A Software Platform for Timed Mobility and Timed Interaction

Gabriel Ciobanu and Călin Juravle

Institute of Computer Science, Romanian Academy, Iaşi
"A.I.Cuza" University of Iaşi, Romania
gabriel@iit.tuiasi.ro, calin.juravle@info.uaic.ro

Abstract. TiMo is a process algebra using timeouts for interactions and adaptable migration between explicit locations. Starting from this formalism, we have implemented a software platform for agent migration, separating the migration mechanism such that it can be reused for other systems with mobility. We describe the platform architecture and functionalities, the software modules and some implementation details, emphasizing the novel aspects and comparing with similar implementations. The implementation corresponds rigorously to the semantics of TiMo. An example illustrates the use of the migration platform for a simple problem.

1 Introduction

Mobile applications represents an important topic in distributed system field. Mobility is difficult in both the modeling part and in implementation, especially when time is also considered. To address the modeling part, many formalisms have been proposed over the years such as π-calculus [8], distributed π-calculus [6], timed distributed π-calculus [4], TiMo [3]. Concerning the implementation, there are several architectures and different programming languages (Telescript [11], Java) which support or facilitate code mobility or mobile agents programming. Although there are several papers on both aspects (theoretical and practical) addressing mobility, the link between the theoretical specification and effective implementation is not clearly defined.

Our aim is to provide a platform for agent migration which corresponds to a formal model. Starting from TiMo, we implement such a platform. To ensure that it corresponds with the high-level operational semantics of TiMo, we define a formal notion of configuration and use it to describe and reason about the evolution of a system.

Since mobility is the main concept, we separate the low-level mobility concerns and the high-level model aspects into two layers. Thus, our implementation consists of an extensible basic framework which can be used to implement various systems based on mobility and a framework inspired by TiMo which facilitates to specify mobile agents. The lower layer is named *MobileCalculi framework* and, besides a migration mechanism, it offers generic implementations of common concepts needed to implement mobile systems. The upper layer is inspired by TiMo and it is referred as the *software framework for TiMo*; it also provides a compiler for an intermediate language in which someone can specify systems of mobile agents. In order to prove the extensibility of *MobileCalculi* framework we also implemented a software framework for dπ-calculus [6]. The whole system is named *MCTools*. Thus, *MCTools* represents a software platform for mobile calculi implementation.

MCTools system is developed according to a choreography based distributed architecture. It is working without a central coordinator. Agents are free to roam and travel in a network of machines which have the system installed, without being orchestrated by a central entity.

The paper is structured as follows. We first briefly present the TiMo model in Section 2, then we describe some implementation details in Sections 3 and 4. We present the correspondence between TiMo and the implementation in Section 5. Before ending with conclusion and related works, an illustrating example is presented in Section 6.

2 TiMo

TiMo [3] is a simple process algebra in which one can formally model distributed systems with explicit locations, migration and temporal constraints. It is a part of the π-calculus family [8], close to the distributed π-calculus [6] and timed distributed π-calculus [4]. TiMo features a simple syntax, dropping the type aspects of distributed π-calculus and focusing on interaction and migration. The time is local and modeled by timers which are associated with basic actions. The result is that the interaction and migration time is no longer indefinite. Moreover, if an action does not happen in a predefined time, then the process continues with a "safety" alternative.

TiMo Syntax is given below. It is assumed that *Chan* is a set of channels, *Loc* is a set of locations, *Var* is a set of location variables and *Ident* is a finite set of process identifiers (each identifier $I \in Ident$ has a fixed arity $m_I \geq 0$).

$$\begin{aligned} P, Q ::= {} & 0 \mid a^{\Delta t}\,!\,\langle v\rangle \,\texttt{then}\, P \,\texttt{else}\, Q \mid a^{\Delta t}\,?\,(u) \,\texttt{then}\, P \,\texttt{else}\, Q \mid \\ & \mathbf{go}^{lt\Delta mt}\, v \,\texttt{then}\, P \,\texttt{else}\, Q \mid I(v_1,\ldots,v_{m_I}) \mid P \mid Q \mid \#P \\ M, N ::= {} & k[\![P]\!] \mid M \mid N \end{aligned}$$

In the above description it is assumed that $a \in Chan$; $t, lt, mt \in \mathbb{N}$,; $v, v_1, \ldots, v_{m_I} \in Loc \cup Var$; $k \in Loc$ and $u \in Var$. Moreover, each process identifier $I \in Ident$ has a unique definition of form $I(u_1,\ldots,u_{m_I}) = P_I$ where $u_i \neq u_j$ (for $i \neq j$) are variable acting here as parameters.

Process $a^{\Delta t}\,!\,\langle v\rangle \,\texttt{then}\, P \,\texttt{else}\, Q$ attempts to send v over channel a for t units of time. If the communication takes place then it continues as P, otherwise it continues as Q. Input process $a^{\Delta t}\,?\,\langle u\rangle \,\texttt{then}\, P \,\texttt{else}\, Q$ has a similar behaviour. Process $\mathbf{go}^{lt\Delta mt}\, v \,\texttt{then}\, P \,\texttt{else}\, Q$ implements mobility. It first waits lt units of time which represents the *local time* dedicated to local work, then it moves to location v in mt units of time (mt stands for *migration time*). If the move is accomplished within the specified time, then the process behaves as P (at v), otherwise it continues as Q at current location. Processes are further constructed from the basic processes together with the terminating process 0 by using the parallel composition $P \mid Q$. A located process $k[\![P]\!]$ is a process running at location k. The symbol # from $\#P$ is a purely technical notation used in the formalization of structural operational semantics of TiMo. Intuitively, it says that the process has finished its action and it is temporally waiting for the next tick of the clock.

