Timed Migration and Interaction with Access Permissions

Gabriel Ciobanu and Maciej Koutny

TECHNICAL REPORT SERIES

No. CS-TR-1291 November 2011

TECHNICAL REPORT SERIES

No. CS-TR-1291 November, 2011

Timed Migration and Interaction with Access Permissions

G. Ciobanu, M. Koutny

Abstract

We introduce and study a process algebra able to model the systems composed of processes (agents) which may migrate within a distributed 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 acquire 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 between 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 permissions 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.

Bibliographical details

CIOBANU, G., KOUTNY, M.

Timed Migration and Interaction with Access Permissions
[By] G. Ciobanu, M. Koutny
Newcastle upon Tyne: Newcastle University: Computing Science, 2011.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1291)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1291

Abstract

We introduce and study a process algebra able to model the systems composed of processes (agents) which may migrate within a distributed 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 acquire 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 between 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 permissions 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.

About the authors

Gabriel Ciobanu is a senior researcher at the Institute of Computer Science, Romanian Academy. He is also a Professor at "Alexandru Ioan Cuza" University of Iasi, Romania.

Maciej Koutny obtained his MSc (1982) and PhD (1984) from the Warsaw University of Technology. In 1985 he joined the then Computing Laboratory of the University of Newcastle upon Tyne to work as a Research Associate. In 1986 he became a Lecturer in Computing Science at Newcastle, and in 1994 was promoted to an established Readership at Newcastle. In 2000 he became a Professor of Computing Science.

Suggested keywords

DISTRIBUTED SYSTEMS
MOBILE AGENTS
COMMUNICATION
ACCESS PERMISSIONS
OPERATIONAL SEMANTICS
SPECIFICATION
STATIC ANALYSIS

Timed Migration and Interaction with Access Permissions

Gabriel Ciobanu[1] and Maciej Koutny[2]

[1] Institute of Computer Science, Romanian Academy
and A.I.Cuza University of Iasi
700483 Iasi, Romania
gabriel@iit.tuiasi.ro

[2] School of Computing Science
Newcastle University
Newcastle upon Tyne, NE1 7RU, United Kingdom
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 distributed 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 acquire 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 between 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 permissions 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 migrating processes and, crucially, security aspects expressed by access permissions to communication channels. Building on our previous work on TiMo presented at FASE'08 [8], we introduce PerTiMo (**Per**missions, **Ti**mers and **Mo**bility) 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

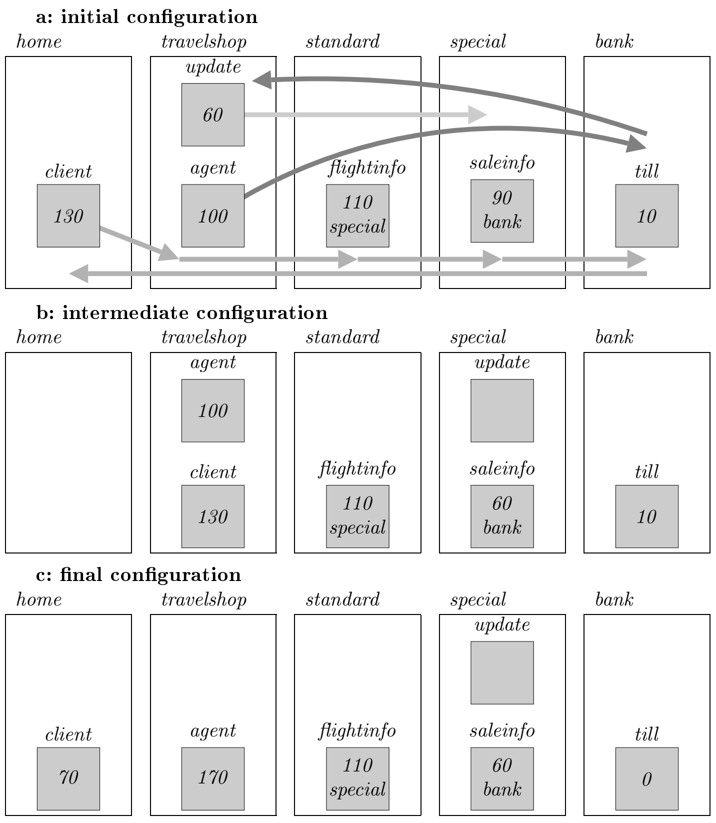

Fig. 1. Three snapshots of the evolution of the running example. In the initial configuration we indicated the intended migration paths of three processes. The intermediate configuration illustrates the phase of the evolution after some initial movements of the client and after updating the second flight price. The final configuration shows the state of the system after a successful payment has been made; the total sum of e-money owned by the client (*70*), agent (*170*) and till (*0*) is exactly the same as the sum at the beginning of the evolution when the client has *130*, agent *100* and till *10*. Note that the channels used by processes to communicate information are not shown.

