PerTiMo: A Model of Spatial Migration with Safe Access Permissions

Gabriel Ciobanu and Maciej Koutny
PerTiMo: A Model of Spatial Migration with Safe Access Permissions

G. Ciobanu, M. Koutny

Abstract

We introduce a process algebra with processes able to migrate between different explicit locations of a distributed environment defined by a number of spatially distinct locations. We use timing constraints to control migration and communication, and local clocks and local maximal parallelism of actions. Two processes may communicate if they are present in the same location and, in addition, they have appropriate access permissions (put or get) to communicate over a shared channel. Access permissions can be acquired or removed while moving from one location to another. Timing constraints coordinate and control both the communication between processes and migration between locations. We completely characterize those situations when a process is always guaranteed to possess safe access permissions. In this way one can design systems in which processes are not blocked (deadlocked) due to the lack of dynamically changing access permissions.
Bibliographical details

CIOBANU, G., KOUTNY, M.
PerTiMo: A Model of Spatial Migration with Safe Access Permissions
[By] G. Ciobanu, M. Koutny
(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1303)

Added entries

NEWCASTLE UNIVERSITY

Abstract

We introduce a process algebra with processes able to migrate between different explicit locations of a distributed environment defined by a number of spatially distinct locations. We use timing constraints to control migration and communication, and local clocks and local maximal parallelism of actions. Two processes may communicate if they are present in the same location and, in addition, they have appropriate access permissions (put or get) to communicate over a shared channel. Access permissions can be acquired or removed while moving from one location to another. Timing constraints coordinate and control both the communication between processes and migration between locations. We completely characterize those situations when a process is always guaranteed to possess safe access permissions. In this way one can design systems in which processes are not blocked (deadlocked) due to 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
SECURITY
COMMUNICATION
MIGRATION
SAFE ACCESS PERMISSION
OPERATIONAL SEMANTICS
SOUNDNESS
COMPLETENESS
PERTiMo: A Model of Spatial Migration with Safe Access Permissions

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

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

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

Abstract. We introduce a process algebra with processes able to migrate between different explicit locations of a distributed environment defined by a number of spatially distinct locations. We use timing constraints to control migration and communication, and local clocks and local maximal parallelism of actions. Two processes may communicate if they are present in the same location and, in addition, they have appropriate access permissions (put or get) to communicate over a shared channel. Access permissions can be acquired or removed while moving from one location to another. Timing constraints coordinate and control both the communication between processes and migration between locations. We completely characterize those situations when a process is always guaranteed to possess safe access permissions. In this way one can design systems in which processes are not blocked (deadlocked) due to the lack of dynamically changing access permissions.

Keywords: distributed systems, security, communication, migration, safe access permission, operational semantics, soundness, completeness.

1 Introduction

The increasing complexity of mobile applications in which the timing aspects are important to the systems means that the need for a sound theoretical approach and their effective analysis and verification is becoming critical. In this paper we explore formal modelling of mobile systems where one can specify time-related aspects of migrating processes in distributed systems with local clocks and maximal parallelism of actions. Processes have appropriate access rights to communicate; the access permissions are dynamic and can change. We provide an operational semantics of this model, and investigate the crucial security aspects expressed by access permissions to communication channels. Building on our previous work presented at FASE’08 [10] and FM’11 [11], we describe
**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 an important security 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:

- **home** is a location where the client process starts and ends its journey;
- **travelshop** is a main location of the service which is initially visible to the client;
- **standard** and **special** are two internal locations of the service where clients can find out about the ticket prices; and
- **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. Table 1 provides a schematic depiction of three possible stages of the evolution of the TravelShop system.

This paper is a revised and extended version of the conference paper [11], and is structured in the following way. We first describe the syntax and semantics of PERTiMo. After that we introduce a scheme aimed at identifying situations when a process enters a state in which it cannot communicate with other processes due to not having sufficient access permissions. We then show that the developed scheme is sound and complete for ensuring safety of communication for networks of migrating processes. In this way we address our main goal which is to treat the serious problem of access permission in a formal way.

Throughout the paper we use the standard mathematical notation. In particular, we use $x$ 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 we use $\prod X$ to denote the Cartesian product $X_1 \times \ldots \times X_k$. We assume that the reader is familiar with the basic concepts of process algebras [17].

2 PERTiMo

We start by describing the syntax and semantics of PERTiMo which uses timing constraints allowing, for example, to specify what is the time window for a process to move from one location to another. For example, a timer (such as $\Delta 5$) of a migration action go$^{\Delta 5}$home indicates that the process will move to the location home within 5 local time units. It is also possible to wait 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 mode. For example, a timer (such as $\Delta 4$) of an output action $a^{\Delta 4} \langle \{ \rangle$ makes the channel available for local communication only for the period of 4 time units.

2.1 Syntax

In order to encode the desired features of PERTiMo, we assume a set of finite types which includes Loc (a set of locations) and Chan (a set of communication channels). We will also use variables and a finite set of process identifiers, $Id$, such that each $id \in Id$ has the 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. In general, the set of access permissions $\Gamma$ of a process is a subset of the overall set of access permissions:

$$\text{AccPerm} \triangleq \{ \text{put, get} \} \times \text{Chan} \times \text{Loc}.$$
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.
We use the notation $\text{get}(a@l)$ to denote an access permission $(\text{get}, a, l) \in \text{AccPerm}$ and $\text{put}(a@l)$ to denote $(\text{put}, a, l) \in \text{AccPerm}$. Intuitively, we work with access permissions to sockets where $l$ represents an IP address and $a$ represents a communication port.

Inspired primarily by the security issues of network migration, 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:

$$\text{put}^+_a@l, \quad \text{get}^+_a@l, \quad \text{put}^-_a@l, \quad \text{get}^-_a@l$$

where $l$ is a location and $a$ is a communication channel. The first two ($\text{put}^+_a@l$ and $\text{get}^+_a@l$) add access permissions, while the latter two ($\text{put}^-_a@l$ 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 $\text{AccPerm}$, 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 instance,

$$\text{get}^+_a@l \circ \text{put}^-_{b@l'}(\Gamma) = \Gamma \cup \{\text{get}(a@l)\} \setminus \{\text{put}(b@l')\}.$$  

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 $\text{apm}$ which, for each pair of locations, returns a permission modification operation. Hence, if a process with the current access permissions $\Gamma$ moves from location $l$ to location $l'$, its new set of access permissions becomes $\text{apm}(l,l')(\Gamma)$.

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:

$$id(u_1, \ldots, u_m : X_1^{id}, \ldots, X_m^{id}) \triangleq P_{id},$$  

where the $u_i$'s are distinct variables playing the role of parameters, and the $X_i^{id}$'s are types. Processes of the form $\text{stop}$ and $id(v)$ are called primitive.

In Table 1, it is assumed that:

- $a \in \text{Chan}$ 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;
- each $u_i$ is a variable and $X_i$ a type;
- $l$ is a location or a variable, and $\Gamma$ a set of action permissions; and
- $\circ$ is a special symbol used to express that a process is temporarily stalled.

The only variable binding construct is $a^{\Delta t} ? (u : X)$ then $P$ else $P'$ which binds the variables $u$ within $P$ (but not within $P'$); we use $\text{fv}(P)$ to denote the free
Processes

\[ P ::= a^{\Delta t}! (v) \text{ then } P \text{ else } P' \quad (\text{output}) \]
\[ a^{\Delta t} ? (u:X) \text{ then } P \text{ else } P' \quad (\text{input}) \]
\[ \text{go}^{\Delta t} l \text{ then } P \quad (\text{move}) \]
\[ P | P' \quad (\text{parallel}) \]
\[ \text{stop} \quad (\text{termination}) \]
\[ \text{id}(v) \quad (\text{recursion}) \]
\[ \otimes P \quad (\text{stalling}) \]

Processes with access Permissions

\[ PP ::= P : \Gamma \mid PP | PP' \]

Networks

\[ N ::= [PP] | N | N' \]

**Short hand notation:**

- \( a! (v) \rightarrow P \) will be used to denote \( a^{\Delta \infty}! (v) \text{ then } P \text{ else stop} \)
- \( a ? (u:X) \rightarrow P \) will be used to denote \( a^{\Delta \infty} ? (u:X) \text{ then } P \text{ else stop} \).

**Table 1.** \( \text{PERThMo} \) syntax. The length of \( u \) is the same as that of \( X \) and the length of \( v \) in \( id(v) \) is \( m_{id} \).

