Timed Migration and Interaction
with Access Permissions

Gabriel Ciobanu\textsuperscript{1} and Maciej Koutny\textsuperscript{2}

\textsuperscript{1} Institute of Computer Science, Romanian Academy
and A.I. Cuza University of Iasi
700483 Iasi, Romania
\texttt{gabriel@iit.tuiasi.ro}

\textsuperscript{2} School of Computing Science
Newcastle University
Newcastle upon Tyne, NE1 7RU, United Kingdom
\texttt{maciej.koutny@newcastle.ac.uk}

Abstract. We introduce and study a process algebra able to model the
systems composed of processes (agents) which may migrate within a dis-
tributed environment comprising a number of distinct locations. Two
processes may communicate if they are present in the same location and,
in addition, they have appropriate access permissions to communicate
over a channel. Access permissions are dynamic, and processes can ac-
quire new access permissions or lose some existing permissions while
migrating from one location to another. Timing constraints coordinate
and control the communication between processes and migration be-
tween locations. Then we completely characterise those situations when
a process is always guaranteed to possess safe access permissions. The
consequences of such a result are twofold. First, we are able to validate
systems where one does not need to check (at least partially) access per-
missions as they are guaranteed not to be violated, improving efficiency
of implementation. Second, one can design systems in which processes
are not blocked (deadlocked) because of the lack of dynamically changing
access permissions.

Keywords: distributed systems, mobile agents, communication, access
permissions, operational semantics, specification, static analysis

1 Introduction

The increasing complexity of mobile applications in which the timing aspects are
important to the system operation means that the need for their effective analysis
and verification is becoming critical. In this paper we explore formal modelling
of mobile systems where one can also specify time-related aspects of migrat-
ing processes and, crucially, security aspects expressed by access permissions
to communication channels. Building on our previous work on Timo presented
at FASE'08 \cite{8}, we introduce Pertimo (Permissions, Timers and Mobility)
which is a process algebra supporting process migration (strong mobility), local
interprocess communication over shared channels controlled by access permissions that processes must possess, and timers (driven by local clocks) controlling the execution of actions. An important feature of the proposed model is that access permissions are dynamic. More precisely, processes can acquire new access permissions, or lose some of their current access permissions while moving from one location to another, modelling a key security related feature. Processes are equipped with input and output capabilities which are active up to pre-defined time deadlines and, if these communications are not taken, alternative continuations for the process behaviour are followed. Another timing constraint allows one to specify the latest time for moving a process from one location to another. These two kinds of timing constraints help in the control and coordination of migration and communication in distributed systems. We provide the syntax and operational semantics of Pertimo which is a discrete time semantics incorporating maximally parallel executions of actions using local clocks.

To introduce the basic components of Pertimo, we use a TravelShop running example in which a client process attempts to pay as little as possible for a ticket to a pre-defined destination. The scenario involves five locations and six processes. The role of each of the locations is as follows: (i) home is a location where the client process starts and ends its journey; (ii) travelshop is a main location of the service which is initially visible to the client; (iii) standard and special are two internal locations of the service where clients can find out about the ticket prices; and (iv) 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 back 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 by the travel shop of the special offer.
- 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.

Pertimo uses timers in order to impose deadlines on the execution of communications and migrations. Moreover, processes need to possess appropriate
access permissions in order to send and receive information. Figure 1 portrays three possible stages of the evolution of the TravelShop system.

Each location has its local clock which determines the timing of actions executed at that location. The timeout of a migration action indicates the network time limit for that action (similar to TTL in TCP/IP).

We use \(\mathbf{x}\) to denote a finite tuple \((x_1, \ldots, x_k)\) whenever it does not lead to a confusion, and if \(X\) is a tuple of sets \((X_1, \ldots, X_k)\) then \(\prod X\) denotes \(X_1 \times \ldots \times X_k\). We assume that the reader is familiar with the basic concepts of process algebras [14]. All proofs our results can be found in [9].
2 Syntax and semantics of \textsc{PerTiMo}

Timing constraints for migration allow one to specify what is the time window for a process to move from one location to another. E.g., a timer (such as \(\Delta 5\)) of a migration action go\[^{\Delta 5}\]\textit{home} indicates that the process will move to \textit{home} within 5 time units. It is also possible to constrain the waiting for a communication on a channel; if a communication action does not happen before a deadline, the waiting process gives up and switches its operation to an alternative. E.g., a timer (such as \(\Delta 4\)) of an output action \(a^{\Delta 4}!()\) makes the channel available for communication only for the period of 4 time units.

We assume suitable data sets including a set \textit{Loc} of locations and a set \textit{Chan} of communication channels. We use a set \(\textit{Id}\) of process identifiers, and each \(id \in \textit{Id}\) has arity \(m_{id}\).