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 $\boldsymbol{x}$ to denote a finite tuple $(x_1, \ldots, x_k)$ whenever it does not lead to a confusion, and if $\boldsymbol{X}$ is a tuple of sets $(X_1, \ldots, X_k)$ then $\prod \boldsymbol{X}$ 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 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 $\mathtt{go}^{\Delta 5}\,home$ indicates that the process will move to *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 $a^{\Delta 4}\,\mathbf{!}\,\langle 13\rangle$ makes the channel available for communication only for the period of 4 time units. We assume suitable data sets including a set *Loc* of *locations* and a set *Chan* of *communication channels*. We use a set *Id* of process identifiers, and each $id \in Id$ 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 Γ of access permissions of a process is a subset of $AccPerm \overset{\text{df}}{=} \{put, get\} \times Chan \times Loc$. We use the notation $get\langle a@l\rangle$ to denote an access permission $(get, a, l) \in AccPerm$, and $put\langle a@l\rangle$ to denote $(put, a, l) \in AccPerm$. 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: $put^{+}_{a@l}$, $get^{+}_{a@l}$, $put^{-}_{a@l}$ and $get^{-}_{a@l}$, where l is a location and a is a communication channel. The first two ($put^{+}_{a@l}$ and $get^{+}_{a@l}$) add access permissions, while the latter two ($put^{-}_{a@l}$ and $get^{-}_{a@l}$) remove access permissions. For instance, $put^{+}_{a@l}(\Gamma) = \Gamma \cup \{put\langle a@l\rangle\}$. Then an *access permission modification* operation is either the identity on *AccPerm*, or a composition of some basic access permission modification operations such that if $put^{+}_{a@l}$ is used in the composition then $put^{-}_{a@l}$ 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 *apm* which, for each pair (l, l') of locations, returns a permission modification operation; if a process with the current access permissions Γ moves from location l to location l', its new set of access permissions becomes $apm(l, l')(\Gamma)$.

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

$$id(u_1, \dots, u_{m_{id}} : X^{id}_1, \dots, X^{id}_{m_{id}}) \overset{\text{df}}{=} P_{id} \ , \tag{1}$$

where the u_i's are distinct variables playing the role of parameters, and the X^{id}_i's are data sets. Processes of the form $\mathtt{stop}$ and $id(\boldsymbol{v})$ are called *primitive*. In Table 1, it is assumed that:

- $a \in Chan$ is a channel, and $t \in \mathbb{N} \cup \{\infty\}$ is a time deadline;
- each v_i is an expression built from values, variables and allowed operations;

Processes	$P ::= a^{\Delta t}\,!\,\langle \boldsymbol{v}\rangle$ `then` P `else` P' $\mid$	(output)
	$a^{\Delta t}\,?\,(\boldsymbol{u}{:}\boldsymbol{X})$ `then` P `else` P' $\mid$	(input)
	$\texttt{go}^{\Delta t}\ l$ `then` P $\mid$	(move)
	$P \mid P'$ $\mid$	(parallel)
	`stop` $\mid$	(termination)
	$id(\boldsymbol{v})$ $\mid$	(recursion)
	ⓈP	(stalling)
Typed Processes	$PP ::= P : \Gamma \mid PP \mid PP'$	
Networks	$N ::= l[\![PP]\!] \mid N \mid N'$	

Shorthand notation:

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

Table 1. PerTiMo syntax. The length of $\boldsymbol{u}$ is the same as that of $\boldsymbol{X}$, and the length of $\boldsymbol{v}$ in $id(\boldsymbol{v})$ is m_{id}.

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

The only variable binding construct is $a^{\Delta t}\,?\,(\boldsymbol{u}{:}\boldsymbol{X})$ `then` P `else` P' which binds the variables $\boldsymbol{u}$ 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 $fv(P_{id}) \subseteq \{u_1, \ldots, u_{m_{id}}\}$ 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 $\boldsymbol{v}$ and $\boldsymbol{u}$ are tuples of the same length then $\{\boldsymbol{v}/\boldsymbol{u}\}P = \{v_1/u_1, v_2/u_2, \ldots, v_k/u_k\}P$.