Variables of a process \( P \) (and similarly for typed processes and networks). For a process definition as in (1), we assume that

\[ f_v(P_{id}) \subseteq \{ u_1, \ldots, u_{m_{id}} \} \]

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 name capture. 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 \( id \) is as in recursive equation (1), for every \( id(v) \) occurring in \( N \) or on the right hand side of any recursive equation, the expression \( v_i \) is of type \( X_i^{id} \) (where we use the standard rules of determining the type of an expression by taking into account the types associated with the binding variables \( u_i \) in input constructs and recursive definitions).

A process \( a^{\Delta t}! (v) \text{ then } P \text{ else } P' \) attempts to send a tuple of values \( v \) over the channel \( a \) for \( t \) local time units. If successful, it then continues as process \( P \), and otherwise it continues as the alternative process \( P' \). Similarly, \( a^{\Delta t} ? (u:X) \text{ then } P \text{ else } P' \) is a process that attempts for \( t \) time units to input a tuple of values of types given by \( X \) and substitute them for the variables \( u \). Mobility is implemented by processes like \( \text{go}^{\Delta t} l \text{ then } P \) which moves from the
TravelShop \(\triangleq\)
\[\begin{align*}
home \{ \text{client} (130) : \emptyset \} \mid \\
\text{travelshop} \{ \text{agent} (100) : \{ \text{put} \langle \text{flight} @ \text{travelshop} \rangle \} \mid \text{update} (60) : \emptyset \} \mid \\
\text{standard} \{ \text{flightinfo} (110, \text{special}) : \{ \text{put} \langle \text{info} @ \text{standard} \rangle, \text{get} \langle \text{info} @ \text{standard} \rangle \} \} \mid \\
\text{special} \{ \text{saleinfo} (90, \text{bank}) : \{ \text{put} \langle \text{info} @ \text{special} \rangle, \text{get} \langle \text{info} @ \text{special} \rangle \} \} \mid \\
\text{bank} \{ \text{till} (10) : \{ \text{put} \langle \text{pay} @ \text{bank} \rangle, \text{get} \langle \text{pay} @ \text{bank} \rangle \} \} \\
\end{align*}\]
\[\begin{align*}
apm \langle \text{home}, \text{travelshop} \rangle & \stackrel{\triangleq}{=} \text{get}^+ \langle \text{flight} @ \text{travelshop} \rangle \\
apm \langle \text{travelshop}, \text{standard} \rangle & \stackrel{\triangleq}{=} \text{get}^+ \langle \text{info} @ \text{standard} \rangle \\
apm \langle \text{travelshop}, \text{special} \rangle & \stackrel{\triangleq}{=} \text{put} \langle \text{info} @ \text{special} \rangle \\
apm \langle \text{standard}, \text{special} \rangle & \stackrel{\triangleq}{=} \text{get}^+ \langle \text{info} @ \text{special} \rangle \circ \text{get}^+ \langle \text{info} @ \text{standard} \rangle \\
apm \langle \text{special}, \text{bank} \rangle & \stackrel{\triangleq}{=} \text{put} \langle \text{pay} @ \text{bank} \rangle \circ \text{get}^- \langle \text{info} @ \text{special} \rangle \circ \text{get}^- \langle \text{pay} @ \text{bank} \rangle \\
apm \langle \text{travelshop}, \text{bank} \rangle & \stackrel{\triangleq}{=} \text{get}^+ \langle \text{pay} @ \text{bank} \rangle \\
\end{align*}\]

Table 2. PETiMo network modelling the running example together with the relevant access permission modification operations (those omitted are all equal to the identity mapping on AccPerm).
which are applied to the typed process expressions when they move around the five nodes of the network. Table 3 gives all the necessary process definitions.\footnote{Regarding the \texttt{flightinfo} process, it is worth noting that we can define strategies of migration based on timing; for instance, round-robin etc.}

\begin{verbatim}
client(init:eMoney) ≡
go^Δ2 travelshop → flight ? (standardoffer:Loc) →
go^Δ4 standardoffer → info ? (p1:eMoney, specialoffer:Loc) →
go^Δ3 specialoffer → info ? (p2:eMoney, paying:Loc) →
go^Δ6 paying → pay ! (min\{p1, p2\}) →
go^Δ4 home → client(init − min\{p1, p2\})

agent(balance:eMoney) ≡
flight ! (standard) → go^Δ10 bank →
pay ? (profit:eMoney) → go^Δ12 travelshop →
agent(balance + profit)

update(saleprice:eMoney) ≡
go^Δ10 special → info ! (saleprice) → stop

flightinfo(price : eMoney, next : Loc) ≡
info ! (price, next) → flightinfo(price, next)

saleinfo(price : eMoney, next : Loc) ≡
info^Δ10 ? (newprice:eMoney)
then saleinfo(newprice, next)
else info ! (price, next) → saleinfo(price, next)

till(cash:eMoney) ≡
pay^Δ1 ? (newpayment:eMoney)
then till(cash + newpayment)
else pay^Δ2 ! (cash) then till(0) else till(cash)

Table 3. Process definitions for the running example.
\end{verbatim}

2.2 Operational Semantics

The first component of the operational semantics of \textsc{Pertimo} is the structural equivalence \(≡\) on networks, similar to that used in [5]. It is the smallest congru-
ence such that the equalities (Eq1–Eq6) in Table 4 hold. Its role is to rearrange a network in order to apply the action rules which are also given in Table 4.

Using (Eq1–Eq6) 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 \( l_i \parallel P_i : \Gamma_i \) will be called a component of \( N \), the set of all components will be denoted by \( \text{comp}(N) \), and the parallel composition (2) will be 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 also Remark 1 below).

In addition to the rules of the structural congruence, Table 4 introduces two kinds of action rules:

\[
N \xrightarrow{\lambda} N' \quad \text{and} \quad N \xrightarrow{\delta} N'.
\]

The former is an execution of an action \( \lambda \), and the latter a time step. In the rule (TIME), \( N \not\xrightarrow{\lambda} \) means that no \( \lambda \)-action \( \lambda \) (i.e., an action of the form \( \text{id} @ l \) or \( l \circ l' \) or \( @l \) or \( a(u) @ l \)) can be applied to \( N \). Moreover, \( \delta_0(N) \) is obtained by taking the component decomposition of \( N \) and simultaneously replacing all components of the form:

\[
l \parallel a^\Delta \omega \text{ then } P \text{ else } Q : \Gamma
\]

where \( \omega \) stands for !⟨\( u \)⟩ or ?⟨(\( u \):\text{X})⟩, by \( l \parallel Q : \Gamma \) if \( t = 0 \), and otherwise by:

\[
l \parallel a^{\Delta t} \omega \text{ then } P \text{ else } Q : \Gamma
\]

After that all the occurrences of the special symbol \( \oplus \) are erased. Note that decrementing the values of timers in the migration construct is taken care of by the (WAIT) rule.

Intuitively, the way networks of located processes evolve can be regarded as conforming to the maximally concurrent paradigm since one executes as many as possible actions before applying a time decrement which signifies the passage of a unit of time.

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

\[\text{Properties of Operational Semantics} \quad \text{We now discuss properties of the operational semantics defined in Table 4 aiming, in particular, at capturing concurrency in the executed action moves. We start by showing that one cannot execute an unbounded sequence of action moves without executing any time moves.}\]
\[
\begin{align*}
(N|N')|N'' & \equiv N|(N'|N'') \\
(N|N')|N'' & \equiv N|(N'|N'') \\
I[PP|PP'] & \equiv I[PP]|I[PP'] \\
I[P|Q : \Gamma] & \equiv I[P : \Gamma|Q : \Gamma] \\
I[PP|PP'] & \equiv I[PP'|PP] \\
I[(PP|PP')|PP''] & \equiv I[PP|(PP'|PP'')] \\
\text{(CALL)} & \quad I[id(v) : \Gamma] \xrightarrow{id@} I[\otimes\{v/u\}P : \Gamma] \\
\text{(MOVE)} & \quad I[\text{go}^\Delta t l' \text{ then } P : \Gamma] \xrightarrow{\text{int}'} I'[\otimes P : \text{apm}(l,l') (\Gamma)] \\
\text{(WAIT)} & \quad t > 0 \\
I[\text{go}^\Delta t l' \text{ then } P : \Gamma] \xrightarrow{\text{go}^{\Delta -1} l} I[\otimes \text{go}^\Delta l' \text{ then } P : \Gamma] \\
\text{(COM)} & \quad \frac{\text{put}(a@l) \in \Gamma \quad \text{get}(a@l) \in \Gamma' \quad v \in \prod X}{I[a^\Delta t! (v) \text{ then } P \text{ else } Q : \Gamma' | a^\Delta t?' (u:X) \text{ then } P' \text{ else } Q' : \Gamma']} \xrightarrow{a(u)@} I[\otimes P : \Gamma | \otimes \{v/u\} P' : \Gamma'] \\
\text{(PAR)} & \quad N \xrightarrow{\lambda} N' \\
N | N'' \xrightarrow{\lambda} N' | N'' \\
\text{(EQUIV)} & \quad N \equiv N' \quad N' \xrightarrow{\lambda} N'' \quad N'' \equiv N''' \\
\text{(TIME)} & \quad N \xrightarrow{\phi_i} N \\
\end{align*}
\]

\textbf{Table 4.} Six rules of the structural equivalence (Eq1-Eq6), and seven action rules (CALL MOVE WAIT COM PAR EQUIV TIME) of the operational semantics of PerTiMo.
Proposition 1. If $N$ is a network and $N \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} N'$ then $k \leq |\text{comp}(N)|$.

