the role of $\equiv$ is to rearrange a system in order to apply the rules of the operational semantics given in Table 3. Using the equalities of Table 2, a given system N can always be transformed into a finite parallel composition of located processes of the form $l_1[[P_1]] \mid \ldots \mid l_n[[P_n]]$ such that no process P_i has the parallel composition operator at its topmost level. Each located process $l_i[[P_i]]$ is called a component of N, and the whole expression $l_1[[P_1]] \mid \ldots \mid l_n[[P_n]]$ is called a *component decomposition* of the system N.

The operational semantics rules of TiMo are presented in Table 3. The multiset labelled transitions of form $N \xrightarrow{\Lambda} N'$ use a multiset Λ to indicate the actions executed in parallel in one step. When the multiset Λ contains only one action λ, in order to simplify the notation, $N \xrightarrow{\{\lambda\}} N'$ is simply written as $N \xrightarrow{\lambda} N'$. The transitions of form $N \overset{t}{\rightsquigarrow} N'$ represent a time step of length t.

(Stop)	$l[[0]] \overset{\lambda}{\not\rightarrow}$ (DStop) $l[[0]] \overset{t}{\rightsquigarrow} l[[0]]$
(DMove)	if $t \geq t'$ then $l[[go^t l' \text{ then } P]] \overset{t'}{\rightsquigarrow} l[[go^{t-t'} l' \text{ then } P]]$
(Move0)	$l[[go^0 l' \text{ then } P]] \xrightarrow{l \triangleright l'} l'[[P]]$
(Com)	$l[[a^{\Delta t}!\langle v\rangle \text{ then } P \text{ else } Q]] \mid l[[a^{\Delta t'}?(u) \text{ then } P' \text{ else } Q']] \xrightarrow{\{v/u\}@l} l[[P]] \mid l[[P'\{v/u\}]]$
(DPut)	if $t \geq t' > 0$ then $l[[a^{\Delta t}!\langle v\rangle \text{ then } P \text{ else } Q]] \overset{t'}{\rightsquigarrow} l[[a^{\Delta t - t'}!\langle v\rangle \text{ then } P \text{ else } Q]]$
(Put0)	$l[[a^{\Delta 0}!\langle v\rangle \text{ then } P \text{ else } Q]] \xrightarrow{a!^{\Delta 0}@l} l[[Q]]$
(DGet)	if $t \geq t' > 0$ then $l[[a^{\Delta t}?(u) \text{ then } P \text{ else } Q]] \overset{t'}{\rightsquigarrow} l[[a^{\Delta t - t'}?(u) \text{ then } P \text{ else } Q]]$
(Get0)	$l[[a^{\Delta 0}?(u) \text{ then } P \text{ else } Q]] \xrightarrow{a?^{\Delta 0}@l} l[[Q]]$
(DCall)	if $l[[P_{id}\{v/x\}]] \overset{t}{\rightsquigarrow} l[[P'_{id}]]$ and $id(v) \overset{def}{=} P_{id}$ then $l[[id(v)]] \overset{t}{\rightsquigarrow} l[[P'_{id}]]$
(Call)	if $l[[P_{id}\{v/x\}]] \xrightarrow{id@l} l[[P'_{id}]]$ and $id(v) \overset{def}{=} P_{id}$ then $l[[id(v)]] \xrightarrow{id@l} l[[P'_{id}]]$
(DPar)	if $N_1 \overset{t}{\rightsquigarrow} N'_1$, $N_2 \overset{t}{\rightsquigarrow} N'_2$ and $N_1 \mid N_2 \overset{\lambda}{\not\rightarrow}$ then $N_1 \mid N_2 \overset{t}{\rightsquigarrow} N'_1 \mid N'_2$
(Par)	if $N_1 \xrightarrow{\Lambda_1} N'_1$ and $N_2 \xrightarrow{\Lambda_2} N'_2$ then $N_1 \mid N_2 \xrightarrow{\Lambda_1 \cup \Lambda_2} N'_1 \mid N'_2$
(DEquiv)	if $N \equiv N'$, $N' \overset{t}{\rightsquigarrow} N''$ and $N'' \equiv N'''$ then $N \overset{t}{\rightsquigarrow} N'''$
(Equiv)	if $N \equiv N'$, $N' \xrightarrow{\Lambda} N''$ and $N'' \equiv N'''$ then $N \xrightarrow{\Lambda} N'''$