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 Ⓢ in N; and
- assuming that id is as in the recursive equation (1), for every $id(\boldsymbol{v})$ 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 $a^{\Delta t}\,!\,\langle \boldsymbol{v}\rangle$ `then` P `else` P' attempts to send a tuple of values $\boldsymbol{v}$ 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, $a^{\Delta t}\,?\,(\boldsymbol{u}{:}\boldsymbol{X})$ `then` P `else` P' is a process that attempts for t time units to input a tuple of values from $\boldsymbol{X}$ and substitute them for the variables $\boldsymbol{u}$. Mobility is implemented by processes $\texttt{go}^{\Delta t}\ l$ `then` P which moves from the current location

$$
\begin{array}{l}
\mathit{TravelShop} \stackrel{\mathrm{df}}{=} \\
\quad \mathit{home} \,[\![\, \mathit{client}(\mathit{130}) : \varnothing \,]\!] \;\mid \\
\quad \mathit{travelshop} \,[\![\, \mathit{agent}(\mathit{100}) : \{\mathit{put}\langle \mathit{flight}@\mathit{travelshop}\rangle\} \;\mid\; \mathit{update}(\mathit{60}) : \varnothing \,]\!] \;\mid \\
\quad \mathit{standard} \,[\![\, \mathit{flightinfo}(\mathit{110}, \mathit{special}) : \{\mathit{put}\langle \mathit{info}@\mathit{standard}\rangle, \mathit{get}\langle \mathit{info}@\mathit{standard}\rangle\} \,]\!] \;\mid \\
\quad \mathit{special} \,[\![\, \mathit{saleinfo}(\mathit{90}, \mathit{bank}) : \{\mathit{put}\langle \mathit{info}@\mathit{special}\rangle, \mathit{get}\langle \mathit{info}@\mathit{special}\rangle\} \,]\!] \;\mid \\
\quad \mathit{bank} \,[\![\, \mathit{till}(\mathit{10}) : \{\mathit{put}\langle \mathit{pay}@\mathit{bank}\rangle, \mathit{get}\langle \mathit{pay}@\mathit{bank}\rangle\} \,]\!]
\end{array}
$$

$$
\begin{array}{ll}
\mathit{apm}(\mathit{home}, \mathit{travelshop}) & \stackrel{\mathrm{df}}{=} \mathit{get}^{+}_{\mathit{flight}@\mathit{travelshop}} \\
\mathit{apm}(\mathit{travelshop}, \mathit{standard}) & \stackrel{\mathrm{df}}{=} \mathit{get}^{+}_{\mathit{info}@\mathit{standard}} \\
\mathit{apm}(\mathit{travelshop}, \mathit{special}) & \stackrel{\mathrm{df}}{=} \mathit{put}^{+}_{\mathit{info}@\mathit{special}} \\
\mathit{apm}(\mathit{standard}, \mathit{special}) & \stackrel{\mathrm{df}}{=} \mathit{get}^{+}_{\mathit{info}@\mathit{special}} \circ \mathit{get}^{-}_{\mathit{info}@\mathit{standard}} \\
\mathit{apm}(\mathit{special}, \mathit{bank}) & \stackrel{\mathrm{df}}{=} \mathit{put}^{+}_{\mathit{pay}@\mathit{bank}} \circ \mathit{get}^{-}_{\mathit{info}@\mathit{special}} \circ \mathit{get}^{-}_{\mathit{pay}@\mathit{bank}} \\
\mathit{apm}(\mathit{travelshop}, \mathit{bank}) & \stackrel{\mathrm{df}}{=} \mathit{get}^{+}_{\mathit{pay}@\mathit{bank}}
\end{array}
$$

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

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 Γ running at a location l. Finally, process expressions of the form ⓈP represent a purely technical device which is used in our formalisation of structural operational semantics of PerTiMo; intuitively, it specifies a process P which is temporarily *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 $Loc = \{home, travelshop, standard, special, bank\}$ and $Chan = \{info, flight, pay\}$. Table 2 shows the process network *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 PerTiMo 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:

$$l_1\,[\![\,P_1{:}\Gamma_1\,]\!] \mid \ldots \mid l_n\,[\![\,P_n{:}\Gamma_n\,]\!] \qquad (2)$$

such that no process P_i has the parallel composition operator at its topmost level. Each sub-network $l_i\,[\![\,P_i{:}\Gamma_i\,]\!]$ is called a *component* of N, the set of all components is denoted by $comp(N)$, and the parallel composition (2) is called a *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 $N \xrightarrow{\surd_l} N'$. The former is an execution of an action λ, and the latter a time step in location l. In the rule (Time), $N \not\to_l$ means that no l-action λ (i.e., an action of the form $id@l$ or $l \triangleright l'$ or $@l$ or $a\langle \boldsymbol{v}\rangle@l$) can be applied to N. Moreover, $\phi_l(N)$ is obtained by taking the component decomposition of N and simultaneously replacing all components of the form $l\,[\![\,a^{\Delta t}\omega$ $\mathtt{then}$ P $\mathtt{else}$ $Q : \Gamma\,]\!]$ — where ω stands for $\boldsymbol{!}\,\langle \boldsymbol{v}\rangle$ or $\boldsymbol{?}\,(\boldsymbol{u}{:}\boldsymbol{X})$ — by $l\,[\![\,Q : \Gamma\,]\!]$ if $t = 0$, and otherwise by $l\,[\![\,a^{\Delta t-1}\omega$ $\mathtt{then}$ P $\mathtt{else}$ $Q : \Gamma\,]\!]$. After that occurrences of the special symbol Ⓢ in N are erased.