To communicate over a channel at a given network location, the sender process should have a ‘put’ access permission, and the receiving process a ‘get’ access permission. The set \(\Gamma\) of access permissions of a process is a subset of \(\textit{AccPerm} \equiv \{\text{put, get}\} \times \text{Chan} \times \textit{Loc}\). We use the notation \(\text{get}(a@l)\) to denote an access permission \((\text{get}, a, l) \in \textit{AccPerm}\), and \(\text{put}(a@l)\) to denote \((\text{put}, a, l) \in \textit{AccPerm}\). Intuitively, we work with access permissions to sockets where \(l\) represents an IP address and \(a\) represents a communication port.

We allow access permissions of a process to change while moving from one location to another. To model this, we use the following four basic access permission modification operations: \(\text{put}_{a@l}^{+}\) , \(\text{get}_{a@l}^{+}\) , \(\text{put}_{a@l}^{-}\) and \(\text{get}_{a@l}^{-}\), where \(l\) is a location and \(a\) is a communication channel. The first two \((\text{put}_{a@l}^{+} \text{ and } \text{get}_{a@l}^{+})\) add access permissions, while the latter two \((\text{put}_{a@l}^{-} \text{ and } \text{get}_{a@l}^{-})\) remove access permissions. For instance, \(\text{put}_{a@l}^{+}(\Gamma) = \Gamma \cup \{\text{put}(a@l)\}\). Then an access permission modification operation is either the identity on \(\textit{AccPerm}\), or a composition of some basic access permission modification operations such that if \(\text{put}_{a@l}^{+}\) is used in the composition then \(\text{put}_{a@l}^{-}\) is not used (giving and at the same time removing an access permission does not make sense). For a given network, we then specify what are the changes to the access permission sets of processes migrating from one location to another. This is specified as a mapping \(\textit{apm}\) which, for each pair \((l, l')\) of locations, returns a permission modification operation; if a process with the current access permissions \(\Gamma\) moves from location \(l\) to location \(l'\), its new set of access permissions becomes \(\textit{apm}(l, l')(\Gamma)\).

The syntax of \textsc{PerTiMo} is given in Table 1, where \(P\) are processes, \(PP\) processes with (access) permissions, and \(N\) networks. Moreover, for each \(id \in \textit{Id}\), there is a unique process definition of the form:

\[
\text{id}(u_{1}, \ldots, u_{m_{id}} : X_{1}^{id}, \ldots, X_{m_{id}}^{id}) \triangleq P_{id},
\]

where the \(u_{i}\)'s are distinct variables playing the role of parameters, and the \(X_{i}^{id}\)'s are data sets. Processes of the form \(\text{stop}\) and \(\text{id}(v)\) are called \textit{primitive}. In Table 1, it is assumed that:

\begin{itemize}
  \item \(a \in \textit{Chan}\) is a channel, and \(t \in \mathbb{N} \cup \{\infty\}\) is a time deadline;
  \item each \(v_{i}\) is an expression built from values, variables and allowed operations;
\end{itemize}
\[ P ::= a^{\Delta t} ! (v) \text{ then } P \text{ else } P' \] (output)
\[ a^{\Delta t} ? (u : X) \text{ then } P \text{ else } P' \] (input)
\[ \text{go}^{\Delta t} l \text{ then } P \] (move)
\[ P \mid P' \] (parallel)
\[ \text{stop} \] (termination)
\[ \text{id}(v) \] (recursion)
\[ \otimes P \] (stalling)

**Processes**

**Typed Processes**

**Networks**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\otimes$</td>
<td>special symbol used to express that a process is temporarily stalled</td>
</tr>
<tr>
<td>$\text{id}(v)$</td>
<td>the length of $v$ is the same as that of $X$, and the length of $v$ in $\text{id}(v)$ is $m_d$.</td>
</tr>
</tbody>
</table>

- each $u_i$ is a variable, and each $X_i$ is a data set;
- $l$ is a location or a variable, and $\Gamma$ a set of action permissions; and
- $\otimes$ is a special symbol used to express that a process is temporarily stalled.

The only variable binding construct is $a^{\Delta t} ? (u : X)$ which binds the variables $u$ within $P$ (but not within $P'$). We use $fv(P)$ to denote the free variables of a process $P$ (and similarly for processes with access permissions and networks). For a process definition as in (1), we assume that $fv(P_{id}) \subseteq \{u_1, \ldots, u_{m_d}\}$ and so the free variables of $P_{id}$ are parameter bound. Processes are defined up to the alpha-conversion, and $\{v/u, \ldots\}P$ is obtained from $P$ by replacing all free occurrences of a variable $u$ by $v$, possibly after alpha-converting $P$ in order to avoid clashes. Moreover, if $v$ and $u$ are tuples of the same length then $\{v/u\}P = \{v_1/u_1, v_2/u_2, \ldots, v_k/u_k\}P$.

A network $N$ is well-formed if the following hold:

- there are no free variables in $N$;
- there are no occurrences of the special symbol $\otimes$ in $N$; and
- assuming that $\text{id}$ is as in the recursive equation (1), for every $\text{id}(v)$ occurring in $N$ or on the right hand side of any recursive equation, the expression $v_i$ is of type corresponding to $X_i^{id}$ (where we use the standard rules of determining the type of an expression).