Table 3. TiMo Operational Semantics

In rule (MOVE0), the process $go^0 l'$ then P migrates from location l to location l' and evolves as process P. In rule (COM), a process $a^{\Delta t}!\langle v\rangle$ then P else Q located at location$l$, succeeds in sending a tuple of values v over channel a to process $a^{\Delta t}?(u)$ then P' else Q' also located at l. Both processes continue to execute at location l, the first one as P and the second one as $P'\{v/u\}$. If a communication action has a timer equal to 0, then by using the rule (PUT0) for output action or the rule (GET0) for input action, the generic process $a^{\Delta 0} * $ then P else Q where $* \in \{!\langle v\rangle, ?(x)\}$ continues as the process Q. Rule (CALL) describes the evolution of a recursion process. The rules (EQUIV) and (DEQUIV) are used to rearrange a system in order to apply a rule. Rule (PAR) is used to compose larger systems from smaller ones by putting them in parallel, and considering the union of multisets of actions.

The rules devoted to the passing of time are starting with D. For instance, in rule (DPAR), $N_1 \mid N_2 \stackrel{\lambda}{\not\rightarrow}$ means that no action λ (i.e, an action labelled by $l' \triangleright l$, $\{v/u\}@l$, $id@l$, $go^{\Delta 0}@l$, $a?^{\Delta 0}@l$ or $a!^{\Delta 0}@l$) can be applied in the system $N_1 \mid N_2$. Negative premises are used to denote the fact that the passing to a new step is performed based on the absence of actions; the use of negative premises does not lead to an inconsistent set of rules.

A complete computational step is captured by a derivation of the form:

$$N \stackrel{\Lambda}{\longrightarrow} N_1 \stackrel{t}{\rightsquigarrow} N'.$$

This means that a complete step is a parallel execution of individual actions of Λ followed by a time step. Performing a complete step $N \stackrel{\Lambda}{\longrightarrow} N_1 \stackrel{t}{\rightsquigarrow} N'$ means that N' is directly reachable from N. If there is no applicable action ($\Lambda = \emptyset$), $N \stackrel{\Lambda}{\longrightarrow} N_1 \stackrel{t}{\rightsquigarrow} N'$ is written $N \stackrel{t}{\rightsquigarrow} N'$ to indicate (only) the time progress.

Proposition 1. *For all systems N, N' and N'',*

$$\text{if } N \stackrel{t}{\rightsquigarrow} N' \text{ and } N \stackrel{t}{\rightsquigarrow} N'', \text{ then } N' \equiv N''.$$

Proposition 1 states that the passage of time does not introduce any nondeterminism into the execution of a process.

Proposition 2. *For all systems N, N' and N'',*

$N \stackrel{(t+t')}{\rightsquigarrow} N'$ *if and only if there is a N'' such that* $N \stackrel{t}{\rightsquigarrow} N''$ *and* $N'' \stackrel{t'}{\rightsquigarrow} N'$.

Proposition 2 states that whenever a process is able to evolve for a certain time t, then it must evolve through every time moment before t; this ensures that the process evolves continuously.

3 Example

Time issues and mobility are essential in several systems in which a correct evolution depends not only on the actions taken, but also when and where the actions happen. A system may crash if an action is taken too early or too late, or in an inappropriate location. We illustrate how TIMO works by a *TravelShop* example used also in [12]. In this example, a client process attempts to pay as

little as possible for a ticket to a predefined destination. The scenario involves five locations and six processes. The role of each of the locations is as follows:

- *home* is a location where the client process starts and ends its journey;
- *travelshop* is the main location; it is initially visible to the client;
- *standard* and *special* are two locations of the service where clients can find out about the ticket prices;
- *bank* is a location where the payment is made.