Operational Semantics of TiMo is given by the rules presented in Table 1.

GO: $$k[\![\mathbf{go}^{0\Delta mt}\, l\, \texttt{then}\, P\, \texttt{else}\, Q\,]\!] \xrightarrow{k:l} l[\![\#P]\!]$$

COM: $$l[\![a^{\Delta t}\,\mathbf{!}\,\langle l\rangle\, \texttt{then}\, P\, \texttt{else}\, Q \mid a^{\Delta t'}\,\mathbf{?}\,(u)\, \texttt{then}\, P'\, \texttt{else}\, Q'\,]\!] \xrightarrow{k:a(l)} l[\![\#P \mid \#\{l/u\}P']\!]$$

PAR: $$\frac{N \xrightarrow{\beta} N'}{N \mid M \xrightarrow{\beta} N' \mid M}$$

STRUC: $$\frac{N \equiv N' \quad N \xrightarrow{\beta} M \quad M \equiv M'}{N' \xrightarrow{\beta} M'}$$

TIME: $$\frac{N \not\rightarrow}{N \xrightarrow{\surd} \phi(N)}$$

Table 1. TiMo operational semantics

Looking to the labels of the transitions, there are two kinds of transition rules: $M \xrightarrow{\beta} N$ and $M \xrightarrow{\surd} N$. The first one corresponds to the execution of an action β, while the second one represents a timing tick. The action β can be either $k : l$ or $k : a(l)$, where k is the location where the action takes place, l is either the location where the process goes, or the location transmitted along the channel a. In rule Time, $N \not\rightarrow$ denotes that no other rule can be applied.

ϕ is the time-passing function which acts in the following way. Each top-level expression $I(l_1, \ldots, l_{m_I})$ is replaced by the corresponding definition $\{l_1/u_1, \ldots, l_{m_I}/u_{m_I}\}P_I$. Each top-level expression of the form $a^{\Delta 0}\ldots\texttt{then}\, P\, \texttt{else}\ Q$ or $\mathbf{go}^{0\Delta 0}\ldots\texttt{then}\ P\, \texttt{else}\, Q$ is replaced by $\#Q$. For the top-level communication expressions with $\Delta t > 0$, t is decreased by 1. Each top-level expression of the form $\mathbf{go}^{lt\Delta mt}\, \texttt{then}\, P\, \texttt{else}\ Q$ is replaced by $\mathbf{go}^{lt'\Delta(mt')}\, \texttt{then}\, P\, \texttt{else}\ Q$ where $lt' = \max\{0, lt-1\}$ and $mt' = mt$ if $lt > 0$ or $mt' = max\{0, mt-1\}$ if $lt = 0$. All occurrences of the special symbol # are deleted.

A top-level expression is not containing a symbol #. Note that only after the lt timer reaches 0, the process migrates to the destination.

3 MobileCalculi Framework

As stated in the introduction *MobileCalculi* framework represents the lower level of the *MCTools* system. Its purpose is to be the low-level link between the theoretical part of mobility (represented by various formalisms such as those from π-calculus family) and the practical part which deals with mobile code and mobile systems implementations. It was designed to abstract the concepts used in the formal models, and to handle low-level details such as network or location management. The correspondence between location names and physical locations, represented by an IP address plus a port, is done at this level.

The framework serves as a base for implementation of models for mobility, dealing with the common part of such formalisms: names, locations, agents, migration, fresh name generation, etc. It provides a default mechanism for migration which makes possible to migrate an agent by its code and data. Moreover, it provides the architecture of an engine for simulation of the formal evolution of a process. It also handles communications with other machines, and thus it can create and initialize a distributed environment from a global specification, making it a useful tool for distributed experiment. The global specification of a system is represented by the agent distribution at their initial locations.

The framework is based on an extensible architecture so that the majority of components can be customized according to needs. It is implemented in Java language, the main reason being the infrastructure offered by Java for working with mobile code and dynamic classes. A formalism is implemented by extending structures from the framework and adapting them to its specific features. The software framework for TiMo serves as an example, but we can also use other formalism implementation. To prove the extensibility of the *MobileCalculi* framework we also implemented a software framework for dπ-calculus.

We developed a generic purpose GUI in order to ease the user interaction with *MCTools* platform, in particular with the *MobileCalculi* framework. Using the GUI one can easily access the majority of framework functionalities without any coding. It is possible to start or stop the system, change the active formalism (the upper layer), compile, load and execute specific formalism specifications and interact with other *MCTools* platforms.

We describe the implementation from a functional viewpoint. A global view of the platform architecture can be seen in Figure 1, where it is presented the interaction between the two layers, the lower layer represented by *MobileCalculi* framework, and the upper layer represented by a formalism framework (in particular TiMo framework). It also present the dependency inside the layers.

The functionality of *MobileCalculi* framework is divided into several modules. The most important ones are the *core* module which deals with common functionalities and general patterns, and the *mobility* module which encapsulates the mobility mechanism. These modules are presented below. To keep the presentation clear and simple the rest of the modules are omitted.