Intuitively, a process $a^{\Delta t} ! (v)$ attempt to send a tuple of values $v$ over the channel $a$ for $t$ time units. If successful, it then continues as process $P$ otherwise it continues as the alternative process $P'$. Similarly, $a^{\Delta t} ? (u : X)$ attempt to receive a tuple of values from $X$ and substitute them for the variables $u$. Mobility is implemented by processes $\text{go}^{\Delta t} l$ which moves from the current location...
TravelShop $\overset{\sim}{\equiv}$

home $\{ \text{client}(130) : \emptyset \} \mid$
travelshop $\{ \text{agent}(100) : \{ \text{put}(\text{flight}@\text{travelshop}) \} \mid \text{update}(60) : \emptyset \} \mid$
standard $\{ \text{flightinfo}(110, \text{special}) : \{ \text{put}(\text{info}@\text{standard}), \text{get}(\text{info}@\text{standard}) \} \} \mid$
special $\{ \text{saleinfo}(90, \text{bank}) : \{ \text{put}(\text{info}@\text{special}), \text{get}(\text{info}@\text{special}) \} \} \mid$
bank $\{ \text{till}(10) : \{ \text{put}(\text{pay}@\text{bank}), \text{get}(\text{pay}@\text{bank}) \} \} \mid$
apm(home, travelshop) $\overset{\sim}{\equiv}$ get$_{\text{flight}@\text{travelshop}}$
apm(travelshop, standard) $\overset{\sim}{\equiv}$ get$_{\text{info}@\text{standard}}$
apm(travelshop, special) $\overset{\sim}{\equiv}$ put$_{\text{info}@\text{special}}$
apm(standard, special) $\overset{\sim}{\equiv}$ get$_{\text{info}@\text{special}} \circ$ get$_{\text{info}@\text{standard}}$
apm(special, bank) $\overset{\sim}{\equiv}$ put$_{\text{pay}@\text{bank}} \circ$ get$_{\text{info}@\text{special}} \circ$ get$_{\text{pay}@\text{bank}}$
apm(travelshop, bank) $\overset{\sim}{\equiv}$ get$_{\text{pay}@\text{bank}}$

client(init:eMoney) $\overset{\sim}{\equiv}$
go$^{\Delta_{11}}$ travelshop $\rightarrow$ flight $?\ (\text{standardoffer}:\text{Loc}) \rightarrow$
go$^{\Delta_{11}}$ standardoffer $\rightarrow$ info $\?\ (p1:eMoney, \text{specialoffer}:\text{Loc}) \rightarrow$
go$^{\Delta_{11}}$ specialoffer $\rightarrow$ info $\?\ (p2:eMoney, \text{paying}:\text{Loc}) \rightarrow$
go$^{\Delta_{11}}$ paying $\rightarrow$ pay $\!\{\min\{p1, p2\}\} \rightarrow$
go$^{\Delta_{11}}$ home $\rightarrow$ client(init $-$ min$\{p1, p2\}$)
agent(balance:eMoney) $\overset{\sim}{\equiv}$
flight $!\ (\text{standard}) \rightarrow$ go$^{\Delta_{10}}$ bank $\rightarrow$
pay $\?\ (\text{profit}:\text{eMoney}) \rightarrow$ go$^{\Delta_{10}}$ travelshop $\rightarrow$
agent(balance + profit)
update(saleprice:eMoney) $\overset{\sim}{\equiv}$
go$^{\Delta_{10}}$ special $\rightarrow$ info $\!\{\text{saleprice}\} \rightarrow$ stop
flightinfo(price : eMoney, next : Loc) $\overset{\sim}{\equiv}$
info $\!\{\text{price}, \text{next}\} \rightarrow$ flightinfo(price, next)
saleinfo(price : eMoney, next : Loc) $\overset{\sim}{\equiv}$
info$^{\Delta_{11}}$ ?(newprice:eMoney)
then saleinfo(newprice, next)
else info $\!\{\text{price}, \text{next}\} \rightarrow$ saleinfo(price, next)
till(cash:eMoney) $\overset{\sim}{\equiv}$
pay$^{\Delta_{11}}$ ?(newpayment:eMoney)
then till(cash + newpayment)
else pay$^{\Delta_{11}}$ !(cash) then till(0) else till(cash)

**Table 2.** PERTiMo network modelling the running example together with the relevant access permission modification operations (those omitted are all equal to the identity mapping on AccPerm).

to the location given by $l$ within $t$ time units. Note that since $l$ can be a variable, and so its value is assigned dynamically through communication with other processes, migration actions support a flexible scheme for movement of processes from one location to another.
A network \( l[P : \Gamma] \) specifies a process \( P \) with the access permissions \( \Gamma \) running at a location \( l \). Finally, process expressions of the form \( \circ P \) represent a purely technical device which is used in our formalisation of structural operational semantics of \( \text{PERTMo} \); intuitively, it specifies a process \( P \) which is temporarily \textit{stalled} and so cannot execute any action.