The role of each of the processes is as follows:

- *client* is a process which initially resides in the *home* location, and is determined to pay for a flight after comparing two offers (standard and special) provided by the travel shop. Upon entering the travel shop, *client* receives the location of the standard offer and, after moving there and obtaining this offer, the client is given the location where a special offer can be obtained. After that, *client* moves to the bank and pays for the cheaper of the two offers, and then returns to *home*.
- *agent* first informs *client* where to look for the standard offer and then moves to *bank* in order to collect the money from the till. After that *agent* returns back to *travelshop*.
- *flightinfo* communicates the standard offer to clients as well as the location of the special offer.
- *saleinfo* communicates the special offer to clients together with the location of the bank. *saleinfo* can also accept an update of the special offer by the travel shop.
- *update* initially resides at the *travelshop* location and then migrates to *special* in order to update the special offer.
- *till* resides at the *bank* location and can either receive e-money paid in by clients, or transfer the e-money accumulated so far to *agent*.

Figure 1 describes schematically the evolution of the *TravelShop* system.

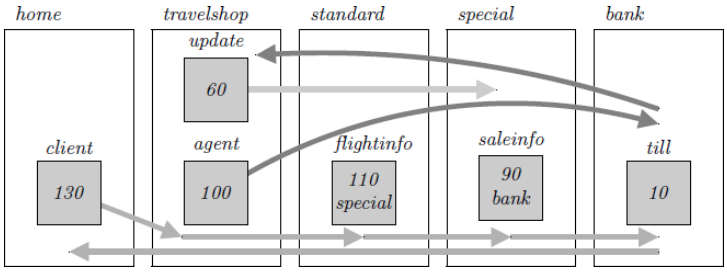

Fig. 1. The agency example

The specification of the running example in TiMo is given in Table 4.

$$
\begin{array}{l}
\mathit{TravelShop} \stackrel{\text{df}}{=} \\
\quad \mathit{home}\,[\![\, \mathit{client}(130) \,]\!] \mid \mathit{travelshop}\,[\![\, \mathit{agent}(100) \mid \mathit{update}(60) \,]\!] \mid \\
\quad \mathit{standard}\,[\![\, \mathit{flightinfo}(110, \mathit{special}) \,]\!] \mid \mathit{special}\,[\![\, \mathit{saleinfo}(90, \mathit{bank}) \,]\!] \mid \\
\quad \mathit{bank}\,[\![\, \mathit{till}(10) \,]\!] \\
\mathit{client}(\mathit{init}) \stackrel{\text{df}}{=} \\
\quad \mathtt{go}^{\Delta 5}\ \mathit{travelshop} \rightarrow \mathit{flight}\,?\,(\mathit{standardoffer}) \rightarrow \mathtt{go}^{\Delta 4}\ \mathit{standardoffer} \rightarrow \\
\quad \mathit{info}\,?\,(\mathit{p1}, \mathit{specialoffer}) \rightarrow \mathtt{go}^{\Delta 3}\ \mathit{specialoffer} \rightarrow \mathit{info}\,?\,(\mathit{p2}, \mathit{paying}) \rightarrow \\
\quad \mathtt{go}^{\Delta 6}\ \mathit{paying} \rightarrow \mathit{pay}\,!\,\langle \min\{\mathit{p1}, \mathit{p2}\} \rangle \rightarrow \\
\quad \mathtt{go}^{\Delta 4}\ \mathit{home} \rightarrow \mathit{client}(\mathit{init} - \min\{\mathit{p1}, \mathit{p2}\}) \\
\mathit{agent}(\mathit{balance}) \stackrel{\text{df}}{=} \\
\quad \mathit{flight}\,!\,\langle \mathit{standard} \rangle \rightarrow \mathtt{go}^{\Delta 10}\ \mathit{bank} \rightarrow \mathit{pay}\,?\,(\mathit{profit}) \rightarrow \mathtt{go}^{\Delta 12}\ \mathit{travelshop} \rightarrow \\
\quad \mathit{agent}(\mathit{balance} + \mathit{profit}) \\
\mathit{update}(\mathit{saleprice}) \stackrel{\text{df}}{=} \\
\quad \mathtt{go}^{\Delta 0}\ \mathit{special} \rightarrow \mathit{info}\,!\,\langle \mathit{saleprice} \rangle \rightarrow \mathtt{stop} \\
\mathit{flightinfo}(\mathit{price}, \mathit{next}) \stackrel{\text{df}}{=} \\
\quad \mathit{info}\,!\,\langle \mathit{price}, \mathit{next} \rangle \rightarrow \mathit{flightinfo}(\mathit{price}, \mathit{next}) \\
\mathit{saleinfo}(\mathit{price}, \mathit{next}) \stackrel{\text{df}}{=} \\
\quad \mathit{info}^{\Delta 10}\,?\,(\mathit{newprice})\ \mathtt{then}\ \mathit{saleinfo}(\mathit{newprice}, \mathit{next}) \\
\quad \mathtt{else}\ \mathit{info}\,!\,\langle \mathit{price}, \mathit{next} \rangle \rightarrow \mathit{saleinfo}(\mathit{price}, \mathit{next}) \\
\mathit{till}(\mathit{cash}) \stackrel{\text{df}}{=} \\
\quad \mathit{pay}^{\Delta 1}\,?\,(\mathit{newpayment}) \\
\quad \mathtt{then}\ \mathit{till}(\mathit{cash} + \mathit{newpayment})\ \mathtt{else}\ \mathit{pay}^{\Delta 2}\,!\,\langle \mathit{cash} \rangle\ \mathtt{then}\ \mathit{till}(0)\ \mathtt{else}\ \mathit{till}(\mathit{cash})
\end{array}
$$