Proof. We observe that each of the components of $N$ is involved in generating of at most one $\lambda_i$ (since the resulting subexpression is blocked by $\ominus$ until the next time step), and that the generation of each $\lambda_i$ involves at least one component of $N$. \hfill $\square$

If we start with a well-formed network, execution proceeds through alternating executions of time steps and contiguous sequences of actions making up what can be regarded as a maximally concurrent step (note the role of the special stalling symbols $\ominus$). 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 also have $N \xrightarrow{\lambda_{i_1}} \cdots \xrightarrow{\lambda_{i_n}} N'$, for every permutation $i_1, \ldots, i_n$ of $1, \ldots, n$.

Proof. We observe that no component is involved in the generation of two $\lambda_i$’s (since the resulting subexpression is blocked by $\ominus$ until the next time move), and the executions in different components do not interfere with each other. \hfill $\square$

In view of the last result, an entire computational step is captured by a derivation $N \xrightarrow{A} N'$, where $A = \{\lambda_1, \ldots, \lambda_n\}$ is a finite multiset of $l$-actions for some location $l$ such that

$$N \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} \xrightarrow{l} N' .$$

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 $A$. We can also show that the operational semantics does not admit infinite branching.

Proposition 3. There are only finitely many networks directly reachable from a given network.

Proof. Each component of a network can be involved in only a single action. Moreover, such a single involvement can only be realized in finitely many ways (note that in the case of an input construct this follows from the assumed finiteness of the types in PERTiMo). Finally, the time evolution is deterministic. \hfill $\square$

It is important to stress that the semantical treatment of PERTiMo — itself a continuation of the idea developed for TiMo [10] — goes beyond interleaving semantics by introducing an explicit representation of maximal concurrency in the execution of actions.

Our last result in this section is that the rules of Table 4 preserve well-formedness of networks.

Proposition 4. All networks directly reachable from a well-formed network are well-formed.
Proof. Let \( N \) be a well-formed network and \( N \xrightarrow{\phi_t} N' \).

Clearly, there are no occurrences of the special symbol \( \otimes \) in \( N' \) since the function \( \phi_t \) applied in a time move removes all its occurrences.

We then observe that \( N' \) has no free variables. The only two cases in Table 4 which need to be checked are (CALL) and (COM). Applying (CALL) does not introduce free variables since \( \text{fv}(P_u) \subseteq \{u_1, \ldots, u_{m_u}\} \) in the recursive definition (1). In an application of the (COM) rule \( v \), the values replacing \( u \) in \( P' \), and we have \( \text{fv}(P') \subseteq \{u\} \). Hence \( \text{fv}(\{v/u\}P') = \emptyset \). In the case of an application of the \( \phi_t \) function, we observe that the construct:

\[ a^{\Delta \theta} \bot (u:X) \text{ then } P \text{ else } Q : \Gamma \]

binds the variables \( u \) within \( P \), but not within \( Q \).

Finally, for every \( id(v) \) occurring in \( N' \), \( v_i \) is of type \( X_i^{id} \) which follows from the assumed well-formedness of \( N \). \( \square \)

Running Example: Tables 5 and 6 give three execution steps based on the scenario illustrated in Figure 1. Each step represents parallel execution of several actions, and in Table 5 we indicate only the main rules used in the derivation of steps. In particular, six instances of (CALL) are applied in the first step to the six processes which make up the TravelShop system. In the second step (MOVE) is applied twice, corresponding to the (non-urgent) migration of the client from home to travelshop, and the urgent migration of the update process from travelshop to special. In the third step, two instances of (COM) are used: one to communicate the location of the standard offer (i.e., standard) over channel flight at travelshop, and the other to update the special offer to 60 using the channel info at the location special.

Each parallel execution step takes a single unit of local time and some timers are decremented by one (for example, the timer \( \Delta \beta \) of channel info in \( U_0 \) is changed to \( \Delta \gamma \) in \( U_1 \)). Other timers which have expired cause an immediate migration (see the migration of the update process in the second execution step) or the selection of the alternative part of a communication action (see \( W_1 \) which is replaced by \( W_2 \) in the third execution step).

Note that the last network expression derived from TravelShop of Table 5 corresponds to the intermediate configuration shown in Figure 1(b). Note also that in the pictorial representation of Figure 1(b) we show the home location, even though it is not present in the last network expression in Table 5. 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 in Figure 1(c) after client has completed its migration and came back to its initial location.

3 Safe Access Permissions for Migrating Processes

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 at-
TravelShop

\[
\begin{align*}
\{ \text{client}@\text{home}, & \text{agent}@\text{travelshop}, \text{update}@\text{travelshop}, \} \\
\{ \text{flightinfo}@\text{standard}, & \text{saleinfo}@\text{special}, \text{tilt}@\text{bank} \}
\end{align*}
\]

6 \times (\text{Call})

\(\text{home} \mid \text{go}^A \text{travelshop} \rightarrow P_0 : \emptyset \) |
\(\text{travelshop} \mid Q_0 : \{ \text{put}(\text{flight}@\text{travelshop}) \} \mid \text{go}^A \text{travelshop} \rightarrow R_0 : \emptyset \) |
\(\text{standard} \mid U_0 : \{ \text{put}(\text{info}@\text{standard}), \text{get}(\text{info}@\text{standard}) \} \) |
\(\text{special} \mid V_0 : \{ \text{put}(\text{info}@\text{special}), \text{get}(\text{info}@\text{special}) \} \) |
\(\text{bank} \mid W_0 : \{ \text{put}(\text{pay}@\text{bank}), \text{get}(\text{pay}@\text{bank}) \} \)

2 \times (\text{Move})

\(\text{travelshop} \mid \text{flight} ? (\text{standardoffer} : \text{Loc}) \rightarrow P_1 : \{ \text{get}(\text{flight}@\text{travelshop}) \} \) |
\(\text{flight} ! (\text{standard}) \rightarrow Q_1 : \{ \text{put}(\text{flight}@\text{travelshop}) \} \) |
\(\text{standard} \mid U_1 : \{ \text{put}(\text{info}@\text{standard}), \text{get}(\text{info}@\text{standard}) \} \) |
\(\text{special} \mid \text{info}^A ? (\text{newprice} : \text{eMoney}) \rightarrow V_1 : \{ \text{put}(\text{info}@\text{special}), \text{get}(\text{info}@\text{special}) \} \) |
\(\text{info} ! (60) \rightarrow \text{stop} : \{ \text{put}(\text{info}@\text{special}) \} \) |
\(\text{bank} \mid W_1 : \{ \text{put}(\text{pay}@\text{bank}), \text{get}(\text{pay}@\text{bank}) \} \)

2 \times (\text{Com})

\(\text{travelshop} \mid P_2 : \{ \text{get}(\text{flight}@\text{travelshop}) \} \mid Q_1 : \{ \text{put}(\text{flight}@\text{travelshop}) \} \) |
\(\text{standard} \mid U_2 : \{ \text{put}(\text{info}@\text{standard}), \text{get}(\text{info}@\text{standard}) \} \) |
\(\text{special} \mid V_2 : \{ \text{put}(\text{info}@\text{special}), \text{get}(\text{info}@\text{special}) \} \mid \text{stop} : \{ \text{put}(\text{info}@\text{special}) \} \) |
\(\text{bank} \mid W_2 : \{ \text{put}(\text{pay}@\text{bank}), \text{get}(\text{pay}@\text{bank}) \} \)

Table 5. Three execution steps for the running example (see Table 6).

tempts 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 \(\text{apm}\). 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 migration of processes in a network.