One might wonder why a process can delay migration to another location. The point is that by allowing this we can model in a simple way the non-determinism in the movement of processes which is, in general, outside the control of a system designer. Thus the timer in this case indicates the upper bound on the migration time.

The specification of the running example which captures the essential features of the scenario described in the introduction is given in Table 2. We assume that \( \text{Loc} = \{\text{home, travelshop, standard, special, bank}\} \) and \( \text{Chan} = \{\text{info, flight, pay}\} \). Table 2 shows the process network \( \text{TravelShop} \) modelling the scenario, as well as the access permission modification operations which are applied to the process expressions when they move around the five nodes of the network.

The first component of the operational semantics of \( \text{PERTMo} \) is the structural equivalence \( \equiv \) on networks, similar to that used in [4]. It is the smallest congruence such that the equalities (Eq1-Eq4) in Table 3 hold. Its role is to rearrange a network in order to apply the action rules which are also given in Table 3. Using (Eq1-Eq4) one can always transform a given network \( N \) into a finite parallel composition of networks of the form:

\[
l_1 \parallel [P_1; \Gamma_1] \parallel \ldots \parallel l_n \parallel [P_n; \Gamma_n]\]

such that no process \( P_i \) has the parallel composition operator at its topmost level. Each sub-network \( l_i \parallel [P_i; \Gamma_i] \) is called a \textit{component} of \( N \), the set of all components is denoted by \( \text{comp}(N) \), and the parallel composition (2) is called a \textit{component decomposition} of the network \( N \). Note that these notions are well-defined since component decomposition is unique up to the permutation of the components (see Remark 1 below).

Table 3 introduces two kinds of action rules, \( N \xrightarrow{\lambda} N' \) and \( N \xrightarrow{\phi} N' \). The former is an execution of an action \( \lambda \), and the latter a time step in location \( l \).

In the rule (\text{TIME}), \( N \not\xrightarrow{\lambda} \) means that no \( l\)-action \( \lambda \) (i.e., an action of the form \( id@l \) or \( lb \@l \) or \( @l \) or \( a(v)@l \)) can be applied to \( N \). Moreover, \( \phi(N) \) is obtained by taking the component decomposition of \( N \) and simultaneously replacing all components of the form \( l \parallel [a^m \omega \text{ then } P \text{ else } Q : \Gamma] \) — where \( \omega \) stands for \( !\langle v \rangle \) or \( ?(w.X) \) — by \( l \parallel [Q : \Gamma] \) if \( t = 0 \), and otherwise by \( l \parallel [a^{m-1} \omega \text{ then } P \text{ else } Q : \Gamma] \). After that occurrences of the special symbol \( \circ \) in \( N \) are erased.

So far we defined located executions of actions. An entire computational step is captured by a derivation \( N \xrightarrow{\Delta} N' \), where \( \Delta = \{\lambda_1, \ldots, \lambda_n\} \) is a finite multiset of \( l\)-actions for some location \( l \) such that

\[
N \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} \xrightarrow{\phi} N'.
\]
(Eq1) \[ N | N' \equiv N' | N \]
(Eq2) \[ (N | N') | N'' \equiv N | (N' | N'') \]
(Eq3) \[ l[ PP | PP'] \equiv l[ PP ] | l[ PP' ] \]
(Eq4) \[ l[ P | Q : \Gamma ] \equiv l[ P : \Gamma | Q : \Gamma ] \]
(Call) \[ l[ id(v) : \Gamma ] \xrightarrow{id@l} l[ \oplus (v/u)P_d : \Gamma ] \]
(Move) \[ l[ go^{\Delta t} l' \text{ then } P : \Gamma ] \xrightarrow{\text{int}} l'[ \oplus P : \text{apm}(l, l')(\Gamma) ] \]
(Wait) \[ l[ go^{\Delta t} l' \text{ then } P : \Gamma ] \xrightarrow{\text{go}} l[ \oplus go^{\Delta t-1} l' \text{ then } P : \Gamma ] \]

| \[ put(a@l) \in \Gamma \quad get(a@l) \in \Gamma' \quad v \in \llbracket X \rrbracket | \]
| \[ l[ a^{\Delta t} ! (v) \text{ then } P \text{ else } Q : \Gamma' \mid a^{\Delta t'} ? (u) : \Gamma' \text{ then } P' \text{ else } Q' : \Gamma' ] \]

| \[ \rightarrow \quad l[ \oplus P : \Gamma \mid \oplus (v/u)P': \Gamma' ] \]

| \[ N \xrightarrow{\lambda_{\text{comp}}} N' \]
| \[ \text{then} \]
| \[ N | N'' \xrightarrow{\lambda} N' | N'' \]

| \[ N' \equiv N' \]
| \[ N' \xrightarrow{\lambda} N'' \]
| \[ N'' \equiv N''' \]

| \[ \text{then} \]
| \[ N \xrightarrow{\lambda_{\phi_i}} \phi_i(N) \]

Table 3. Four rules of the structural equivalence (Eq1-Eq4), and seven action rules (Call Move Wait Com Par Equiv Time) of the operational semantics.