Table 4. TiMo description of the running example.

4 Type System

Let X be a set of clocks ranging over $x_1, \ldots, x_n$ and taking values in $\mathbb{R}^{\geq 0}$, where each participant i has assigned its own clock x_i. A clock assignment $v : X \rightarrow \mathbb{R}^{\geq 0}$ returns the time of the clocks in X. We write $v + t$ for the assignment mapping all $x \in X$ to $v(x) + t$. We denote by v_0 the assignment mapping all the clocks to 0; in our approach, when an action is performed by a participant i, the clock x_i is set to the value 0. The set $\Phi(X)$ of clock constraints over X is given by:

$$\varphi ::= \mathit{true} \mid x > c \mid x = c \mid \neg\varphi.$$

An interaction in a global type consists of a send action together with a receive action, each annotated with a clock constraint. The clock constraint specifies when the action could be executed. A difference from the approach described in [4] is that we do not need to use reset predicates to specify which clocks must be reset, because each participant has only one clock assigned to him and the clock is reset after each consumed action.

The syntax for sorts S, timed global types G, and timed local types T is:

$S ::= bool \mid nat \mid \ldots \mid G_c$

$G_c ::= p \to q : \langle S \rangle \{C\}.G_c \mid G_c \mid G_c \mid \mu t.G_c \mid t \mid \mathsf{end} \qquad C ::= \{\varphi_p, \varphi_q\}$

$G_m ::= l \to l' : \langle p \rangle \{M\}.G_m \mid \mu t.G_m \mid t \mid \mathsf{end} \qquad M ::= \{\varphi_p\}$

$T_c ::= !p : \langle S \rangle \{B\}.T_c \mid ?p : \langle S \rangle \{B\}.T_c \mid \mu t.T_c \mid t \mid \mathsf{end} \qquad B ::= \{\varphi_p\}$

The sorts S include base types (bool, nat, etc.), and G_c is for communication session (for the creation of a new session of type G). In G_c, type $p \to q : \langle S \rangle \{C\}.G$ models an interaction: p sends to q a message of sort S; the session then continues as prescribed by G. The step is annotated with a time assertion $C ::= \{\varphi_p, \varphi_q\}$, where φ_p and φ_q are the clock constraints for the output and input actions, respectively. Parallel type $G_c \mid G_c$ puts in parallel the global types of several sessions. Recursive type $\mu t.G$ associates a type variable t to a recursion body G; we assume that type variables are guarded in the standard way and end occurs at least once in G. We denote by $\mathcal{P}(G)$ the set of participants involved in the sessions of the distributed system that are typed by the global type G.

By G_m we model the movement of the participant between the different sessions of the distributed system. The types have a similar explanation as in G_c, except that each one involves only one participant and not two.