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

$$N \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} \xrightarrow{\surd_l} N' \,.$$

(Eq1)	$N \mid N' \;\equiv\; N' \mid N$
(Eq2)	$(N \mid N') \mid N'' \;\equiv\; N \mid (N' \mid N'')$
(Eq3)	$l\,[\![\, PP \mid PP' \,]\!] \;\equiv\; l\,[\![\, PP \,]\!] \mid l\,[\![\, PP' \,]\!]$
(Eq4)	$l\,[\![\, P \mid Q : \Gamma \,]\!] \;\equiv\; l\,[\![\, P : \Gamma \mid Q : \Gamma \,]\!]$
(Call)	$l\,[\![\, id(\boldsymbol{v}) : \Gamma \,]\!] \xrightarrow{id@l} l\,[\![\, \circledS\, \{\boldsymbol{v}/\boldsymbol{u}\}P_{id} : \Gamma \,]\!]$
(Move)	$l\,[\![\, \mathtt{go}^{\Delta t}\ l'\ \mathtt{then}\ P : \Gamma \,]\!] \xrightarrow{l \rhd l'} l'\,[\![\, \circledS\, P : apm(l,l')(\Gamma) \,]\!]$
(Wait)	$\dfrac{t > 0}{l\,[\![\, \mathtt{go}^{\Delta t}\ l'\ \mathtt{then}\ P : \Gamma \,]\!] \xrightarrow{@l} l\,[\![\, \circledS\, \mathtt{go}^{\Delta t-1}\ l'\ \mathtt{then}\ P : \Gamma \,]\!]}$
(Com)	$\dfrac{put\langle a@l\rangle \in \Gamma \qquad get\langle a@l\rangle \in \Gamma' \qquad \boldsymbol{v} \in \prod \boldsymbol{X}}{\begin{array}{c} l\,[\![\, a^{\Delta t}\,\mathtt{!}\,\langle\boldsymbol{v}\rangle\ \mathtt{then}\ P\ \mathtt{else}\ Q : \Gamma \;\mid\; a^{\Delta t'}\,\mathtt{?}\,(\boldsymbol{u}{:}\boldsymbol{X})\ \mathtt{then}\ P'\ \mathtt{else}\ Q' : \Gamma' \,]\!] \\ \xrightarrow{a\langle\boldsymbol{v}\rangle@l} \quad l\,[\![\, \circledS\, P : \Gamma \;\mid\; \circledS\, \{\boldsymbol{v}/\boldsymbol{u}\}P' : \Gamma' \,]\!] \end{array}}$
(Par)	$\dfrac{N \xrightarrow{\lambda} N'}{N \mid N'' \xrightarrow{\lambda} N' \mid N''}$
(Equiv)	$\dfrac{N \equiv N' \qquad N' \xrightarrow{\lambda} N'' \qquad N'' \equiv N'''}{N \xrightarrow{\lambda} N'''}$
(Time)	$\dfrac{N \not\longrightarrow_{l}}{N \xrightarrow{\surd_{l}} \phi_{l}(N)}$

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 Λ 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 unbounded 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 $k \leq |comp(N)|$.*

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 $\circledS$). 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 $N \xrightarrow{\lambda_{i_1}} \cdots \xrightarrow{\lambda_{i_n}} N'$, for every permutation $i_1, \ldots, i_n$ of $1, \ldots, n$.*

It is worth noting that the semantical treatment of PerTiMo — itself a continuation of the idea developed for TiMo — 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 Λ_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 $\Delta 3$ of channel *info* in U_0 is changed to $\Delta 2$ 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 $l [\![id(\boldsymbol{v}) : \Gamma]\!] \equiv l [\![\{\boldsymbol{v}/\boldsymbol{u}\} P_{id} : \Gamma]\!]$. □

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.