We also call \( N' \) directly reachable from \( N \). In other words, we can capture the cumulative effect of the concurrent execution of the multiset of actions \( \Lambda \) at location \( l \). Intuitively, networks evolution conforms to the locally maximally parallelism paradigm since one executes in a single location \( l \) as many as possible concurrent action before applying a local time move which signifies the passage of a unit of time at location \( l \).

The two results below ensure that derivations are well defined. First, one cannot execute an unbound ed sequence of action moves without time progress.

**Proposition 1.** If \( N \) is a network and \( N \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} N' \), then \( k \leq |\text{comp}(N)| \).

Second, if we start with a well-formed network, execution proceeds through alternating executions of time steps and contiguous sequences of local actions making up what can be regarded as a maximally concurrent step (note the role of the special symbols \( \oplus \)). This intuition is reinforced by the following result.
Proposition 2. Let $N$ be a well-formed network. If $N \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} N'$, then we have $N \xrightarrow{i_1} \cdots \xrightarrow{i_n} N'$, for every permutation $i_1, \ldots, i_n$ of $1, \ldots, n$.

It is worth noting that the semantical treatment of PERTM0 — itself a continuation of the idea developed for T|M0 — goes beyond interleaving semantics by introducing an explicit representation of local maximal parallelism and local time progress in the network evolution.

Our last result in this section is that the rules of Table 3 preserve well-formedness of networks.

Proposition 3. Networks reachable from a well-formed network are well-formed.

Table 4 illustrates execution steps based on the scenario illustrated in Figure 1 (note that $A_2$ represents a parallel execution of two actions). We indicate only the main rules used in the derivation of steps. Each execution step takes a single unit of time in the location at which it has been executed and some timers are decremented by one (e.g., the timer $\Delta ?$ of channel $\text{info}$ in $U_0$ is changed to $\Delta ?$ in $U_1$). Other timers which have expired cause an immediate migration or the selection of the alternative part of a communication action (see $W_1$ which is replaced by $W_2$).

Note that the last network expression derived from TravelShop in Table 4 corresponds to the intermediate configuration shown in Figure 1 (b). Note also that in the representation of Figure 1 (b) we show the home location, even though it is not present in the last network expression in Table 4. The reason is that the client process has moved to travelshop, and there is at present no process residing at home. This situation changes in the final configuration of Figure 1 (c) where client has completed its migration and came back to its initial location.

Remark 1. Component decomposition is unique since the rule (CALL) treats recursive definitions as function calls which take a unit of time. Another consequence of such a treatment is that it is impossible to execute an infinite sequence of action steps without executing any time steps. Both these properties would not hold if, instead of an action rule (CALL), we would have a structural rule of the form $l \{ \text{id}(u) : \Gamma \} \equiv l \{ \{v/u\}P_{id} : \Gamma \}$. □

3 Safe Access Permissions

In this section, we attempt to verify that a migrating process possesses a sufficiently rich set of initial access permissions such that whenever later on it attempts to communicate over a channel, it has the required safe access permission. While doing so, we need to take into account that migrating processes have their access permission sets modified according to the mapping $apm$. If we succeed, then an important security problem related to migration and access permissions is solved in the sense that never an unauthorised attempt to communicate over a channel happens during network evolutions.
TravelShop

\[
\begin{array}{c}
A_1, A_2, A_3, A_4, A_5 \\
6 \times \text{(CALL)}
\end{array}
\]

\[
\begin{array}{c}
\text{home} \Rightarrow [go^{A_5} \text{travelshop} \rightarrow P_0 : \emptyset] \\
\text{travelshop} \Rightarrow [Q_0 : \{put(\text{flight}@\text{travelshop})\}] \Rightarrow [go^{A_0} \text{special} \rightarrow R_0 : \emptyset] \\
\text{standard} \Rightarrow [U_0 : \{put(\text{info}@\text{standard}), get(\text{info}@\text{standard})\}] \\
\text{special} \Rightarrow [V_0 : \{put(\text{info}@\text{special}), get(\text{info}@\text{special})\}] \\
\text{bank} \Rightarrow [W_0 : \{put(\text{pay}@\text{bank}), get(\text{pay}@\text{bank})\}]
\end{array}
\]

\[
\begin{array}{c}
(\text{home } \lor \text{travelshop}) \Rightarrow (\text{travelshop } \lor \text{special}) \\
2 \times \text{(MOVE)}
\end{array}
\]

\[
\begin{array}{c}
\text{travelshop} \Rightarrow [flight ? (\text{standardoffer}:\text{Loc}) \rightarrow P_1 : \{get(\text{flight}@\text{travelshop})\}] \\
\text{standard} \Rightarrow [U_1 : \{put(\text{info}@\text{standard}), get(\text{info}@\text{standard})\}] \\
\text{special} \Rightarrow [V_1 : \{put(\text{info}@\text{special}), get(\text{info}@\text{special})\}] \\
\text{bank} \Rightarrow [W_1 : \{put(\text{pay}@\text{bank}), get(\text{pay}@\text{bank})\}]
\end{array}
\]