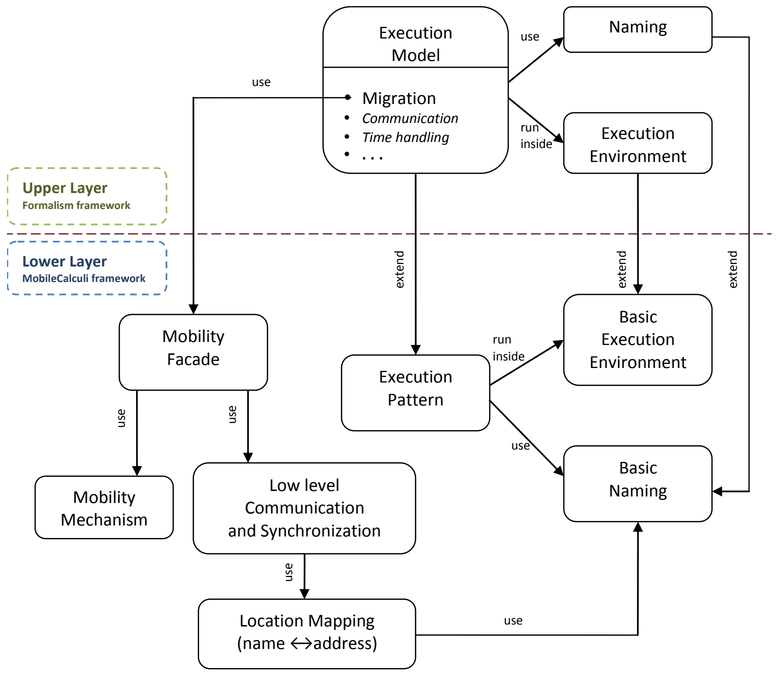


Fig. 1. MCTools Functional Architecture

Core Module The *core* module is the heart of the *MobileCalculi* framework. It contains the main functionalities and propose the patterns which must be followed by a formalism implementation. The implementation of a formalism either extends these patterns and enhances agent execution with specific features, or just uses some of the basic functionalities.

The main entities in a formalism for mobility are agents, locations and names. An agent is represented by an object which contains a main method with its actions/instructions. This representation defines an execution pattern by assuming that agent execution is equivalent with its main method execution. The agent runs at a specific location, in a private thread. The location acts as an execution environment for agents. It keeps a list with resources, such as communication channels, which can be used by agents. All the entities (including resources) are referred by name, so the name concept is also defined as a separate entity.

A formalism implementation must provide at least an *execution model* and an *execution environment*. The *execution model* is defined by the formalism primitives (such as migration, communication) which have to be described according to the formal specification. One must focus only on these primitives since the basic ones such as starting or stooping the execution, joining with other execution threads or spawning addition workers are implemented in the default pattern. This model runs inside a specific *execution environment* and uses a naming structure. For example the execution environment for TiMo defines a virtual clock and the agents defined in TiMo are governed by this clock. Again, some basic functionalities of an execution environment are implemented at the framework low-level (adding or removing new agents, generating unique names).

Since the framework is built for mobility formalism, it assumes that migration primitives are present in every formalism implementation. Considering this, the *core* module facilitate the use of the mobility mechanism presented in Section 3, by managing the low level details and acting as a mobility façade.

It is worth noting that this module also incorporates many other functionalities which are transparent to the developer of a certain formalism implementation. It handles communication with other machine, not just for transmitting agents, but also for synchronization and control. It manages the execution environment, and it sets up a distributing environment from a global specification. Moreover, it manages the several formalisms providing a way to switch between them dynamically; this enable the possibility to change the execution model (in other words the upper layer of *MCTools* system) without shutting down the platform and independently of other platforms.

Another important functionality which can be use directly by the upper layer is the formal evolution engine. Given a formal specification, this engine enables to execute locally the evolution of a formal specification corresponding to the formalism semantic rules. Using this feature one can detect possible discrepancies between formal specifications and their implementations.

Mobility Module This module creates the needed infrastructure for agent migration. It also provides a default migration mechanism based on bytecode migration. This module abstracts the migration objects by providing an interface which must be implemented by all the entities which want to migrate. This maintains a decoupled architecture and makes it possible to easily change the migration mechanism. The main feature of this module is the proposed migration architecture.

Among the possible alternatives we consider the solution based on bytecode migration. It ensure the dependencies migration by using special class loaders. The main idea is to retain the bytecode of agents in a local repository. In order to access a class bytecode, the class must be loaded with a special class loader which saves the bytecode at loading. At migration the agent definition and dependencies are searched in this repository. The definitions of agent and its dependencies are stored at destination in a similar repository from where they can be loaded. After loading the agent, data can be recovered.

Note that we preferred to migrate the dependencies together with the agent rather than implementing a lazy mechanism. The motivation behind this is given by the fact that after a valid migration the agent should work correctly independent of other locations. In a lazy situation it is possible that the location containing the dependencies does not work when the agent needs a certain dependency. Thus, bringing the dependency in a lazy way determines the failure of the agent. Having all the dependencies transported together with the agent, we avoid this scenario and let the agent to execute independently of previous locations. Our choice has also the advantage of simplifying the handling of disconnected operations (the agent can execute even if the owner is not connected).

4 Software Framework for TiMo

The main features of TiMo implementation are:

– creating and executing TiMo agents in a distributed environment;
– the possibility of introducing native Java code into the agents body;
– an intermediate language *TLang* and a (typed) compiler which can generate the Java code from a simple syntax;
– an operational correspondence between implementation and its formal model.

In order to help writing the TiMo agents, we develop a language called *TLang* to intermediate between the high-level TiMo and the low-level Java code. *TLang* uses only a limited set of Java types and instructions. Even if TLang does not include all Java functionalities, it provides several important advantages like type checking, syntax for agents distribution, possibility to embed Java code, mechanisms for simulation of strong mobility. We also create a compiler which translates TiMo agents (written in a simple syntax) into the appropriate Java code. The compiler also builds the objects necessary to run the agents in a distributed environment using *MobileCalculi* framework.