In what follows, we use judgements of the form

\[ \Gamma \vdash_1 P \]

to mean that a single-component network \(l \mid P : \Gamma \) has safe access permissions.
\[ P_0 = \text{flight}? (\text{standardoffer}:\text{Loc}) \rightarrow P_1 \]
\[ P_1 = \text{go}^A \text{ standardoffer} \rightarrow \text{info}? (p1:\text{eMoney}, \text{specialoffer}:\text{Loc}) \rightarrow \]
\[ \text{go}^A \text{ specialoffer} \rightarrow \text{info}? (p2:\text{eMoney}, \text{paying}:\text{Loc}) \rightarrow \]
\[ \text{go}^A \text{ paying} \rightarrow \text{pay}! (\min\{p1, p2\}) \rightarrow \]
\[ \text{go}^A \text{ home} \rightarrow \text{client}(130 - \min\{p1, p2\}) \]
\[ P_2 = \{\text{standard}/\text{standardoffer}\} P_1 \]
\[ Q_0 = \text{flight}! (\text{standard}) \rightarrow Q_1 \]
\[ Q_1 = \text{go}^{A\tau} \text{ bank} \rightarrow \]
\[ \text{pay}? (\text{profit}:\text{eMoney}) \rightarrow \text{go}^{A\tau} \text{ travelshop} \rightarrow \text{agent}(100 + \text{profit}) \]
\[ R_0 = \text{info}! (60) \rightarrow \text{stop} \]
\[ U_0 = \text{info}^{A\tau} ! (110, \text{special}) \rightarrow \text{flightinfo}(110, \text{special}) \]
\[ U_1 = \text{info}^{A\tau} ! (110, \text{special}) \rightarrow \text{flightinfo}(110, \text{special}) \]
\[ U_2 = \text{flightinfo}(110, \text{special}) \]
\[ V_0 = \text{info}^{A\tau} ? (\text{newprice}:\text{eMoney}) \text{ then } \text{saleinfo}(\text{newprice}, \text{bank}) \]
\[ \text{else } \text{info}! (90, \text{bank}) \rightarrow \text{saleinfo}(90, \text{bank}) \]
\[ V_1 = \text{info}^{A\tau} ? (\text{newprice}:\text{eMoney}) \text{ then } \text{saleinfo}(\text{newprice}, \text{bank}) \]
\[ \text{else } \text{info}! (90, \text{bank}) \rightarrow \text{saleinfo}(90, \text{bank}) \]
\[ V_2 = \text{saleinfo}(60, \text{bank}) \]
\[ W_0 = \text{pay}^{A\tau} ? (\text{newpayment}:\text{eMoney}) \text{ then } \text{till}(10 + \text{newpayment}) \]
\[ \text{else } \text{pay}^{A\tau} ! (10) \text{ then till}(0) \text{ else till}(10) \]
\[ W_1 = \text{pay}^{A\tau} ? (\text{newpayment}:\text{eMoney}) \text{ then } \text{till}(10 + \text{newpayment}) \]
\[ \text{else } \text{pay}^{A\tau} ! (10) \text{ then till}(0) \text{ else till}(10) \]
\[ W_2 = \text{pay}^{A\tau} ! (10) \text{ then till}(0) \text{ else till}(10) \]

**Table 6.** Process expressions used in Table 5.

Given a set of locations Loc together with the apm mapping as well a process \( P \) and location \( l \), we want to devise typing rules for checking that a set of access permissions \( \Gamma \) satisfies \( \Gamma \vdash_l P \). However, this may be impossible due to conflicts between access permission modifications resulting from migrations, and the subsequent communication actions. For example, if

\[ P = \text{go}^{A\tau} l' \text{ then } a^{A\tau} ! l \rightarrow \text{stop} \]

and \( \text{apm}(l, l') = \text{put}_{a^{l'\downarrow}}^- \), then there is no \( \Gamma \) such that \( \Gamma \vdash_l P \).
If \( P \) does not involve recursive definitions, the task is relatively straightforward. One just needs to follow the syntactic structure of the process and incrementally derive \( \Gamma \). Dealing with recursive process definitions is more complicated and the solution we propose consists in unfolding a recursive process expression sufficiently many times to cover all migration possibilities. For all \( id \in Id, n \geq 0 \) and \( v \in \prod X^{id} \), the \( n \)-th unfolding of \( id(v) \) is given by:

\[
id(v)^\uparrow n \triangleq \begin{cases} 
\text{stop} & \text{if } n = 0 \\
P & \text{if } n > 0
\end{cases}
\]

where \( P \) is obtained from \( \{v/u\}P_{id} \) by replacing each subexpression of the form \( id'(w) \) with \( id'(w)^\uparrow n-1 \). Moreover, \( \text{stop}^\uparrow n \triangleq \text{stop} \).

The derivation rules for judgements \( \Gamma \vdash_1 P \) are given in Table 7. The constant \( H \) in the rule (TRec) for recursive processes is:

\[
H \triangleq 2 \cdot |\text{Loc}| \cdot \left( 1 + \sum_{id \in Id} |X^{id}_1| \cdot \ldots \cdot |X^{id}_{m_{id}}| \right).
\]

The value of the above constant comes from rather technical considerations needed to prove results concerning safe access permissions of process networks presented later on.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Derivation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TSub)</td>
<td>( \Gamma' \subseteq \Gamma ) ( \Gamma' \vdash_1 P )</td>
<td>( \Gamma \vdash_1 P )</td>
</tr>
<tr>
<td>(TStop)</td>
<td>( \emptyset \vdash_1 \text{stop} )</td>
<td>( \emptyset \vdash_1 P )</td>
</tr>
<tr>
<td>(TMove)</td>
<td>( \text{apm}(l, \Gamma)(\Gamma) \vdash_1 ) ( P )</td>
<td>( \Gamma \vdash_1 \text{go}^{\Delta t} l' ) then ( P )</td>
</tr>
<tr>
<td>(TOut)</td>
<td>( \text{put}(a@l) \in \Gamma ) ( \Gamma \vdash_1 P ) ( \Gamma \vdash_1 Q )</td>
<td>( \Gamma \vdash_1 a^{\Delta t} ! \langle v \rangle ) then ( P ) else ( Q )</td>
</tr>
<tr>
<td>(TIn)</td>
<td>( \text{get}(a@l) \in \Gamma ) ( \forall v \in \prod X : \Gamma \vdash_1 {v/u}P ) ( \Gamma \vdash_1 Q )</td>
<td>( \Gamma \vdash_1 a^{\Delta t} ? \langle u \rangle \rangle ) then ( P ) else ( Q )</td>
</tr>
<tr>
<td>(TRec)</td>
<td>( \Gamma \vdash id(v)^{\uparrow H} )</td>
<td>( \Gamma \vdash id(v) )</td>
</tr>
<tr>
<td>(TPar)</td>
<td>( \Gamma \vdash_1 P ) ( \Gamma \vdash_1 Q )</td>
<td>( \Gamma \cup \Gamma \vdash_1 P \lor Q )</td>
</tr>
</tbody>
</table>

Table 7. Derivation rules for processes with safe access permissions.
The \( \text{(TMOVE)} \) rule concerns a migration from location \( l \) to \( l' \). In order to have \( l \parallel \text{go}^{\Delta t} l' \) then \( P : \Gamma \) with safe access permissions, it is necessary to have \( l' \parallel [P : \Gamma'] \) 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 \( \text{(TOUT)} \) simply requires that a process attempting to send a message along a channel \( a \) should possess the access permission \( \text{put}(a@l) \). Similarly, the rule \( \text{(TIN)} \) requires that a process attempting to receive a message along a channel \( a \) should possess the access permission \( \text{get}(a@l) \); moreover, after receiving this message it should have safe access permissions in terms of access permissions with the current \( \Gamma \) irrespective of the values carried by that message.

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 has safe access permissions. These two definitions are consistent in the sense that \( \Gamma \sqsupseteq \sigma_P \) iff \( \Gamma' \sqsupseteq \sigma_{P_i} \) for every component network \( l[ P_i; \Gamma] \) of a single-component network \( l[ P; \Gamma] \), which follows from the rule \( \text{(TPAR)} \).

**Properties of Networks with Safe Access Permissions** Our first main result will be that being a network with safe access permissions is preserved over the evolutions defined by our operational semantics. Its formulation and proof are preceded by a series of auxiliary definitions and results.

Throughout the rest of this section we will assume that the right hand side \( P_{id} \) of each recursive definition \( (1) \) is either a primitive process (i.e., it is of the form \( P_{id} = \text{stop} \) or \( P_{id} = id'(w) \)) or \( P_{id} \) uses exactly one application of one of the process operators to some primitive process(es). (We will later argue in Remark 2 at the end of this section that such an assumption does not diminish the generality of our results.) Moreover, we will use signed access permissions of the form \( \alpha^+ \) and \( \alpha^- \). Intuitively, \( \alpha^+ \) corresponds to adding of the access permission \( \alpha \), and \( \alpha^- \) corresponds to its withdrawal.