$$\mathit{TravelShop}$$

$$\xRightarrow{\Lambda_1\ \ \Lambda_2\ \ \Lambda_3\ \ \Lambda_4\ \ \Lambda_5} \qquad 6\ \times\ (\textsc{Call})$$

$$\begin{array}{l}
\mathit{home}\,[\![\, \mathsf{go}^{\Delta 5}\ \mathit{travelshop} \to P_0 : \varnothing \,]\!]\ \mid \\
\mathit{travelshop}\,[\![\, Q_0 : \{\mathit{put}\langle \mathit{flight@travelshop}\rangle\}\ \mid\ \mathsf{go}^{\Delta 0}\ \mathit{special} \to R_0 : \varnothing \,]\!]\ \mid \\
\mathit{standard}\,[\![\, U_0 : \{\mathit{put}\langle \mathit{info@standard}\rangle, \mathit{get}\langle \mathit{info@standard}\rangle\} \,]\!]\ \mid \\
\mathit{special}\,[\![\, V_0 : \{\mathit{put}\langle \mathit{info@special}\rangle, \mathit{get}\langle \mathit{info@special}\rangle\} \,]\!]\ \mid \\
\mathit{bank}\,[\![\, W_0 : \{\mathit{put}\langle \mathit{pay@bank}\rangle, \mathit{get}\langle \mathit{pay@bank}\rangle\} \,]\!]
\end{array}$$

$$\xRightarrow{\{\mathit{home} \rhd \mathit{travelshop}\}\ \ \{\mathit{travelshop} \rhd \mathit{special}\}} \qquad 2\ \times\ (\textsc{Move})$$

$$\begin{array}{l}
\mathit{travelshop}\,[\![\, \mathit{flight}\,?\,(\mathit{standardoffer}{:}\mathit{Loc}) \ \to P_1{:}\{\mathit{get}\langle \mathit{flight@travelshop}\rangle\}\ \mid \\
\qquad\qquad \mathit{flight}\,!\,\langle \mathit{standard}\rangle \ \to Q_1{:}\{\mathit{put}\langle \mathit{flight@travelshop}\rangle\} \,]\!]\ \mid \\
\mathit{standard}\,[\![\, U_1 : \{\mathit{put}\langle \mathit{info@standard}\rangle, \mathit{get}\langle \mathit{info@standard}\rangle\} \,]\!]\ \mid \\
\mathit{special}\,[\![\, \mathit{info}^{\Delta 9}\,?\,(\mathit{newprice} : \mathit{eMoney}) \\
\qquad\qquad \to V_1 : \{\mathit{put}\langle \mathit{info@special}\rangle, \mathit{get}\langle \mathit{info@special}\rangle\}\ \mid \\
\qquad\quad \mathit{info}\,!\,\langle 60\rangle \ \to \mathtt{stop} : \{\mathit{put}\langle \mathit{info@special}\rangle\} \,]\!]\ \mid \\
\mathit{bank}\,[\![\, W_1 : \{\mathit{put}\langle \mathit{pay@bank}\rangle, \mathit{get}\langle \mathit{pay@bank}\rangle\} \,]\!]
\end{array}$$

$$\xRightarrow{\{\mathit{flight}\langle \mathit{standard}\rangle @\mathit{travelshop}\}\ \ \{\mathit{info}\langle 60\rangle @\mathit{special}\}} \qquad 2\ \times\ (\textsc{Com})$$

$$\begin{array}{l}
\mathit{travelshop}\,[\![\, P_2{:}\{\mathit{get}\langle \mathit{flight@travelshop}\rangle\}\ \mid\ Q_1{:}\{\mathit{put}\langle \mathit{flight@travelshop}\rangle\} \,]\!]\ \mid \\
\mathit{standard}\,[\![\, U_2 : \{\mathit{put}\langle \mathit{info@standard}\rangle, \mathit{get}\langle \mathit{info@standard}\rangle\} \,]\!]\ \mid \\
\mathit{special}\,[\![\, V_2 : \{\mathit{put}\langle \mathit{info@special}\rangle, \mathit{get}\langle \mathit{info@special}\rangle\}\ \mid\ \mathtt{stop} : \{\mathit{put}\langle \mathit{info@special}\rangle\} \,]\!]\ \mid \\
\mathit{bank}\,[\![\, W_2 : \{\mathit{put}\langle \mathit{pay@bank}\rangle, \mathit{get}\langle \mathit{pay@bank}\rangle\} \,]\!]
\end{array}$$