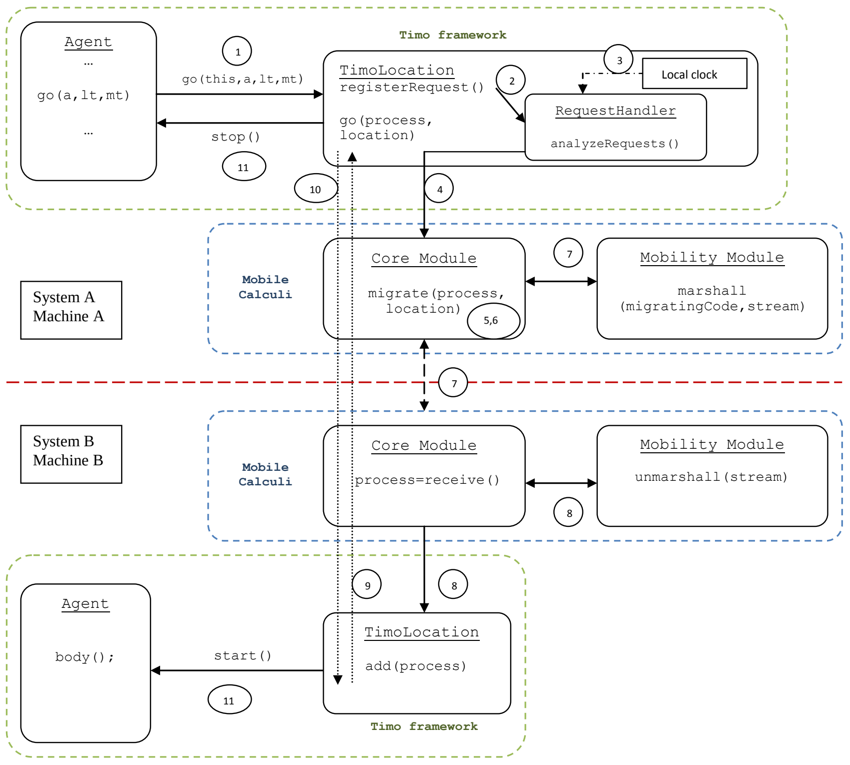


Fig. 2. Migration of a TiMo agent

Before presenting the implementation, we analyze some constraints imposed by the transition from theory to practice. We allow other values than locations to be transmitted on channels. Only allowing locations to be transmitted, our implementation would serve mainly for theoretical simulation and not for practical use. The communication values can be of any Java type if the agent is written directly in Java, and of some restricted type if it is developed with *TLang*. Communication on channel is well typed. This means that a channel has an associated type. For instance, if an agent wants to sent or receive a *location* on a channel dedicated to *strings*, an error appears (more exactly, a compile error if the compiler is used, or a runtime error if Java is used). The safety process is activated either when time expires for a communication (as in the formalism), or when an exception is thrown out and the agent is about to fail.

More implementation details are presented in [2]. Here we briefly summarize TiMo primitives and temporal aspects. The temporal aspects are implemented with the help of a *virtual clock*. The clock is local to each location, and so has a predefined frequency. At each tick it triggers an event, and the subscribers take the appropriate actions. One subscriber is the *TimoLocation* which analyzes at every tick the requests it has received.

Migration uses the infrastructures provided by the *MobileCalculi* framework. Since the framework implements a "weak" mobility mechanism, it falls to the programmer to retain the program counter and manage the point from which the execution of agents is restarted at destination. The semantic of migration timers is implemented by using a distributed protocol. The local timer *lt* is represented as the waiting time before the migration. It is the first one which is decremented. The migration timer *mt* is implemented as a distributed protocol. After the local time reaches 0, the migration procedure is initiated and the migration timer starts to be decremented. After the agent arrives at its destination, a *receive message* is sent back. If the message is receive before the migration timer becomes 0 the agent is remove from its initial location and another message, a *confirmation message* is sent to destination; otherwise the agent activates its safety process at the current location. At destination the agent restarts only after receiving the *confirmation message*. The default behaviour if this message is not received is to remove the agent (at destination). A successful migration can be visualized in Figure 2.

Communication between agents respects faithfully the TiMo definition, and it is based on the *rendez-vous* mechanism [7].

5 Implementation Soundness

In this section we show that our implementation corresponds with TiMo high-level semantics. We first define an abstract notion of *configuration* and then use it to reason about the implementation soundness with respect to TiMo semantics.

Definition 1. *Given a process R specified in TiMo, we define the process stack $S(R)$, or simply S, as in Table 2.*

Remark 1. This definition is consistent with both the theoretical view which presents the process as a sequence of actions, and the practical view where

R	$\mapsto$	S(R)
$\mathbf{go}^{lt\Delta mt}\, l\, \texttt{then}\, P\, \texttt{else}\, Q$	$\mapsto$	$\begin{array}{\|c\|}\hline (go,\,(lt,\,mt),\,l) \\ \hline P \\ \hline Q \\ \hline \end{array}$
$a^{\Delta t}\, \textbf{?}\, (v)\, \texttt{then}\, P\, \texttt{else}\, Q$	$\mapsto$	$\begin{array}{\|c\|}\hline (in\, c,\, t,\, v) \\ \hline P \\ \hline Q \\ \hline \end{array}$
$a^{\Delta t}\, \textbf{!}\, \langle u\rangle\, \texttt{then}\, P\, \texttt{else}\, Q$	$\mapsto$	$\begin{array}{\|c\|}\hline (out\, c,\, t,\, u) \\ \hline P \\ \hline Q \\ \hline \end{array}$
$P \mid Q$	$\mapsto$	S(P) and S(Q) distinct stacks

Table 2. Process Stack Definition

each process has a stack from where the next action is executed. Moreover, it is consistent with the software framework for TiMo. In implementation, only the primitives of communication and migration are considered when the virtual clock ticks. All the other actions are internal. Thus, from a temporal point of view, we can abstract the process as being composed only from primitives of communication and migration presented as a stack.