In T interactions are modelled from a participant's viewpoint either as sending types $!p : \langle S \rangle \{B\}.T_c$ or receiving types $?p : \langle S \rangle \{B\}.T_c$. We denote the projection of G_c onto $p \in \mathcal{P}(G)$ by $G \downarrow_p$; the definition is standard (see [24]), except that each $\{\varphi_p, \varphi_q\}$ is projected onto the sender (resp. receiver) by keeping only the output/input part φ_p and φ_q, respectively. E.g., if $G_c = p \to q : \langle S \rangle \{\varphi_p, \varphi_q\}.G'_c$, then $G'_c \downarrow_p = !p : \langle S \rangle \{\varphi_p\}.G'_c \downarrow p$ and $G'_c \downarrow_q = ?q : \langle S \rangle \{\varphi_q\}.G'_c \downarrow q$.

Example 1. The specifications of our example from Figure 1 and Table 4 get several distinct global types. Even if usually we use numbers to represent participants, in this case we use the process names just to be easier to follow. Since we have five locations and six participants, we have two global types G_c and G_m, each one formed out of six types.

$G_{home} = G_{flightinfo} = G_{saleinfo} = G_{till} = \mathsf{end}$

$G_{travelshop} = \mu t.agent \to client : \langle string \rangle \{x_{agent} \geq 0, x_{client} \geq 0\}.t$

$G_{standard} = \mu t.$
$\quad flightinfo \to agent : \langle int, string \rangle \{x_{flightinfo} \geq 0, x_{agent} \geq 0\}.t$

$G_{special} = update \to saleinfo : \langle int \rangle .\{x_{update} \geq 0, x_{caleinf} \leq 10\}.$
$\quad \mu t.saleinfo \to client : \langle int, string \rangle \{x_{saleinfo} \geq 0, x_{client} \geq 0\}.t$

$G_{bank} = \mu t.client \to till : \langle int \rangle \{x_{client} \geq 0, x_{till} \leq 1\}.$
$\quad till \to agent : \langle int \rangle \{x_{till} \leq 2, x_{agent} \geq 0\}.t$

$G_{client} = \mu t.home \to travelshop : \langle client \rangle \{x_{client} = 5\}.$
$\quad travelshop \to standard : \langle client \rangle \{x_{client} = 4\}.$
$\quad standard \to special : \langle client \rangle \{x_{client} = 3\}.$
$\quad special \to bank : \langle client \rangle \{x_{client} = 6\}.$
$\quad bank \to home : \langle client \rangle \{x_{client} = 4\}.t$

$G_{agent} = \mu t.travelshop \to bank : \langle agent \rangle \{x_{agent} = 10\}.$
$\quad bank \to travelshop : \langle agent \rangle \{x_{agent} = 12\}.t$

$G_{update} = travelshop \to special : \langle update \rangle \{x_{update} = 0\}$

The fact that $G_{home} = \mathsf{end}$ means that there is no communication performed at location *home*, while $G_{flightinfo} = G_{saleinfo} = G_{till} = \mathsf{end}$ means that the participants $flightinfo$, $saleinfo$ and $till$ are static (they do not perform any migration). The definition of the global type $G_{travelshop}$ means that at location $travelshop$ always only the *agent* should be able to send a message containing a string to the *client*. In a similar manner, the global types $G_{standard}$, $G_{special}$ and G_{bank} describe the expected patterns of interactions inside their corresponding locations. Note that the last three types, namely G_{client}, G_{agent} and G_{update}, represent the desired behaviour of the moving participants as depicted in Figure 1 by arrows.

The typing system uses a map from shared names to either their sorts $(S, S', \ldots)$ or to a special sort $\langle G_c \rangle$ used to type various communication sessions (represented as locations in our approach). Since a type is inferred for each participating process in a certain session, we use the notation $T@l$ (called located type) to represent a local type T assigned to a participating process p in a communication session. As we deal with multiple sessions, the type of each participant is given by the value of its local clock and the local communication types for all locations of the system, and also by its mobility type. Using all these ingredients, we define the following typing system:

$$\Gamma ::= \emptyset \mid \Gamma, \langle G_m \rangle \mid \Gamma, l : \langle G_c \rangle \mid \Gamma, t : \Delta$$
$$\Delta ::= \emptyset \mid \Delta, p : (v, \{T@l\}_{l \in Loc}, G_p).$$

A sorting $(\Gamma, \Gamma', \ldots)$ is a finite map from names to sorts, and from process variables to sequences of types. Typings $(\Delta, \Delta', \ldots)$ record linear usage of session channels and migration capabilities by assigning a family of located types to a vector of session channels. We use the judgement $\Gamma \vdash P \triangleright \Delta$ saying that "under the environment Γ, process P has typing Δ".