The auxiliary results that we now develop will refer to an arc-labelled graph \( G_{id} \) which encodes potential unfoldings of process identifiers, defined thus:

- The vertex set of \( G_{id} \) is: \( (\{\text{stop}\} \cup \{id(v) \mid id \in Id \land v \in \prod X') \times \text{Loc} \), and so there are exactly \( |I| \) vertices.
- There are no arcs outgoing from the vertices in \( \{\text{stop}\} \times \text{Loc} \).
- For each vertex \( q = (id(v), l) \) the outgoing arcs depend on the right hand side \( P_{id} \) of the recursive definition for \( id \), as follows:
  - If \( \{v/u\}P_{id} = \text{stop} \) then there is an \( \varnothing \)-labelled arc from \( q \) to \( (\text{stop}, l) \).
  - If \( \{v/u\}P_{id} = id'(w) \) then there is an \( \varnothing \)-labelled arc from \( q \) to \( (id'(w), l) \).
  - If \( \{v/u\}P_{id} = P' | P'' \) then there are \( \varnothing \)-labelled arcs from \( q \) to \( (P', l) \) and \( (P'', l) \).
  - If \( \{v/u\}P_{id} = a^{\Delta t}!(v) \) then \( P' \) else \( P'' \) then there are arcs (labelled by \( \{\text{put}(a@l)\} \)) from \( q \) to \( (P', l) \) and \( (P'', l) \).
  - If \( \{v/u\}P_{id} = a^{\Delta t}?(z:Y) \) then \( P' \) else \( P'' \) then there are arcs (labelled by \( \{\text{get}(a@l)\} \)) from \( q \) to \( (P', l) \) and \( (w/z)P', l) \), for every \( w \in \prod Y \).
• If \( \{v/u\}P_{id} = go^{At} l' \) then \( P' \) then there is an arc from \( q \) to \( (P', l') \) labelled by:

\[
A = \{ \alpha^+ \mid \alpha \in apm(l, l')(\emptyset) \} \cup \{ \alpha^- \mid \alpha \notin apm(l, l')(\text{AccPerm}) \}.
\]

Note that this means that if \( \alpha^+ \in A \) (or \( \alpha^- \in A \)) then \( \alpha \in apm(l, l')(\Gamma) \) (resp. \( \alpha \notin apm(l, l')(\Gamma) \)), for every set of access permissions \( \Gamma \).

In what follows, for any path \( \pi \) starting at a vertex \( q' \) and an arc from \( q \) to vertex \( q' \) (note that there is at most one such arc), we denote by \( \langle q, q' \rangle \circ \pi \) the path obtained by first following this arc and then \( \pi \).

Intuitively, \( G_{id} \) encodes changes of the set of access permissions of a migrating process (using \( \alpha^+ \) and \( \alpha^- \)) and the required access permissions of a communicating process (using \( \alpha \)), while the directed paths correspond to the potential evolutions of a migrating process. Some of these paths correspond to potentially unsafe situation, as described next.

Let \( \alpha \) be an access permission. Then a directed path \( \pi \) in \( G_{id} \) of the form:

\[
q \xrightarrow{A_1} q_1 \xrightarrow{A_2} \ldots \xrightarrow{A_n} q_n
\]

is called

- an \( \alpha \)-path if \( \alpha \in A_n \) and \( \alpha, \alpha^+, \alpha^- \notin A_i \) for \( i = 0, \ldots, n-1 \).
- an \( \alpha^- \)-path if \( \alpha \in A_n \) and there is \( m < n \) such that \( \alpha^- \in A_m \) and \( \alpha, \alpha^+, \alpha^- \notin A_i \) for \( i = m+1, \ldots, n-1 \).

We denote this by \( \pi^\alpha_n(q) \) and \( \pi^\alpha^-_{n}(q) \), respectively. Then, for every set of access permissions \( \Gamma \), every vertex \( q \) of \( G_{id} \), and every \( n \geq 0 \),

\[
\Gamma \models^\alpha q
\]

if \( \pi^\alpha_n(q) = \emptyset \) and \( \pi^\alpha^-_{n}(q) = \emptyset \Rightarrow \alpha \in \Gamma \), for every access permission \( \alpha \). Moreover,

\[
\Gamma \models q
\]

if \( \Gamma \models^\alpha q \) for all \( n \geq 0 \). Intuitively, by writing \( \Gamma \models q \), where \( q = (id(v), l) \), we would like to capture the fact that the component \( l \models [id(v) : \Gamma] \) has safe access permissions in any environment made up of other components, and since its potential evolutions correspond to the paths starting at \( q \), we can use such paths to verify whether the component has safe access permissions. For example, if there is an \( \alpha \)-path starting at \( q \) with \( \alpha \notin \Gamma \), there is a potential evolution in which one of the descendants of \( id(v) : \Gamma \) placed initially at location \( l \) wants to communicate on a channel without the necessary access permission. Similarly, \( \alpha^- \)-paths are indicative of potential attempts to communicate on channels after losing the necessary access permissions.

We will link the notion of a component having safe access permissions with the presence of \( \alpha \)-paths and \( \alpha^- \)-paths in \( G_{id} \).

**Lemma 1.** Let \( q = (id(v), l) \) be a vertex of \( G_{id} \), \( \Gamma \) a set of access permissions, \( \alpha \) an access permission, and \( n \geq 0 \).
1. If \( \{v/u\} p_{id} = id'(w) \) then
\[
\Gamma \vdash^{n+1} q \iff \Gamma \vdash^n q'
\]
where \( q' = (id'(w), l) \).

2. If \( \{v/u\} p_{id} = P' \mid P'' \) then
\[
\Gamma \vdash^{n+1} q \iff \Gamma \vdash^n q' \land \Gamma \vdash^n q''
\]
where \( q' = (P', l) \) and \( q'' = (P'', l) \).

3. If \( \{v/u\} p_{id} = a \Delta \alpha ! (v) \) then \( P' \text{ else } P'' \) then
\[
\Gamma \vdash^{n+1} q \iff \text{put}(a@l) \in \Gamma \land \Gamma \vdash^n q' \land \Gamma \vdash^n q''
\]
where \( q' = (P', l) \) and \( q'' = (P'', l) \).

4. If \( \{v/u\} p_{id} = a \Delta \alpha ? (z:Y) \) then \( P' \text{ else } P'' \) then
\[
\Gamma \vdash^{n+1} q \iff \text{get}(a@l) \in \Gamma \land \Gamma \vdash^n q'' \land \forall q' \in Q : \Gamma \vdash^n q'
\]
where \( q'' = (P'', l) \) and \( Q = \{(w/z)P', l) \mid w \in \prod Y\} \).

5. If \( \{v/u\} p_{id} = \text{go} \Delta \alpha l' \) then \( P' \) then
\[
\Gamma \vdash^{n+1} q \iff \Gamma \setminus \{\alpha \mid \alpha^- \in A\} \cup \{\alpha \mid \alpha^+ \in A\} \vdash^n q'
\]
where \( A \) is the label of the arc from \( q \) to \( q' \), and \( q' = (P', l') \).

Proof. (1) From the definition of \( G_{id} \) we obtain that:
\[
\pi_{\alpha^+}^n(q) = \{(q, q') \circ \pi \mid \pi \in \pi^n_{\alpha}(q')\}
\pi_{\alpha^-}^n(q) = \{(q, q') \circ \pi \mid \pi \in \pi^n_{\alpha^-}(q')\}.
\]

Hence \( \Gamma \vdash^{n+1} q \iff \Gamma \vdash^n q' \).

(2) From the definition of \( G_{id} \) we obtain that:
\[
\pi_{\alpha^+}^n(q) = \{(q, q') \circ \pi \mid \pi \in \pi^n_{\alpha}(q')\} \cup \{(q, q'') \circ \pi \mid \pi \in \pi^n_{\alpha}(q'')\}
\pi_{\alpha^-}^n(q) = \{(q, q') \circ \pi \mid \pi \in \pi^n_{\alpha^-}(q')\} \cup \{(q, q'') \circ \pi \mid \pi \in \pi^n_{\alpha^-}(q'')\}.
\]

Hence \( \Gamma \vdash^{n+1} q \iff \Gamma \vdash^n q' \land \Gamma \vdash^n q'' \).