\[
\begin{array}{c}
\{\text{flight}(\text{standard})@\text{travelshop}\} \Rightarrow \{\text{info}(60)@\text{special}\} \\
2 \times \text{(COM)}
\end{array}
\]

\[
\begin{array}{c}
P_0 = \text{flight } ? (\text{standardoffer}:\text{Loc}) \rightarrow P_1 \\
P_1 = go^{A_1} \text{standardoffer} \rightarrow \text{info } ? (p1:e\text{Money}, \text{specialoffer}:\text{Loc}) \rightarrow go^{A_2} \text{specialoffer} \rightarrow \text{info } ? (p2:e\text{Money}, \text{paying}:\text{Loc}) \rightarrow go^{A_3} \text{paying} \rightarrow \text{pay } ! (\min\{p1, p2\}) \rightarrow go^{A_4} \text{home} \rightarrow \text{client}(100 - \min\{p1, p2\}) \\
P_2 = \{\text{standard}/\text{standardoffer}\} P_1 \\
Q_0 = \text{flight } ? (\text{standard}) \rightarrow Q_1 \\
Q_1 = go^{A_{10}} \text{bank} \rightarrow pay ? (\text{profit}:e\text{Money}) \rightarrow go^{A_{12}} \text{travelshop} \rightarrow \text{agent}(100 + \text{profit}) \\
R_0 = \text{info} ! (60) \rightarrow \text{stop} \\
U_0 = \text{info}^{A_3} ! (110, \text{special}) \rightarrow \text{flightinfo}(110, \text{special}) \\
U_1 = \text{info}^{A_5} ! (110, \text{special}) \rightarrow \text{flightinfo}(110, \text{special}) \\
U_2 = \text{flightinfo}(110, \text{special}) \\
V_0 = \text{info}^{A_{10}} ? (\text{newprice}:e\text{Money}) \text{ then saleinfo(newprice, bank)} \\
\quad \text{else info} ! (90, \text{bank}) \rightarrow \text{saleinfo}(90, \text{bank}) \\
V_1 = \text{info}^{A_9} ? (\text{newprice}:e\text{Money}) \text{ then saleinfo(newprice, bank)} \\
\quad \text{else info} ! (90, \text{bank}) \rightarrow \text{saleinfo}(90, \text{bank}) \\
V_2 = \text{saleinfo}(60, \text{bank}) \\
W_0 = \text{pay}^{A_2} ? (\text{newpayment}:e\text{Money}) \text{ then till}(10 + \text{newpayment}) \\
\quad \text{else pay}^{A_2} !(10) \text{ then till}(0) \text{ else till}(10) \\
W_1 = \text{pay}^{A_0} ? (\text{newpayment}:e\text{Money}) \text{ then till}(10 + \text{newpayment}) \\
\quad \text{else pay}^{A_2} !(10) \text{ then till}(0) \text{ else till}(10) \\
W_2 = \text{pay}^{A_2} !(10) \text{ then till}(0) \text{ else till}(10)
\end{array}
\]

Table 4. Execution steps for the running example where \(A_1 = \{\text{client}@\text{home}\}\), \(A_2 = \{\text{agent}@\text{travelshop}, \text{update}@\text{travelshop}\}\), \(A_3 = \{\text{flightinfo}@\text{standard}\}\), \(A_4 = \{\text{saleinfo}@\text{special}\}\) and \(A_5 = \{\text{till}@\text{bank}\}\).
Throughout this section we assume that all the data sets are finite (see Remark 2), and that the r.h.s. \( P_{id} \) of each recursive definition (1) is either a primitive process (i.e., it is of the form \( P_{id} = \text{stop} \) or \( P_{id} = \text{id}(w) \)) or \( P_{id} \) uses exactly one application of one of the process operators to some primitive processes. This does not diminish the generality of the proposed method since we can always transform all recursive definition into the simple form using additional process identifiers and recursive definitions without affecting the results that follow (e.g., \( P \overset{def}{=} a \rightarrow b \rightarrow P \) is replaced by \( P \overset{def}{=} a \rightarrow P' \) and \( P' \overset{def}{=} b \rightarrow P \)).

We use judgements of the form \( \Gamma \vdash l \mid P \) to mean that a single-component network \( l \mid P : \Gamma \) has safe access permissions. We assume the open system context which means that we cannot know precisely the migration patterns of a process and its communication channels which can be acquired through interaction with (unknown) processes. We plan to deal with close systems in future, and then take into account the time aspects (here we use time for process coordination).