The configuration of a location is represented by the set of stacks $S_1^1,\, S_2^2, ..., S_n^n$ of the processes which run at that location l, and it is written as $l[S_1^1,\, S_2^2, ...,\, S_n^n]$. The configuration of a distributed system is a network of location configurations where each node contains the set of stacks corresponding to the local processes. We denote by $l_1[S_1^1, ..., S_{n_1}^1] \times ... \times l_n[S_1^n, ..., S_{n_n}^n]$ a network with n locations, where for each location l_i the number of processes is provided by n_i.

We denote by $Config$ the set of all possible configurations, and usually refer to a configuration only thinking to the top of its stacks. We write 0 for the empty stack corresponding to a terminated process. When it is not explicitly specified, by configuration we understand the configuration of a system.

Definition 2. *Over the configuration set, we define the transition function* $\delta : Config \times CT \to Config$*, where* $CT = \{tick,\, subst,\, go,\, fail_{com},\, fail_{go}\}$*.*

In the following we write $\delta(c,\, ctype) = c'$ *as* $c \xrightarrow{ctype} c'$*.*

- $l[S_1, ..., \#(chAct\, a,\, t,\, x), ...,\, S_n] \xrightarrow{tick} l[S_1, ..., (chAct\, a,\, t-1,\, x), ...,\, S_n]$ *where* $chAct \in \{in,\, out\}$ *and* $x \in Val \cup Var$*.*
- $l[S_1, ..., \#(go,\, (lt,\, mt),\, l), ...,\, S_n] \xrightarrow{tick} l[S_1, ..., (go,\, (lt-1,\, mt),\, l), ...,\, S_n]$ *provided that* $lt > 0$ *and* $l \in Loc$*.*
- $l[S_1, ..., \#(go,\, (0,\, mt),\, l), ...,\, S_n] \xrightarrow{tick} l[S_1, ..., (go,\, (0,\, mt),\, l), ...,\, S_n]$ *provided that* $mt \geq 0$ *and* $l \in Loc$*.*
- $l[S_1, ..., (in\, a,\, t,\, v), ..., (out\, a,\, t',\, u), ...,\, S_n] \xrightarrow{subst} l[S_1, ..., \#S(\{u/v\}P), ..., \#S(P'), ...,\, S_n]$ *provided that* $min(t,\, t') \geq 0$*, the stack of in action is* $[(in\, a,\, t,\, v),\, P,\, Q]$ *and that of out action is* $[(out\, a,\, t',\, u),\, P',\, Q']$*.*

- $l[S_1, ..., (in\, a,\ t,\ x), ...,\ S_n] \xrightarrow{fail_{com}} l[S_1, ...,\ \#S(Q), ...,\ S_n]$
  provided that $t < 0$ *and the stack of in action is* $[(in\, a,\ t,\ v),\ P\ ,\ Q]$.
- $l[S_1, ...,\ (out\, a,\ t',\ u), ...,\ S_n] \xrightarrow{fail_{com}} l[S_1, ...,\ \#S(Q), ...,\ S_n]$
  provided that $t < 0$ *and the stack of out action is* $[(out\, a,\ t,\ u),\ P,\ Q]$.
- $l[S^l_1, ...,\ (go, (lt, mt),\ k), ...,\ S^l_{nl}] \times k[S^k_1, ...S^k_{nk}] \xrightarrow{go}$
  $l[S^l_1, ..., 0, ...,\ S^l_{nl}] \times k[S^k_1, ...,\ S^k_{nk},\ \#S(P)]$
  provided that $lt = 0$, $mt = 0$ *and the stack of go at* l *is* $[(go,\ (lt,\ mt),\ l),\ P,\ Q]$.
- $l[S^l_1, ...,\ (go, (lt, mt),\ k), ...,\ S^l_{nl}] \xrightarrow{fail_{go}} l[S^l_1, ..., \#S(Q), ...,\ S^l_{nl}]$
  provided that $lt = 0$, $mt = 0$, *location* k *is unreachable and the stack of go at* l *is* $[(go,\ (lt,\ mt),\ l),\ P,\ Q]$.
- *if none of the above rules can be applied, we apply one of the following rules:*
  - $l[S_1, ..., (chAct\, a,\ t,\ x), ...,\ S_n] \xrightarrow{tick} l[S_1, ..., (chAct\, a,\ t-1,\ x), ...,\ S_n]$
    where $chAct \in \{in,\ out\}$ *and* $x \in Val \cup Var$.
  - $l[S_1, ..., (go,\ (lt,\ mt),\ l), ...,\ S_n] \xrightarrow{tick} l[S_1, ..., (go,\ (lt-1,\ mt),\ l), ...,\ S_n]$
    provided that $lt > 0$ *and* $l \in Loc$.
  - $l[S_1, ..., (go,\ (0,\ mt),\ l), ...,\ S_n] \xrightarrow{tick} l[S_1, ..., (go,\ (0,\ mt),\ l), ...,\ S_n]$
    provided that $mt \geq 0$ *and* $l \in Loc$.

Note that the rules are maximally applied for all possible stacks of a configuration.

Proposition 1. *The implementation of migration and communication primitive corresponds operationally to the rules* GO *and* COM *of the* TIMO *formalism.*

Proof. We prove this by showing that for each process R and for each possible evolution rule of type GO or COM which takes R into R', there exists a sequence of transitions which takes the configuration corresponding to R into a configuration which corresponds to R'. This is summarized in the following diagram, where $\beta = k : l$ or $k : a(l)$.

$$\begin{tikzcd} R \arrow[r, "\beta"] \arrow[d] & R' \arrow[d] \\ config \arrow[r, "*"] & config' \end{tikzcd}$$

There are several cases which must be analyzed including success actions, failed actions, and actions that come right next after a blocking.