(3) From the definition of \( G_{id} \) we obtain that, for \( \alpha \neq \text{put}(a@l) \):
\[
\pi_{\alpha^+}^n(q) = \{(q, q') \circ \pi \mid \pi \in \pi^n_{\alpha}(q')\} \cup \{(q, q'') \circ \pi \mid \pi \in \pi^n_{\alpha}(q'')\}
\pi_{\alpha^-}^n(q) = \{(q, q') \circ \pi \mid \pi \in \pi^n_{\alpha^-}(q')\} \cup \{(q, q'') \circ \pi \mid \pi \in \pi^n_{\alpha^-}(q'')\}.
\]

Moreover, if \( \alpha = \text{put}(a@l) \) then
\[
\pi_{\alpha^+}^n(q) = \{(q, q'), (q, q'')\}
\pi_{\alpha^-}^n(q) = \{(q, q') \circ \pi \mid \pi \in \pi^n_{\alpha^-}(q')\} \cup \{(q, q'') \circ \pi \mid \pi \in \pi^n_{\alpha^-}(q'')\}.
\]
Hence $\Gamma \models_{-n+1} q \iff \text{put}(a@l) \in \Gamma \land \Gamma \models_{-n} q' \land \Gamma \models_{-n} q''$.

(4) From the definition of $G_{td}$ we obtain that, for $\alpha \neq \text{get}(a@l)$:

$$\pi_{\alpha}^{n+1}(q) = \{ (q, q') : \pi \models_{\alpha}^{n} (q') \} \cup \bigcup_{u \in Q} \{ (q, q') : \pi \models_{\alpha}^{n} (q') \}$$

$$\pi_{\alpha^{-}\alpha}^{n+1}(q) = \{ (q, q') : \pi \models_{\alpha^{-}\alpha}^{n} (q') \} \cup \bigcup_{u \in Q} \{ (q, q') : \pi \models_{\alpha^{-}\alpha}^{n} (q') \}.$$ 

Moreover, if $\alpha = \text{get}(a@l)$ then

$$\pi_{\alpha}^{n+1}(q) = \{ (q, q') \} \cup \{ (q, q') : \pi \models_{\alpha}^{n} (q') \}$$

$$\pi_{\alpha^{-}\alpha}^{n+1}(q) = \{ (q, q') : \pi \models_{\alpha^{-}\alpha}^{n} (q') \} \cup \bigcup_{u \in Q} \{ (q, q') : \pi \models_{\alpha^{-}\alpha}^{n} (q') \}.$$ 

Hence $\Gamma \models_{-n+1} q \iff \text{get}(a@l) \in \Gamma \land \Gamma \models_{-n} q'' \land \forall q' \in Q : \Gamma \models_{-n} q'$.

(5) From the definition of $G_{td}$ we obtain that, for $\alpha$ such that $\alpha^{-}, \alpha^{+} \notin A$:

$$\pi_{\alpha}^{n+1}(q) = \{ (q, q') : \pi \models_{\alpha}^{n} (q') \}$$

$$\pi_{\alpha^{-}\alpha}^{n+1}(q) = \{ (q, q') : \pi \models_{\alpha^{-}\alpha}^{n} (q') \}.$$ 

Moreover, if $\alpha^{+} \in A$, then

$$\pi_{\alpha}^{n+1}(q) = \emptyset$$

$$\pi_{\alpha^{-}\alpha}^{n+1}(q) = \{ (q, q') : \pi \models_{\alpha^{-}\alpha}^{n} (q') \}.$$ 

and, if $\alpha^{-} \in A$, then

$$\pi_{\alpha}^{n+1}(q) = \emptyset$$

$$\pi_{\alpha^{-}\alpha}^{n+1}(q) = \{ (q, q') : \pi \models_{\alpha^{-}\alpha}^{n} (q') \land \pi \models_{\alpha}^{n} (q') \}.$$ 

To show the desired equivalence, we denote $\Gamma' = \Gamma \setminus \{ \alpha \mid \alpha^{-} \in A \} \cup \{ \alpha \mid \alpha^{+} \in A \}$, and then proceed as follows.

($\iff$) We consider two cases.

Case 1: $\pi \in \pi_{\alpha}^{n+1}(q)$ and $\alpha \notin \Gamma$. Then $\alpha^{-}, \alpha^{+} \notin A$ and we have that $\pi = (q, q')$ for some $\pi' \in \pi_{\alpha}^{n}(q')$ as well as $\alpha \notin \Gamma'$.

Case 2: $\pi \in \pi_{\alpha^{-}\alpha}^{n+1}(q)$. Then $\pi = (q, q')$ for $\pi' \in \pi_{\alpha^{-}\alpha}^{n}(q')$ or $\pi' \in \pi_{\alpha}^{n}(q')$ and $\alpha^{-} \in A$ and so $\alpha \notin \Gamma'$.

($\implies$) We consider two cases.

Case 1: Suppose that $\pi \in \pi_{\alpha^{-}\alpha}^{n}(q')$. Then $(q, q') \in \pi_{\alpha^{-}\alpha}^{n+1}(q)$.

Case 2: Suppose now that $\pi \in \pi_{\alpha}^{n}(q')$ and $\alpha \notin \Gamma'$. Then $\alpha^{+} \notin A$. Now, if $\alpha^{-} \in A$ then $(q, q') \in \pi_{\alpha^{-}\alpha}^{n+1}(q)$, and if $\alpha^{-} \notin A$ then $(q, q') \in \pi_{\alpha}^{n+1}(q)$ and $\alpha \notin \Gamma'$. □

**Lemma 2.** Let $q = (P, l)$ be a vertex of $G_{td}$, $\Gamma$ a set of access permissions, and $n \geq 0$. Then

$$\Gamma \models_{-n} q \iff \Gamma \vdash_{1} P^{\ast n}.$$
Proof. If \( P = \text{stop} \) the equivalence follows from the rules (TStop) and (TSub). Hence we assume that \( P = \text{id}(v) \) and proceed by induction on \( n \).

In the base case, \( n = 0 \) we have that \( \text{id}(v)^{\uparrow 0} = \text{stop} \) and so the result holds by the rules (TStop) and (TSub). In the induction step, assuming that the result holds for \( n \), we proceed by case analysis to show that it also holds for \( n + 1 \).

Case 1: \( \{v/u\}P_{id} = \text{go}^{\Delta t} l' \) then \( P' \). Then, assuming \( q' = (P', l') \):

\[
\Gamma \vdash n^{+1} q \iff \text{by Lemma 1(5)} \\
apm(l, l')(\Gamma) \vdash n q' \iff \text{by ind. hyp. and the stop part} \\
apm(l, l')(\Gamma) \vdash_{\nu} P''^{\uparrow n} \iff \text{TMove} \\
\Gamma \vdash_{\nu} P''^{\uparrow n+1}.
\]

Case 2: \( \{v/u\}P_{id} = P' \mid P''. \) Then, assuming \( q' = (P', l) \) and \( q'' = (P'', l) \):

\[
\Gamma \vdash n^{+1} q \iff \text{by Lemma 1(2)} \\
\Gamma \vdash n q' \land \Gamma \vdash n q'' \iff \text{by ind. hyp. and the stop part} \\
\Gamma \vdash_{\nu} P''^{\uparrow n} \land \Gamma \vdash_{\nu} P''^{\uparrow n} \iff \text{TPAR} \\
\Gamma \vdash_{\nu} P''^{\uparrow n+1}.
\]

Case 3: \( \{v/u\}P_{id} = a^{\Delta t} ! (v) \) then \( P' \) else \( P'' \). Then, assuming \( q' = (P', l) \) and \( q'' = (P'', l) \):

\[
\Gamma \vdash n^{+1} q \iff \text{by Lemma 1(3)} \\
\text{put}(a@l) \in \Gamma \land \Gamma \vdash n q' \land \Gamma \vdash n q'' \iff \text{by ind. hyp. and the stop part} \\
\Gamma \vdash_{\nu} P''^{\uparrow n} \land \Gamma \vdash_{\nu} P''^{\uparrow n} \iff \text{TOUt} \\
\Gamma \vdash_{\nu} P''^{\uparrow n+1}.
\]

Case 4: \( \{v/u\}P_{id} = a^{\Delta t} ? (z:Y) \) then \( P' \) else \( P'' \). Then, assuming \( q'' = (P'', l) \) and \( Q = \{(w/z)P', l \mid w \in [\prod Y] \} \) and \( P = \{(w/z)P' \mid w \in [\prod Y] \} \):

\[
\Gamma \vdash n^{+1} q \iff \text{by Lemma 1(4)} \\
\text{get}(a@l) \in \Gamma \land \Gamma \vdash n q' \land \\
\forall q' \in Q : \; \Gamma \vdash n q' \iff \text{by ind. hyp. and the stop part} \\
\text{get}(a@l) \in \Gamma \land \Gamma \vdash_{\nu} P''^{\uparrow n} \land \\
\forall P'' \in P : \; \vdash_{\nu} P''^{\uparrow n} \iff \text{TIN} \\
\Gamma \vdash_{\nu} P''^{\uparrow n+1}.
\]
Case 5: \( \{v/u\} P_{id} = id'(w) \). Then, assuming \( q' = (id'(w), l) \):

\[
\Gamma \models q 
\iff \text{by Lemma 1(1)}
\]

\[
\Gamma \models q' 
\iff \text{by ind. hyp. and the stop part}
\]

\[
id'(w)^n 
\iff \text{by definition}
\]

\[
\Gamma \models P^\perp_{n+1}
\]

This concludes the proof.