We get some results dealing with the type preservation under structural equivalence and operational reduction. According to these results, if a well-typed process takes a reduction step of any kind, the resulting process will be also well-typed. As processes interact, the types need to follow this evolution. This dynamics is formalised by a type reduction relation $\Rightarrow$ on environments Δ.

The next theorem guarantees that if a process is well-typed, than any process congruent with it (by using the structural equivalence $\equiv$) is still well-typed.

Theorem 1. *(subject congruence)* $\Gamma \vdash N \triangleright \Delta$ *and* $N \equiv N'$ *imply* $\Gamma \vdash N' \triangleright \Delta$.

Subject reduction guarantees that typing is preserved by reductions; if a process is well-typed, then the process obtained after a computational step is also well-typed.

Theorem 2. *(subject reduction)* $\Gamma \vdash N \triangleright \Delta$ *and* $N \to N'$ *imply* $\Gamma \vdash N' \triangleright \Delta'$, *where* $\Delta = \Delta'$ *or* $\Delta \Rightarrow \Delta'$.

5 Conclusion and Related Work

A first version of TiMo was proposed by Ciobanu and Koutny in [10] as a simplified version of timed distributed π-calculus [13]. Several variants were developed during the last decade. The same co-authors defined the access permissions given by a type system in PerTiMo [12], while other authors proposed a probabilistic extension pTiMo in [14], as well as a real-time extension rTiMo [1]. Recently, a calculus for structure-aware mobile systems that combines TiMo and the bigraph model was defined in [27]. Inspired by TiMo , a flexible software platform was introduced in [9] to support the specification of distributed systems involving timed migration and safe communication [8]. Interesting properties of distributed systems described by TiMo refer to process migration, time constraints, bounded liveness and optimal reachability [2, 11]. A verification tool called TiMo@PAT [15] was developed by using Process Analysis Toolkit, an extensible platform for various model checkers. A probabilistic temporal logic called PLTM was introduced in [14] to verify complex properties making explicit reference to specific locations, temporal constraints over local clocks and multisets of actions.

A distributed calculus that provides nested multiparty and dynamically joinable sessions, and use intra-session and intra-site communications was proposed in [5]. Processes of the Conversation calculus are used in [6, 26] to treat dynamic join and leave of the participants. The dynamic join and leaving mechanism based on the multiparty session types was extended in [19] by introducing the notion of roles. In order to obtain more expressivity, a nested, higher-order multiparty session types was proposed in [18].

Multiparty session types usually guarantee progress only within a single session [20, 23, 19]. Progress for interleaved sessions was considered for modelling sessions in Java [16, 22]. However, the guarantee of progress can be given only for one single active binary session. In [21, 7] there are constructions of processes providing missing participants in dyadic sessions, which are simpler than the static interaction type system for global progress in dynamically interleaved and interfered multiparty sessions as developed in [17]. In [4], the global times are enriched with time constraints such that the multiparty session types are able to express temporal properties in a way similar to timed automata. In order to express such temporal properties, Scribble was extended with timed constrains in [25]. General conditions of progress and non-zero properties of timed communicating automata at the top of multiparty compatibility were proposed in [3].

Our approach combines these approaches by treating in an unifying framework the time constraints, dynamical join and leave mechanisms and interleaved sessions. To our knowledge, this is the first work trying to accomplish this.

Acknowledgement We enjoy very much to collaborate and be co-authors with Maciej. We send him many thanks and our best wishes.