- We first consider the communication case: $k[\![a^{\Delta t}\,\mathbf{?}\,(v)\,\mathtt{then}\,P\,\mathtt{else}\,Q \mid a^{\Delta t'}\,\mathbf{!}\,(u)\,\mathtt{then}\ P'\,\mathtt{else}\,Q']\!] \xrightarrow{k:a(u)} k[\![\#\{u/v\}P \mid\ \#P']\!]$. The configuration $k[(in\, a,\ t,\ v),\ (out\, a,\ t',\ u)]$ corresponds to process R. We apply *subst* rule: $k[(in\, a,\ t,\ v),\ (out\, a,\ t',\ u)] \xrightarrow{subst} k[\#S(P),\ \#S(P')]$. It is easy to see that the resulting configuration corresponds to the process R'.
- The migration case is as follows: $k[\![\mathbf{go}^{t,\,\Delta t'} m\,\mathtt{then}\,P\,\mathtt{else}\ Q\,]\!] \xrightarrow{k:l} m[\![\#P]\!]$. The corresponding configuration of the left-hand side is $k[(go,\ (t,\ t'),\ m)]$. We apply *go* rule and get $k[(go,\ (t,\ t'),\ m)] \times m[0] \xrightarrow{go} k[0] \times m[\#P]$. The resulting configuration is the corresponding one for the right-hand side which proves this case.

– The failure cases are similarly treated, and not presented here.
– When an action comes after a #, we add an extra *tick* transition in order to keep the consistency between processes and configurations. Suppose that we have the following case: $k[\![\mathbf{go}^{t,\,\Delta t'} m\,\texttt{then}\,P\,\texttt{else}\;Q\,]\!] \xrightarrow{k:m} m[\![\#P]\!]$ with $P = a^{\Delta t}\mathbf{?}(v)\,\texttt{then}\,P_1\,\texttt{else}\;Q_1\;\;|\,a^{\Delta t'}\mathbf{!}(u)\,\texttt{then}\,P_1'\,\texttt{else}\;Q_1'$. The evolution continues with the *tick* rule: $m[\![\#P]\!] \xrightarrow{tick} m[\![a^{\Delta t-1}\mathbf{?}(v)\,\texttt{then}\,P_1\texttt{else}\,Q_1\,|\,a^{\Delta t'-1}\mathbf{!}(u)\,\texttt{then}\,P_1'\,\texttt{else}\;\;Q_1'\,]\!] \xrightarrow{m:a(u)} m[\![\#P_1\,|\,\#P_1']\!]$. The corresponding configuration transitions are: $k[(go,\,(t,\,t'),\,m)] \times m[0] \xrightarrow{go} k[0] \times m[\#S(P)]$. Expanding $S(P)$ we get: $m[\#S(P)] = m[\#(in\,a,\;t,\,v),\#(out\;\;a,\,t',\,u] \xrightarrow{subst} m[\#S(P_1),\,\#S(P_1')]$ which corresponds to the resulting process.
– The other cases are similarly treated, and not presented here.

Remark 2. Each syntactic structure from TiMo can be represented in *TLang* (the compiler language) which then can be translated into a Java implementation.

$\mathbf{go}^{lt\Delta mt}\,l\,\texttt{then}\,P\,\texttt{else}\;Q$	try (go[lt,mt] l) {P} else {Q}
$a^{\Delta t}\,\mathbf{?}\,(u)\,\texttt{then}\,P\,\texttt{else}\;Q$	try (on C read[t] u) {P} else {Q}
$a^{\Delta t}\,\mathbf{!}\,\langle v\rangle\,\texttt{then}\,P\,\texttt{else}\;Q$	try (on C write[t] v) {P} else {Q}

$k[\![P\,|\,Q]\!]$	`system sys−name`
	`  @location k`
	` P
		Q`
		`  endlocation`
		`endsystem`

Table 3. *timo2lang* function

We show how a high-level structure from TiMo becomes a low-level implementation by defining two functions, *timo2lang* and *lang2impl*, which translate a process expression into a *TLang* program, and then a *TLang* program into a Java implementation. Let *TimoProc* be the set of all TiMo processes, *TLangProg* the set of all programs/specifications which can be written in *TLang* language, and *JavaCode* the set of correct Java programs. The functions are defined as follows:

$timo2lang : TimoProc \rightarrow TLangProg$
$lang2impl : TLangProg \rightarrow JavaCode$

Since the TiMo processes are built structurally, it is enough to show how the basic syntactic structures are handled. For the basic cases, functions *timo2lang* and *lang2impl* are presented in Tables 3 and 4. The left column represents the argument, and the right one is the result of function application.

TiMo processes can also be encoded directly in Java without using the intermediate language *TLang*. This can easily be proved by composing the functions *lang2impl* and *timo2lang*; $lang2impl \circ timo2lang$ takes a process expression and returns its Java program. Note that $timo2lang(TimoProc) \subset TLangProg$ and $timo2lang(TLangProg) \subset JavaCode$, thus not every program written in TLang or Java encodes a TiMo process.

Proposition 1 and Remark 2 show a sound way of deriving Java code for mobility starting from TiMo specification. Thus we conclude with the following statement.

try (go[lt,mt] l){P} else {Q}	`try{` `if(!moved){` `moved = true;go(l, lt, mt);` `} else {` `try {` `//P body` `}catch(Exception e){` `// agent failed` `}` `}catch(Exception e1){//Q body}`
try (on C read[t] u){P} else {Q}	`try {` `u = in(c,u.getClass(), t);` `// P code` `}catch(Exception e){// Q code}`
try (on C write[t] v){P} else {Q}	`try {` `out(c, v, t);` `// P code` `}catch(Exception e){// Q code}`
`system sys-name` `@location k` `P | Q` `endlocation` `endsystem`	specific functions which create an object containing the system description (agents and their distribution).