\(\square\)

**Lemma 3.** Let \( q \) be a vertex of \( G_{id} \), and \( \Gamma \) a set of access permissions. Then

\[
\Gamma \models q^H \implies \Gamma \models q.
\]

**Proof.** Suppose that \( \Gamma \not\models q \). Then there is a path \( \pi \) in \( G_{id} \) starting at \( q \) and an access permission \( \alpha \) such that one of the following two cases holds.

Case 1: \( \alpha \notin \Lambda \) and \( \pi \) is an \( \alpha \)-path as in (3) such that \( \alpha \in A_n \) and \( \alpha, \alpha^+, \alpha^- \notin A_i \) for \( i = 0, \ldots, n-1 \). Moreover, we can assume that \( \pi \) is a shortest \( \alpha \)-path of this property.

Then \( n < \frac{4}{H} \) since \( \frac{4}{H} \) is the number of vertices of \( G_{id} \). Indeed, otherwise we could find \( 0 \leq i < j \leq n-1 \) such that \( q_i = q_j \) (assuming that \( q_0 = q \)) and delete the arcs in-between \( q_i \) and \( q_j \), creating a strictly shorter \( \alpha \)-path starting at \( q \). Hence \( \Gamma \not\models q^H \).

Case 2: \( \pi \) is an \( \alpha^- \)-\( \alpha \)-path as in (3) such that \( \alpha \in A_n \) and there is \( m < n \) satisfying \( \alpha^- \in A_m \) and \( \alpha, \alpha^+, \alpha^- \notin A_i \) for \( i = m+1, \ldots, n-1 \). Moreover, we can assume that \( \pi \) is a shortest \( \alpha^- \)-\( \alpha \)-path of this property.

Then, by using twice an argument similar to that used in Case 1, we can show that \( n < 2 \cdot \frac{4}{H} \). Hence \( \Gamma \not\models q^H \).

\(\square\)

**Lemma 4.** If \( \Gamma \models q \) and there is an arrow from \( q \) to \( q' \) labelled with \( A \) then \( \Gamma' \models q' \), where

\[
\Gamma' = \Gamma \setminus \{\alpha \mid \alpha^- \in A\} \cup \{\alpha \mid \alpha^+ \in A\}.
\]

**Proof.** Suppose that \( \Gamma \models q \). We first observe that each path starting at \( q' \) can be prepended by the arc from \( q \) to \( q' \). Hence any \( \alpha^- \)-\( \alpha \)-path starting at \( q' \) would give rise to an \( \alpha^- \)-\( \alpha \)-path starting at \( q \). Consequently, if \( \Gamma' \not\models q' \) then there is an access permission \( \alpha \notin \Gamma' \) and an \( \alpha^- \)\( \alpha \)-path \( \pi \) starting at \( q' \). The latter means \( \alpha^+ \notin A \). We now consider two cases.

Case 1: \( \alpha^- \in A \). Then \( \langle q, q' \rangle \circ \pi \) is an \( \alpha^- \)-\( \alpha \)-path starting at \( q \).

Case 2: \( \alpha^- \notin A \). Then \( \alpha \notin \Gamma \) and \( \langle q, q' \rangle \circ \pi \) is an \( \alpha \)-path starting at \( q \).

\(\square\)

We are now ready to prove the first main result of this paper.

**Theorem 1 (soundness).** If a well-formed network \( N \) has safe access permissions, and \( N' \) is reachable from \( N \), then \( N' \) also has safe access permissions.
Proof. We only need to show the result assuming that \( N' \) is directly reachable from \( N \). Then the result follows from the definition of the operational semantics of networks that each component \( Q' \) of \( N' \) has been derived from some component \( Q \) of \( N \) in one of the following ways (we only discuss some representative cases).

Case 1: \( Q = l \left[ id(v) : \Gamma \right] \) and \( Q' = l \left[ \{v/u\} P_{id} : \Gamma \right] \). Then we consider subcases depending on the form of \( P_{id} \). For example, if \( \{v/u\} P_{id} = go^{\Delta t} l' \) then \( id'(w) \), then we have

\[
\Gamma \vdash_1 id(v) \quad \Rightarrow \text{by TREC}
\]
\[
\Gamma \vdash_1 id(v)^{1H} \quad \Rightarrow \text{by Lemma 2}
\]
\[
\Gamma \vdash^H (id(v), l) \quad \Rightarrow \text{by Lemma 3}
\]
\[
\Gamma \vdash (id(v), l) \quad \Rightarrow \text{by Lemma 4}
\]
\[
apm(l, l')(\Gamma) \vdash (id'(w), l') \quad \Rightarrow
\]
\[
apm(l, l')(\Gamma) \vdash^H (id'(w), l') \quad \Rightarrow \text{by Lemma 2}
\]
\[
apm(l, l')(\Gamma) \vdash l' \quad id'(w) \quad \Rightarrow \text{by TMOVE}
\]
\[
\Gamma \vdash_1 go^{\Delta t} l' \text{ then } id'(w)
\]

Case 2: \( Q = l \left[ go^{\Delta t} l' \text{ then } P : \Gamma \right] \) and \( Q' = l' \left[ P : apm(l, l')(\Gamma) \right] \). Then, by TMOVE, we have:

\[
\Gamma \vdash_1 go^{\Delta t} l' \text{ then } P \Rightarrow apm(l, l')(\Gamma) \vdash P .
\]

Case 3: \( Q = l \left[ go^{\Delta t} l' \text{ then } P : \Gamma \right] \) and \( Q' = l' \left[ go^{\Delta t-1} l' \text{ then } P : \Gamma \right] \). Then \( Q' \) has safe access permissions since \( Q \) has safe access permissions, and timers do not have any impact on having safe access permissions. Similar conclusions apply to the other two cases of decrementing timers.

Our second main result is that in a network with safe access permissions there is no attempt 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 networks with safe access permissions.

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

\[
l \left[ a^{\Delta t} ! (v) \text{ then } P \text{ else } Q : \Gamma \right] \in \text{comp}(N) \quad \Rightarrow \quad \text{put}(a@l) \in \Gamma
\]
\[
l \left[ a^{\Delta t} ? (u:X) \text{ then } P \text{ else } Q : \Gamma \right] \in \text{comp}(N) \quad \Rightarrow \quad \text{get}(a@l) \in \Gamma .
\]

Proof. The result follows from the fact that the final stages of the derivation of the judgments in these cases consist of applications of the corresponding rules (TOUT) and (TIN), possibly followed by some applications of the rule (TSUB) (which can only add access permissions and never remove any).
Our third main result is that the notion of being a network with safe access permissions is complete in the sense that a network without safe access permissions can always be placed in an environment which reveals its potential to breach safety of interprocess communication.

**Theorem 3 (soundness).** Let \( N = l \parallel P : \Gamma \) be a well-formed network such that \( \Gamma \vdash l \parallel P \). Then there is a well-formed network \( N' \) with safe access permissions and a well-formed network \( N'' \) reachable from \( N \mid N' \) such that one of the following holds:

- There is at least one component \( l' \parallel [a^{\Delta}!\langle v \rangle \text{ then } P' \text{ else } P'': \Gamma' \] of \( N'' \) such that \( \text{put}(a@l') \notin \Gamma' \).
- There is at least one component \( l' \parallel [a^{\Delta}?(u:X) \text{ then } P' \text{ else } P'': \Gamma' \] of \( N'' \) such that \( \text{get}(a@l') \notin \Gamma' \).

**Proof (Sketch).** We can assume that \( P = id(v) \). Since \( \Gamma \vdash l \parallel P \), there is an \( \alpha \)--\( \alpha \) path originating at \( q = (id(v), l) \), or an \( \alpha \)--\( \alpha \) path originating at \( q \) and \( \alpha \notin \Gamma \). We can then simulate the execution of \( N \) following the path leading to a problem. All we need to do is provide the necessary processes communicating with \( P \) waiting at appropriate locations.

We can then conclude that in this paper we developed a sound and complete system for ensuring safety of communication for networks of migrating processes.