References

1. B. Aman, G. Ciobanu. Verification of critical systems described in real-time TiMo . *Int'l Journal on Software Tools for Technology Transfer* **19**(4), 395–408 (2017).
2. B. Aman, G. Ciobanu, M. Koutny. Behavioural Equivalences over Migrating Processes with Timers. *Lecture Notes in Computer Science* **7273**, 52–66 (2012).
3. L. Bocchi, J. Lange, N. Yoshida. Meeting Deadlines Together. In *Proceedings CONCUR, LIPIcs* **42**, 283–296 (2015).
4. L. Bocchi, W. Yang, N. Yoshida. Timed Multiparty Session Types. *Lecture Notes in Computer Science* **8704**, 419–434 (2014).
5. R. Bruni, I. Lanese, H. Melgratti, E. Tuosto. Multiparty Sessions in SOC. *Lecture Notes in Computer Science* **5052**, 67–82 (2008).
6. L. Caires, H.T. Vieira. Conversation Types. *Theoretical Computer Science* **411**(51-52), 4399–4440 (2010).
7. M. Carbone, S. Debois. A Graphical Approach to Progress for Structured Communication in Web Services. *Electronic Proceedings in Theoretical Computer Science* **38**, 13–27 (2010).
8. G. Ciobanu. Finding Network Resources by Using Mobile Agents. *Intelligent Distributed Computing IV. Studies in Computational Intelligence* **315**, 305–313 (2010).
9. G. Ciobanu, C. Juravle. Flexible Software Architecture and Language for Mobile Agents. *Concurrency and Computation: Practice and Experience* **24**, 559–571 (2012).
10. G. Ciobanu, M. Koutny. Modelling and Verification of Timed Interaction and Migration. *Lecture Notes in Computer Science* **961**, 215–229 (2008).
11. G. Ciobanu, M. Koutny. Timed Mobility in Process Algebra and Petri Nets. *Journal of Logic and Algebraic Programming* **80**(7), 377–391 (2011).
12. G. Ciobanu, M. Koutny. PerTiMo : A Model of Spatial Migration with Safe Access Permissions. *Computer Journal* **58**(5), 1041–1060 (2015).
13. G. Ciobanu, C. Prisacariu. Timers for Distributed Systems. *Electronic Notes in Theoretic Computer Science* **164**(3), 81–99 (2006).
14. G. Ciobanu, A. Rotaru. A Probabilistic Logic for pTiMo . *Lecture Notes in Computer Science* **8049**, 141–158 (2013).
15. G. Ciobanu, M. Zheng. Automatic Analysis of TiMo Systems in PAT. In *Proceedings 18th ICECCS*, IEEE Computer Society, 121–124 (2013).
16. M. Coppo, M. Dezani-Ciancaglini, N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. *Lecture Notes in Computer Science* **4468**, 1–31 (2007).
17. M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, L. Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science* **26**(2), 238–302 (2015).
18. R. Demangeon, K. Honda. Nested Protocols in Session Types. *Lecture Notes in Computer Science* **7454**, 272–286 (2012).
19. P.-M. Deniélou, N. Yoshida. Dynamic Multirole Session Types. In Proceedings POPL, 435–446 (2011).
20. M. Dezani-Ciancaglini, U. de'Liguoro. Sessions and Session Types: an Overview. *Lecture Notes in Computer Science* **6194**, 1–28 (2010).
21. M. Dezani-Ciancaglini, U. de'Liguoro, N. Yoshida. On Progress for Structured Communications. *Lecture Notes in Computer Science* **4912**, 257–275 (2008).

22. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, S. Drossopoulou. Session Types for Object-Oriented Languages. *Lecture Notes in Computer Science* **4067**, 328–352 (2006).
23. K. Honda, V.T. Vasconcelos, M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. *Lecture Notes in Computer Science* **1381**, 22–138 (1998).
24. K. Honda, N. Yoshida, M. Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM* **63**(1): article 9 (2016).
25. R. Neykova, L. Bocchi, N. Yoshida. Timed Runtime Monitoring for Multiparty Conversations. *Electronic Proceedings in Theoretical Computer Science* **162**, 19–26 (2014).
26. H.T. Vieira, L. Caires, J.C. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. *Lecture Notes in Computer Science* **4960**, 269–283 (2008).
27. W. Xie, H. Zhu, M. Zhang, G. Lu, Y. Fang. Formalization and Verification of Mobile Systems Calculus Using the Rewriting Engine Maude. In *Proceedings 42nd COMPSAC*, IEEE Computer Society, 213–218 (2018).