Given a set of locations \( \text{Loc} \) together with the \( \text{apm} \) mapping as well a process \( P \) and location \( l \), we want to devise rules for checking that a set of access permissions \( \Gamma \) satisfies \( \Gamma \vdash l \mid P \). For instance, if \( P = \text{go} \triangleleft l' \) then \( a \rightarrow \text{stop} \) and \( \text{apm}(l, l') = \text{put}_{a\triangleleft l'} \) then there is no \( \Gamma \) such that \( \Gamma \vdash l \mid P \).

If \( P \) does not involve recursive definitions, the task is straightforward. One just needs to follow the syntactic structure of the process and incrementally derive \( \Gamma \). Dealing with recursion is more complicated, and the solution we propose consists in unfolding a recursive process expression sufficiently many times to cover all possibilities resulting from migration. For all \( id \in \text{Id} \), \( n \geq 0 \) and \( v \in \prod X^{id} \), the \( n \)-th unfolding of \( \text{id}(v) \) is given by \( \text{id}(v)^n \overset{def}{=} \text{stop} \) and, for \( n > 0 \), \( \text{id}(v)^n \overset{def}{=} P \) where \( P \) is obtained from \( \{v/u\} P_{id} \) by replacing each subexpression of the form \( id'(w) \) with \( id'(w)^{n-1} \).

The derivation rules for \( \Gamma \vdash l \mid P \) are given in Table 5. The (TMove) rule concerns a migration from location \( l \) to \( l' \). In order to have \( l \mid \text{go} \triangleleft l' \) then \( P : \Gamma \) with safe access permissions, it is necessary to have \( l' \mid P : \Gamma' \) with safe access permissions after applying the access permission modification to \( \Gamma \) when moving from \( l \) to \( l' \) (note that \( \Gamma' = \text{apm}(l, l')(\Gamma) \)). The rule (TOut) simply requires that a process attempting to send a message along a channel \( a \) should possess the access permission \( \text{put}(a \triangleleft l) \). Similarly, the rule (TIn) requires that a process attempting to receive a message along a channel \( a \) should possess the access permission \( \text{get}(a \triangleleft l) \); moreover, after receiving this message it has to have safe access permissions with the current \( \Gamma \) irrespective of the values carried by that message. The constant \( H \) in the rule (TRec) is \( H \overset{def}{=} 2 \cdot |\text{Loc}| \cdot (1 + \sum_{id \in \text{Id}} |X^{id}|) \cdots |X^{id}_{n,a}| \). The value of \( H \) comes from rather technical considerations needed to prove results. We can always ensure that \( H \) is a well-defined integer and (TIn) is a finitary rule according to the following argument.

Remark 2. The judgement system in Table 5 makes important use of data through the (TOut) rule as a received message may carry a location or channel name which may later be used by other rules. Other kinds of values carried by messages or present in process descriptions are ignored. Hence, for the purpose of
safe access permissions, we can replace all non-location and non-channel values by a special value $\tau$, and all the data types different from $\text{Loc}$ and $\text{Chan}$ by a singleton type $X = \{ \tau \}$. In this way, all the data sets become finite. Hence, in particular, $H$ is an integer value, and $\prod X$ in (TIN) is a finite set. \hfill \square

We have defined what it means to have safe access permissions in the case of a single-component network. In the general case, a network $N$ has safe access permissions if each of its components does. These two definitions are consistent in the sense that $\Gamma \vdash_{\iota} P$ iff $\Gamma \vdash_{\iota} P_t$, for every component $l \llbracket P_t : \Gamma \rrbracket$ of a single-component network $l \llbracket P : \Gamma \rrbracket$, which follows from the rule (TPAR).

The first main result states that safe access permissions is preserved over the network evolutions defined by the operational semantics.

**Theorem 1 (soundness).** If a well-formed network $N$ has safe access permissions, and $N'$ is reachable from $N$, then $N'$ has also safe access permissions.

The second main result is that in a network with safe access permissions there are no attempts to access a communication channel without an appropriate access permission. This result should be seen as a justification of our interest in the notion of safe access permissions.

**Theorem 2 (safety of communications).** Let $N$ be a well-formed network with safe access permissions.

\begin{align*}
& l \llbracket a^{\Delta t}! (v) \rrbracket \text{ then } P \text{ else } Q : \Gamma \rrbracket \in \text{comp}(N) \quad \text{implies} \quad \text{put}(a@l) \in \Gamma \\
& l \llbracket a^{\Delta t}? (u:X) \rrbracket \text{ then } P \text{ else } Q : \Gamma \rrbracket \in \text{comp}(N) \quad \text{implies} \quad \text{get}(a@l) \in \Gamma.
\end{align*}
As an immediate corollary of Theorem 2, for a network with safe access permissions it is possible to simplify the operational rule for process communication, by deleting $\text{put}(a@l) \in \Gamma$ and $\text{get}(a@l) \in \Gamma'$ in rule ($\text{Com}$), and so simplifying the implementation.

The third main result is that the notion of a network with safe access permissions is complete in the sense that a network which does not satisfy this property can always be placed in an environment which reveals its potential to break safety of interprocess communication.