$$\begin{array}{lll}
P_0 & = & \mathit{flight}\,?\,(\mathit{standardoffer}{:}\mathit{Loc}) \ \to P_1 \\
P_1 & = & \mathsf{go}^{\Delta 4}\ \mathit{standardoffer} \to \mathit{info}\,?\,(\mathit{p1}{:}\mathit{eMoney}, \mathit{specialoffer}{:}\mathit{Loc}) \ \to \\
 & & \mathsf{go}^{\Delta 3}\ \mathit{specialoffer} \to \mathit{info}\,?\,(\mathit{p2}{:}\mathit{eMoney}, \mathit{paying}{:}\mathit{Loc}) \ \to \\
 & & \mathsf{go}^{\Delta 6}\ \mathit{paying} \to \mathit{pay}\,!\,\langle \min\{\mathit{p1}, \mathit{p2}\}\rangle \ \to \\
 & & \mathsf{go}^{\Delta 4}\ \mathit{home} \to \mathit{client}(\mathit{130} - \min\{\mathit{p1}, \mathit{p2}\}) \\
P_2 & = & \{\mathit{standard}/\mathit{standardoffer}\}P_1 \\
Q_0 & = & \mathit{flight}\,!\,\langle \mathit{standard}\rangle \ \to Q_1 \\
Q_1 & = & \mathsf{go}^{\Delta 10}\ \mathit{bank} \to \\
 & & \mathit{pay}\,?\,(\mathit{profit}{:}\mathit{eMoney}) \ \to \mathsf{go}^{\Delta 12}\ \mathit{travelshop} \to \mathit{agent}(\mathit{100} + \mathit{profit}) \\
R_0 & = & \mathit{info}\,!\,\langle 60\rangle \ \to \mathtt{stop} \\
U_0 & = & \mathit{info}^{\Delta 3}\,!\,\langle 110, \mathit{special}\rangle \ \to \mathit{flightinfo}(110, \mathit{special}) \\
U_1 & = & \mathit{info}^{\Delta 2}\,!\,\langle 110, \mathit{special}\rangle \ \to \mathit{flightinfo}(110, \mathit{special}) \\
U_2 & = & \mathit{flightinfo}(110, \mathit{special}) \\
V_0 & = & \mathit{info}^{\Delta 10}\,?\,(\mathit{newprice}{:}\mathit{eMoney})\ \mathtt{then}\ \mathit{saleinfo}(\mathit{newprice}, \mathit{bank}) \\
 & & \qquad\qquad\qquad\qquad\quad \mathtt{else}\ \mathit{info}\,!\,\langle 90, \mathit{bank}\rangle \ \to \mathit{saleinfo}(90, \mathit{bank}) \\
V_1 & = & \mathit{info}^{\Delta 9}\,?\,(\mathit{newprice}{:}\mathit{eMoney})\ \mathtt{then}\ \mathit{saleinfo}(\mathit{newprice}, \mathit{bank}) \\
 & & \qquad\qquad\qquad\qquad\quad \mathtt{else}\ \mathit{info}\,!\,\langle 90, \mathit{bank}\rangle \ \to \mathit{saleinfo}(90, \mathit{bank}) \\
V_2 & = & \mathit{saleinfo}(60, \mathit{bank}) \\
W_0 & = & \mathit{pay}^{\Delta 1}\,?\,(\mathit{newpayment}{:}\mathit{eMoney})\ \mathtt{then}\ \mathit{till}(\mathit{10} + \mathit{newpayment}) \\
 & & \qquad\qquad\qquad\qquad\quad \mathtt{else}\ \mathit{pay}^{\Delta 2}\,!\,\langle 10\rangle\ \mathtt{then}\ \mathit{till}(0)\ \mathtt{else}\ \mathit{till}(10) \\
W_1 & = & \mathit{pay}^{\Delta 0}\,?\,(\mathit{newpayment}{:}\mathit{eMoney})\ \mathtt{then}\ \mathit{till}(\mathit{10} + \mathit{newpayment}) \\
 & & \qquad\qquad\qquad\qquad\quad \mathtt{else}\ \mathit{pay}^{\Delta 2}\,!\,\langle 10\rangle\ \mathtt{then}\ \mathit{till}(0)\ \mathtt{else}\ \mathit{till}(10) \\
W_2 & = & \mathit{pay}^{\Delta 2}\,!\,\langle 10\rangle\ \mathtt{then}\ \mathit{till}(0)\ \mathtt{else}\ \mathit{till}(10)
\end{array}$$

Table 4. Execution steps for the running example where $\Lambda_1 = \{\mathit{client@home}\}$, $\Lambda_2 = \{\mathit{agent@travelshop}, \mathit{update@travelshop}\}$, $\Lambda_3 = \{\mathit{flightinfo@standard}\}$, $\Lambda_4 = \{\mathit{saleinfo@special}\}$ and $\Lambda_5 = \{\mathit{till@bank}\}$.

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 $P_{id} = \texttt{stop}$ or $P_{id} = id'(\boldsymbol{w})$) or P_{id} uses exactly one application of one of the process operators to some primitive process(es). 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., $P \overset{\mathrm{df}}{=} a \to b \to P$ is replaced by $P \overset{\mathrm{df}}{=} a \to P'$ and $P' \overset{\mathrm{df}}{=} b \to P$).