**Remark 2.** We assumed above a simple form of recursive definitions. However, this does not diminish the generality of the proposed method since we can always proceed as follows. First, we transform all recursive definition into the simple form using additional process identifiers and recursive definitions. Then we apply our procedures to check the safe access permissions for the modified recursive definitions. If the answer is ‘no’, we know that the network with the original recursive definitions fails to satisfy Theorem 3 as the transformation does not affect this kind of result. Similarly, the answer ‘yes’ implies that the original system satisfies Theorem 3. As a result, all three main results can be recovered in the general case as well.

It is worth noting that the transformation we just described does affect the operational semantics, but in a way which is harmless from the point of view of checking safe access permissions.

\[
\frac{\Gamma \neq \mathbf{vt} \Rightarrow \text{put}(a@l) \in \Gamma \quad \Gamma' \neq \mathbf{vt} \Rightarrow \text{get}(a@l) \in \Gamma' \quad v \in \llbracket X \rrbracket}{\text{(Com')} \quad l[a^{\Delta}!\langle v \rangle \text{ then } P' \text{ else } Q : \Gamma \mid a^{\Delta}?(u:X) \text{ then } P' \text{ else } Q' : \Gamma'] \\
\xrightarrow{a(u)@l} l\llbracket P : \Gamma \mid \oplus\{u/u\}P' : \Gamma' \rrbracket}
\]

**Table 8.** Modified rules of the operational semantics.
**Simplifying the Operational Semantics** 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. More precisely, if we consider a network with safe access permissions, we may delete in rule (Com) the preconditions
\[
\text{put}(a@l) \in \Gamma \quad \text{and} \quad \text{get}(a@l) \in \Gamma',
\]
and execute the network according to the rules in Tables 4 with and without the modified rule (Com). Then we obtain equivalent derivation systems. In other words, for networks with safe access permissions one can separate all concerns related to access permissions from other behavioural aspects (for example, deadlock analysis or redundant code detection), allowing much more efficient property verification.

Following further this idea and making it more precise, we can consider networks in which only some component networks have safe access permissions. For such components, we can ignore all the aspects relating to access permissions, but keep them unchanged for those component networks which are not have safe access permissions. In more concrete terms, we can introduce a special dummy access permission set \(\text{wt}\) to indicate that a judgement of the form \(\Gamma \vdash P\) can be established, and use \(l[P : \text{wt}]\) in the initial description of the system instead of \(l[P : \Gamma]\). After that we need to make some adjustments to the rules of operational semantics, by re-defining the (Com) rule, as shown in Table 8. Moreover, \(apm(l, l')(\text{wt}) = \text{wt}\). We then obtain the following result. If we consider a network and replace all components with safe access permissions \(l[P : \Gamma]\) by \(l[P : \text{wt}]\) and run the resulting network according to the rules in Tables 4 and 8, then we obtain derivation systems which is strongly bisimilar [17] to that of the original network. In the extreme case, when all component networks have safe access permissions, we can simply forget about access permissions when analyzing other aspects of network behaviour. What is crucial is that such modifications can lead to great reductions in the size of the state space of the resulting system.

**Running Example** Below we show an amended version of the network modelling our running example after taking into account that some of the processes have safe access permissions, and dropping the \(\text{wt}\) symbols which are implicit.

\[
\text{TravelShop'} \overset{\text{def}}{=} \\
\text{home} \parallel \text{client}(130):\emptyset \mid \text{travelshop} \parallel \text{agent}(100) \mid \text{update}(60) \mid \text{standard} \parallel \text{flightinfo}(110, \text{special}) \mid \text{special} \parallel \text{saleinfo}(90, \text{bank}) \mid \text{bank} \parallel \text{till}(10)
\]
For example, we observe that $\emptyset \vdash_{\text{travelshop}} \text{update}(60)$ since we have the following derivation:

$\emptyset \vdash_{\text{special}} \text{stop}$ by (TSTOP)

$\{\text{put}(\text{info}@\text{special})\} \vdash_{\text{special}} \text{stop}$ by (TSTOP)

$\{\text{put}(\text{info}@\text{special})\} \vdash_{\text{special}} \text{info}!\langle 60 \rangle \rightarrow \text{stop}$ by (TOUT)

$\emptyset \vdash_{\text{travelshop}} \text{go}^{\Delta^0} \text{special} \rightarrow \text{info}!\langle 60 \rangle \rightarrow \text{stop}$ by (TMOVE)

Note that $\text{info}!\langle 60 \rangle \rightarrow \text{stop}$ in the third line above is in fact

$$\text{info}^{\Delta^\infty}!\langle 60 \rangle \text{ then stop else stop}.$$  

One may observe that $\emptyset \not\vdash_{\text{home}} \text{client}(130)$ since this process is of the form

$$\text{go}^{\Delta^5} \text{travelshop} \rightarrow \text{flight} ? (\text{standardoffer}: \text{Loc}) \rightarrow$$

$$\text{go}^{\Delta^4} \text{standardoffer} \rightarrow \text{info} ? (p1: \text{eMoney}, \text{specialoffer}: \text{Loc}) \rightarrow P$$

and so we must have

$$\{\text{get}(\text{flight}@\text{travelshop})\} \vdash_{\text{travelshop}}$$

$$\text{flight} ? (\text{standardoffer}: \text{Loc}) \rightarrow$$

$$\text{go}^{\Delta^4} \text{standardoffer} \rightarrow \text{info} ? (p1: \text{eMoney}, \text{specialoffer}: \text{Loc}) \rightarrow P$$

and so we must have

$$\{\text{get}(\text{flight}@\text{travelshop})\} \vdash_{\text{travelshop}}$$

$$\text{go}^{\Delta^4} \text{bank} \rightarrow \text{info} ? (p1: \text{eMoney}, \text{specialoffer}: \text{Loc}) \rightarrow P$$

and so we must have

$$\{\text{get}(\text{flight}@\text{travelshop})\} \vdash_{\text{bank}} \text{info} ? (p1: \text{eMoney}, \text{specialoffer}: \text{Loc}) \rightarrow P$$

This means that we cannot proceed any further since

$$\text{get}(\text{info}@\text{bank}) \notin \{\text{get}(\text{flight}@\text{travelshop})\}.$$  

Note however, that

$$\{\text{get}(\text{info}@\text{standard}), \text{get}(\text{info}@\text{special}), \text{put}(\text{pay}@\text{bank})\} \vdash_{\text{home}} \text{client}(130)$$

### 4 Coordination via Timers

One of the primary applications of timers is the coordination of the migration of processes. For instance, we may change the definition of a client process so that it captures urgency in the wish to complete the transaction, by insisting that the second offer is obtained immediately (the urgency is expressed by the
timer $\Delta(0)$; if this does not happen, the outcome of the visit to the travel shop is decided only on the basis of the first offer:

\[
\text{client'(init:eMoney)} \overset{a}{=} \\
\text{go}^{\Delta_5} \text{ travelshop} \rightarrow \text{flight} ? (\text{standardoffer:Loc}) \rightarrow \\
\text{go}^{\Delta_4} \text{ standardoffer} \rightarrow \text{info} ? (p1:eMoney, \text{specialoffer:Loc}) \rightarrow \\
\text{go}^{\Delta_3} \text{ specialoffer then} \rightarrow \text{info}^{\Delta_0} ? (p2:eMoney, \text{paying:Loc}) \text{ then }

\text{go}^{\Delta_0} \text{ paying} \rightarrow \text{pay} !(\min\{p1, p2\}) \rightarrow \\
\text{go}^{\Delta_0} \text{ home} \rightarrow \text{client}(\text{init} - \min\{p1, p2\})
\]

\text{else}

\text{go}^{\Delta_0} \text{ bank} \rightarrow \text{pay} !(p1) \rightarrow \\
\text{go}^{\Delta_0} \text{ home} \rightarrow \text{client}(\text{init} - p1)

5 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 [16] 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 K\textsc{aim} and its extensions, using type systems that enable the dynamic exchange of access rights. The paper [9] combines a weak form of information flow control with typed cryptographic operations to ensure safe static access control and secure network communications. The paper [6] 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 true 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 [20]. On the other hand, several prototype languages have been designed and experimental implementations derived from process calculi like K\textsc{aim} [5] and A\textsc{cute} [19]. These
prototype languages did not become a practical programming language because hard questions revolving mainly around security issues. 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 [12] and TiMo [10]. PERTIMo derives from TiMo model (a simplified distributed π-calculus with explicit timeouts) presented in [10] 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 [15] by using a type system. The paper [4] studies a π-calculus extension with a timer construct, and then enriches the timed π with locations. Other timed extensions of process algebras have been studied in [3, 13]. In [8] 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, [2] 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