Table 4. *lang2impl* function

Remark 3. Each agent specified in TiMo can be implemented by the software platform defined by *MCTools*, and its execution reflects the operational semantics of TiMo.

6 Example

We present a simple problem which demonstrates the usability of the migration platform and timing constraints. The scenario is given by the discovery of a specific resource, in our case a shop location (though it could be any other like a printer, a scanner etc). We first describe the problem, then we show how it can be encoded into TiMo. Then, we briefly discuss the *TLang* implementation, and the running Java code.

Suppose that we have a *Client* who wishes to find the best *Shop* for a specific product. Although the client does not know where to find the specific product, it knows a location where a *Broker* may inform about the right place. The problem is that the *Broker* is available only for some limited amount of time. Moreover, the best shop changes over time in such a way that in the first 4 units of time the best one is *shopA* and then, in the next 7 units of time the best one is *shopB*. Besides, the *Client* has to do some internal work and cannot leave its location in the first 2 units of time. After that, it may move in 3 units of time to the *Broker* location, and it cannot afford to spend more than 2 units of time at the *Broker* location. The communication channel between the *Client* and the *Broker* is *A*. The *Client* is located initially at *home*, and the *Broker* at location *info*. The whole system is named *Shops*. The TiMo specification for *Shops* is as follows:

$$Client = \mathbf{go}^{2\Delta 3}\, info\, \mathtt{then}\, (A^{\Delta 2}\, \mathbf{?}\, (u)\, \mathtt{then}\, \mathbf{go}^{0\Delta 0}\, u\, \mathtt{else}\, \mathbf{go}^{0\Delta 3}\, home\,)$$

$$Broker = A^{\Delta 4}\, \mathbf{!}\, \langle shopA\rangle\, \mathtt{then}\, 0\, \mathtt{else}\, A^{\Delta 7}\, \mathbf{!}\, \langle shopB\rangle$$

$$Shops = home[\![Client]\!] \mid info[\![Broker]\!]$$

Minimally, the *Shops* system may be encoded in *TLang* as in Figure 3. We say minimally because we do not see any result from this, and the agent does not do anything besides communicating and migrating. A possible running result of this system, completed with some output, is presented in Figure 4. We say "a possible running" because if the agent does not arrive in time at location *info*, or a destination is unreachable, then the output would be different.

```
#extended-language
#location home(192.168.1.2:9000,0);
#location info(192.168.1.2:9009,0);
#location shopA(192.168.1.2:9099,0);
#location shopB(192.168.1.2:9999,0);


const channel<location> A;

agent Client
 location shop;
 try (go[2,3] info){
  try (on A read[2] shop) {
   try (go[0,0] shop){}
  } else {
   try (go[0,3] home){}
  }
 }
endagent
```

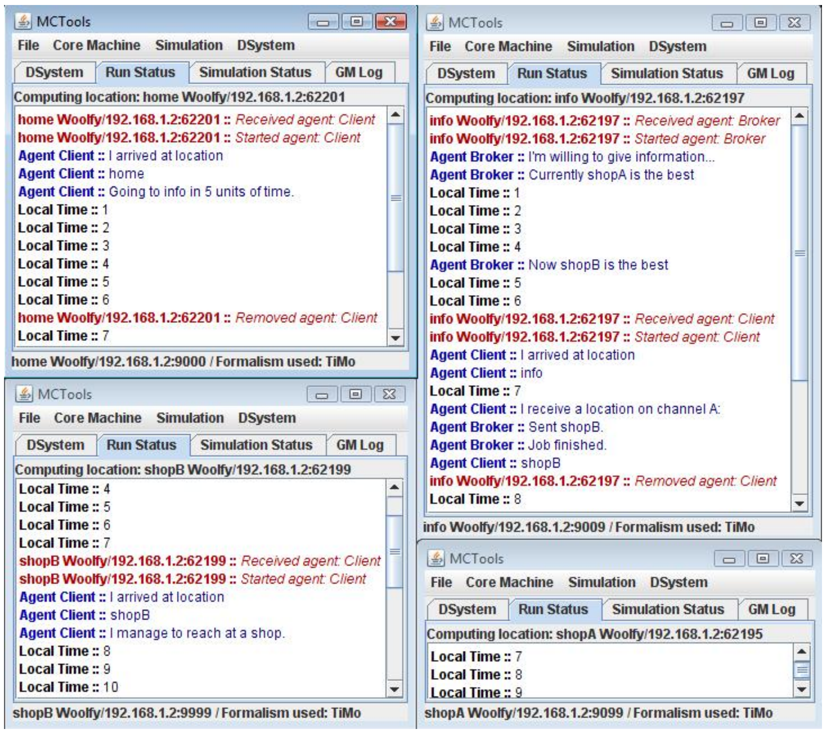
```
agent Broker
 try (on A write[4] shopA)
 {
 } else {
  try (on A write[7] shopB)
  {
  }
 }
endagent

system Shops
 @location home
  Client
 endlocation
 @location info
  Broker
 endlocation
endsystem
```

Fig. 3. *TLang* encoding of *Shops* system

Some explanations are needed in order to understand the implementation. The first line tells the compiler that the program will use Java types and instructions. The next line describes the location addresses and communication ports. For example *home (192.168.1.2: 9000, 0)* tells that *home* location has the IP address *192.168.1.2*, it runs the basic framework at port *9000* and has no preferred port for receiving agents. The next line declare a global channel named A for messages of type *location*. The rest of the specification deals with agents code and distribution.