**Theorem 3 (completeness).** Let $N = I \mid P : \Gamma \mid$ be a well-formed network such that $\Gamma \not\models I P$. Then there is a well-formed network $N'$ with safe access permissions as well as a well-formed network $N''$ reachable from $N \mid N'$ such that one of the following holds.

- There is a component $I'' \mid a^{\Delta t} ! (v)$ then $P'' \models I'' : \Gamma''$ of $N''$ such that $\text{put}(a@l') \notin \Gamma'$.
- There is a component $I'' \mid a^{\Delta t} ? (u:X)$ then $P'' \models I'' : \Gamma''$ of $N''$ such that $\text{get}(a@l') \notin \Gamma'$.

We developed a sound and complete system for safe of communication and migration in open networks. Hence we are able to validate systems where one does not need to check access permissions as they are guaranteed not to be violated, improving implementation. Moreover, the results can be extended allow designing systems in which processes are not blocked (deadlocked) because of the lack of dynamically changing access permissions.

### 4 Conclusions and Related Work

We introduced a distributed process algebra with processes able to migrate between different locations and timing constraints used to control migration and communication. We use local clocks and local maximal parallelism of actions. Processes have appropriate access rights to communicate; the access permissions are dynamic and can change. We have provided an operational semantics of this model, and investigated the safety of communication and migration in terms of access permissions. While we are not aware of any approach combining all these aspects regarding mobility with timing constraints, local clocks, and dynamic access permission mechanism, our work is related to a large body of literature using process algebra in (type-based) security. Several systems encompass various forms of access control policies in distributed systems. Among them, the work on the $\text{Dpi}$ calculus in [13] uses type systems to control statically the access to the resources at the different locations of a distributed system. Other related work on access control in distributed systems is done in the context of the language $\text{K} \text{LaIm}$ and its extensions, using type systems that enable the dynamic exchange of access rights. The paper [7] combines a weak form of information flow control with typed cryptographic operations to ensure safe static access control and secure network communications. The paper [5] uses cryptographic operations
and capability types to get a secure implementation of a typed $\pi$-calculus in order to realise various policies for accessing the communication channels. None of these systems, however, uses together mobility as a first class citizen controlled by timing constraints, dynamic aspects of the access permissions, local clocks and parallelism. These advantages of the new model can allow to specify and enforce more diverse and expressive security policies based on access permissions. This could be done in the context of designing good programming language supporting migration in a distributed environment [16]. On the other hand, several prototype languages have been designed and experimental implementations derived from process calculi like Klam [4] and Acute [15]. These prototype languages did not become a practical programming language because hard questions revolving mainly around issues relating to security. PERTiMo is intended to help bridging the gap between the existing foundational process algebras and forthcoming realistic languages. It extends some previous attempts related to TDP [10] and TiMo [8]. PERTiMo derives from TiMo model (a simplified distributed $\pi$-calculus with explicit timeouts) presented in [8] by adding a type system in order to express security aspects related to access permissions. The basic notion of a timeout in TiMo seemed useful and elegant. PERTiMo retains this notion and, in addition, it incorporates access permissions in order to provide formal foundations for security problems relating to the adequate protection of access control information in distributed environment.

As related work, we should mention distributed $\pi$-calculus having an explicit notion of location, and dealing with static resources access [12] by using a type system. The paper [3] studies a $\pi$-calculus extension with a timer construct, and then enriches the timed $\pi_t$ with locations. Other timed extensions of process algebras have been studied in [2, 11]. In [6] the authors present a typed $\pi$-calculus with groups and group creation in which each name belongs to a group. The rules for good environments ensure that the names and groups declared in an environment are distinct, and that all the types mentioned in an environment are good. A consequence of the typing discipline is the ability to preserve secrets, namely preventing certain communications that would leak secrets. The type system is used for regulating the mobile computation, allowing to partition the processes into disjoint groups in order to specify the behaviour of both communication and mobility. Somehow related to our dynamic access permissions, [11] presents a parametric calculus for processes exchanging code which may contain free variables to be bound by the receiver’s code (called open mobile code). Type safety is ensured by a combination of static and dynamic checks of such an exchange of open code. In this way it is possible to express rebinding of code in a distributed environment in a relatively simple way.

Deriving concrete implementation from PERTiMo is part of future work, and the approach presented in this paper is just a first step in this direction. In our future work we plan to extend the current model as follows:

- access permissions to locations to control migrations of processes;
- security levels for migrating processes to control access permissions to channels and locations;
relaxing the synchronisation resulting from the maximally parallel semantics, by retaining maximal parallelism within each location, but allowing locations to proceed with different speed;
- rules for well-typing of values in exchanged messages;
- defining and analysing security policies for access and migration control; and
- introducing and analysing failures in process migration.

**Acknowledgement** We would like to thank the anonymous reviewers for their constructive suggestions. This research was supported by the International Joint Project of the Royal Society of London, the EPSRC VERDAD project, and NSFC Grants 60910004 and 2010CB328102.

**References**