We use judgements of the form $\Gamma \vdash_l P$ to mean that a single-component network $l \llbracket P{:}\Gamma \rrbracket$ 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 *Loc* together with the *apm* mapping as well a process P and location l, we want to devise rules for checking that a set of access permissions Γ satisfies $\Gamma \vdash_l P$. For instance, if $P = \texttt{go}^{\Delta 0}\ l'\ \texttt{then}\ a^{\Delta 1}\,\texttt{!}\,\langle 1\rangle \to \texttt{stop}$ and $apm(l,l') = put^{-}_{a@l'}$ then there is no Γ such that $\Gamma \vdash_l P$.

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 Γ. 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 $id \in Id$, $n \geq 0$ and $\boldsymbol{v} \in \prod \boldsymbol{X}^{id}$, the n-th *unfolding* of $id(\boldsymbol{v})$ is given by $id(\boldsymbol{v})^{\uparrow 0} \overset{\mathrm{df}}{=} \texttt{stop}$ and, for $n > 0$, $id(\boldsymbol{v})^{\uparrow n} \overset{\mathrm{df}}{=} P$ where P is obtained from $\{\boldsymbol{v}/\boldsymbol{u}\}P_{id}$ by replacing each subexpression of the form $id'(\boldsymbol{w})$ with $id'(\boldsymbol{w})^{\uparrow n-1}$.

The derivation rules for $\Gamma \vdash_l P$ are given in Table 5. The (TMove) rule concerns a migration from location l to l'. In order to have $l \llbracket \texttt{go}^{\Delta t}\ l'\ \texttt{then}\ P : \Gamma \rrbracket$ with safe access permissions, it is necessary to have $l' \llbracket P : \Gamma' \rrbracket$ with safe access permissions after applying the access permission modification to Γ when moving from l to l' (note that $\Gamma' = apm(l,l')(\Gamma)$). The rule (TOut) simply requires that a process attempting to send a message along a channel a should possess the access permission $put\langle a@l\rangle$. Similarly, the rule (TIn) requires that a process attempting to receive a message along a channel a should possess the access permission $get\langle a@l\rangle$; moreover, after receiving this message it has to have safe access permissions with the current Γ irrespective of the values carried by that message. The constant H in the rule (TRec) is $H \overset{\mathrm{df}}{=} 2 \cdot |Loc| \cdot \big(1 + \sum_{id \in Id} |X_1^{id}| \cdot \ldots \cdot |X_{m_{id}}^{id}|\big)$. 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

$$(\mathrm{TS{\scriptsize UB}}) \qquad \frac{\Gamma' \subseteq \Gamma \quad \Gamma' \vdash_l P}{\Gamma \vdash_l P}$$

$$(\mathrm{TS{\scriptsize TOP}}) \qquad \varnothing \vdash_l \mathtt{stop}$$

$$(\mathrm{TM{\scriptsize OVE}}) \qquad \frac{apm(l,l')(\Gamma) \vdash_{l'} P}{\Gamma \vdash_l \mathtt{go}^{\Delta t}\ l' \ \mathtt{then}\ P}$$

$$(\mathrm{TO{\scriptsize UT}}) \qquad \frac{put\langle a@l\rangle \in \Gamma \quad \Gamma \vdash_l P \quad \Gamma \vdash_l Q}{\Gamma \vdash_l a^{\Delta t}\,\mathbf{!}\,\langle \boldsymbol{v}\rangle\ \mathtt{then}\ P\ \mathtt{else}\ Q}$$

$$(\mathrm{TI{\scriptsize N}}) \qquad \frac{get\langle a@l\rangle \in \Gamma \qquad \forall \boldsymbol{v} \in \prod \boldsymbol{X} : \Gamma \vdash_l \{\boldsymbol{v}/\boldsymbol{u}\}P \qquad \Gamma \vdash_l Q}{\Gamma \vdash_l a^{\Delta t}\,\mathbf{?}\,(\boldsymbol{u}{:}\boldsymbol{X})\ \mathtt{then}\ P\ \mathtt{else}\ Q}$$

$$(\mathrm{TR{\scriptsize EC}}) \qquad \frac{\Gamma \vdash_l id(\boldsymbol{v})^{\uparrow H}}{\Gamma \vdash_l id(\boldsymbol{v})}$$

$$(\mathrm{TP{\scriptsize AR}}) \qquad \frac{\Gamma \vdash_l P \quad \Gamma \vdash_l Q}{\Gamma \cup \Gamma \vdash_l P \,|\, Q}$$

Table 5. Derivation rules for processes with safe access permissions.

safe access permissions, we can replace all non-location and non-channel values by a special value τ, and all the data types different from *Loc* and *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 $\prod \boldsymbol{X}$ in (TIn) is a finite set. □

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 $\Gamma \vdash_l P$ iff $\Gamma \vdash_l P_i$, for every component network $l\,[\![\, P_i{:}\Gamma \,]\!]$ of a single-component network $l\,[\![\, P{:}\Gamma \,]\!]$, 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.*

$$\begin{array}{lll} l\,[\![\, a^{\Delta t}\,\mathbf{!}\,\langle \boldsymbol{v}\rangle\ \mathtt{then}\ P\ \mathtt{else}\ Q : \Gamma \,]\!] \in comp(N) & \textit{implies} & put\langle a@l\rangle \in \Gamma \\ l\,[\![\, a^{\Delta t}\,\mathbf{?}\,(\boldsymbol{u}{:}\boldsymbol{X})\ \mathtt{then}\ P\ \mathtt{else}\ Q : \Gamma \,]\!] \in comp(N) & \textit{implies} & get\langle a@l\rangle \in \Gamma\ . \end{array}$$

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 $put\langle a@l\rangle \in \Gamma$ and $get\langle a@l\rangle \in \Gamma'$ in rule (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 $N = l\,[\![\, P : \Gamma \,]\!]$ be a well-formed network such that $\Gamma \not\vdash_l P$. 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 $l'\,[\![\, a^{\Delta t}\,!\,\langle \boldsymbol{v}\rangle$ `then` P' `else` $P'' : \Gamma' \,]\!]$ of N'' such that $put\langle a@l'\rangle \notin \Gamma'$.*
- *There is a component $l'\,[\![\, a^{\Delta t}\,?\,(\boldsymbol{u}{:}\boldsymbol{X})$ `then` P' `else` $P'' : \Gamma' \,]\!]$ of N'' such that $get\langle a@l'\rangle \notin \Gamma'$.*

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 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 Klaim 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] use 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 KLAIM [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 TDPI [10] and TIMO [8]. PERTIMO derives from TIMO model (a simplified distributed π-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 π-calculus extension with a timer construct, and then enriches the timed π_t with locations. Other timed extensions of process algebras have been studied in [2, 11]. In [6] the authors present a typed π-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, [1] 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.

References

1. D.Ancona, S.Fagorzi and E.Zucca: A Parametric Calculus for Mobile Open Code. *ENTCS* 192 (2008) 3–22.
2. J.Baeten and J.A.Bergstra: Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time. *Fundamenta Informaticae* 29 (1997) 51–76.
3. M.Berger: *Towards Abstractions For Distributed Systems.* Imperial College, Department of Computing (2002).
4. L.Bettini et al.: The KLAIM Project: Theory and Practice. Proc. of *Global Computing: Programming Environments, Languages, Security and Analysis of Systems*, Springer, LNCS 2874 (2003) 88–150.
5. M.Bugliesi and M.Giunti: Secure Implementations of Typed Channel Abstractions. Proc. of *POPL*, ACM (2007) 251–262.
6. L.Cardelli, G.Ghelli and A.Gordon: Secrecy and Group Creation. *Inf. Comput.* 196 (2005) 127–155.
7. T.Chothia, D.Duggan and J.Vitek: Type-based Distributed Access Control. Proc. of *CSFW'03*, IEEE Computer Society (2003) 170–184.
8. G.Ciobanu and M.Koutny: Modelling and Verification of Timed Interaction and Migration. Proc. of *FASE'08*, Springer, LNCS 4961 (2008) 215–229.
9. G.Ciobanu and M.Koutny: TiMoTy: Timed Mobility with Types. Technical Report of Formal Methods Laboratory, Romanian Academy, Institute of Computer Science, Iasi (2010).
10. G.Ciobanu and C.Prisacariu: Timers for Distributed Systems. *ENTCS* 164 (2006) 81–99.
11. F.Corradini, G.L.Ferrari and M.Pistore: On the Semantics of Durational Actions. *Theoretical Computer Science* 269 (2001) 47–82.
12. M.Hennessy: *A Distributed π-calculus.* Cambridge University Press (2007).
13. M.Hennessy and J.Riely: Resource Access Control in Systems of Mobile Agents. *Information and Computation* 173 (2002) 82–120.
14. R.Milner: *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press (1999).
15. P.Sewell et al.: ACUTE: High-Level Programming Language Design for Distributed Computation. *Journal of Functional Programming* 17 (2007) 547–612.
16. T.Thorn: Programming Languages for Mobile Code. *ACM Computing Surveys* 29 (1997) 213–239.