Figure 4 presents the result of system execution after the agents were completed with some text output. Each window corresponds to a location which is written in the status bar. The text boxes contains system messages and agents output, providing useful information about the system evolution. The *Client* starts at location *home*, and after 5 units of time it moves to location *info*. At *info* he communicates with the *Broker* and receives the name *shopB* along channel A. It is important to observe that he does not interact at local time 6, when he arrives, but after another tick. To understand why this happens it is enough to follow the *Client* configuration evolution: $home[(go,\, (2,\, 3),\, info)] \xrightarrow{tick}^{5} \xrightarrow{go} info[\#(in\, A,\, 2\, shop)] \xrightarrow{tick} info[(in\, A,\ 2,\ shop)]$. This emphasizes the correspondence between the implementation and the TiMo semantics. Then the agent moves to location *shopB* where it prints a confirmation message. Location *shopA* remains empty during the entire period of time. If we describe the system by a configuration perspective, we get the following evolution which abstracts the system execution and follows the TiMo semantics:


MCTools
File Core Machine Simulation DSystem
DSystem Run Status Simulation Status GM Log

```
Computing location: home Woolfy/192.168.1.2:62201
home Woolfy/192.168.1.2:62201 :: Received agent: Client
home Woolfy/192.168.1.2:62201 :: Started agent: Client
Agent Client :: I arrived at location
Agent Client :: home
Agent Client :: Going to info in 5 units of time.
Local Time :: 1
Local Time :: 2
Local Time :: 3
Local Time :: 4
Local Time :: 5
Local Time :: 6
home Woolfy/192.168.1.2:62201 :: Removed agent: Client
Local Time :: 7
home Woolfy/192.168.1.2:9000 / Formalism used: TiMo
```

```
Computing location: shopB Woolfy/192.168.1.2:62199
Local Time :: 4
Local Time :: 5
Local Time :: 6
Local Time :: 7
shopB Woolfy/192.168.1.2:62199 :: Received agent: Client
shopB Woolfy/192.168.1.2:62199 :: Started agent: Client
Agent Client :: I arrived at location
Agent Client :: shopB
Agent Client :: I manage to reach at a shop.
Local Time :: 8
Local Time :: 9
Local Time :: 10
shopB Woolfy/192.168.1.2:9999 / Formalism used: TiMo
```

```
Computing location: info Woolfy/192.168.1.2:62197
info Woolfy/192.168.1.2:62197 :: Received agent: Broker
info Woolfy/192.168.1.2:62197 :: Started agent: Broker
Agent Broker :: I'm willing to give information...
Agent Broker :: Currently shopA is the best
Local Time :: 1
Local Time :: 2
Local Time :: 3
Local Time :: 4
Agent Broker :: Now shopB is the best
Local Time :: 5
Local Time :: 6
info Woolfy/192.168.1.2:62197 :: Received agent: Client
info Woolfy/192.168.1.2:62197 :: Started agent: Client
Agent Client :: I arrived at location
Agent Client :: info
Local Time :: 7
Agent Client :: I receive a location on channel A:
Agent Broker :: Sent shopB.
Agent Broker :: Job finished.
Agent Client :: shopB
info Woolfy/192.168.1.2:62197 :: Removed agent: Client
Local Time :: 8
info Woolfy/192.168.1.2:9009 / Formalism used: TiMo
```

```
Computing location: shopA Woolfy/192.168.1.2:62195
Local Time :: 7
Local Time :: 8
Local Time :: 9
shopA Woolfy/192.168.1.2:9099 / Formalism used: TiMo
```

Fig. 4. Running of the *Shops* system

$home[(go,(2,3),\ info)] \times info[(out\,A,4,shopA)] \xrightarrow{tick}{}^{4} \xrightarrow{com_{fail}} home[(go,(0,1),\ info)] \times info[\ \#(out\,A,7,shopB)] \xrightarrow{tick} \xrightarrow{go} \xrightarrow{tick} home[0] \times info[(out\,A,6,\ shopB),(in\,A,2,shop)] \xrightarrow{com} \xrightarrow{tick} info[(go,(0,0),shopB),0_B] \xrightarrow{go} \xrightarrow{tick} info[0_B]$ $\times shopB[0_C]$. By 0_C we denote the terminated *Client* process and by 0_B the terminated *Broker* process. We omit the empty location configuration.

7 Related Work and Concluding Remarks

The paper presents a software platform for timed migration. We develop this platform starting from a process algebra which uses time constraints to control both the communication between processes and movement between locations.

We design this platform in two layers. The lower layer deals with low-level details and provides the migration mechanism. It also implements the general concepts used in process algebra of the upper layer, and so it can be re-used for the implementation of other formalisms with mobility. We emphasize the upper layer implementing TiMo, a process algebra with communication, migration and temporal aspects. An intermediate language called *TLang* is used to specify a TiMo distributed systems. The novel features of the lower layer are given by a reusable mobility mechanism using various Java class-loading techniques, as well as the possibility to see the formal evolutions (defined by their semantics) for both TiMo and dπ-calculus which can emphasize possible discrepancies between formal specifications and their implementations. Another feature is represented by the implementation of a distributed protocol without a central coordinator; it allows a sound development methodology of agents on a single machine followed by their distribution among locations.

In TiMo the novelty is provided by the use of two timers lt and mt, and a safety process depending on the the migration timer mt. This aspects are reflected in the corresponding implementation of TiMo.

A similar platform called IMC is presented in [1]. Based on this platform, the authors have implemented the distributed π-calculus. *MCTools* lower layer corresponds to IMC, and offers more functionalities based on a different architecture. Let us mention few differences: the naming mechanism, an integrated formal evolution engine, remote actions which allow to initialize a distributed environment based on a specification. Moreover, using two layers, *MCTools* implements a handling mechanism of various formalism which is not available with IMC.

Several formalisms and implementations have been proposed in the recent years. Among them, we mention Facile [10], join calculus [5] and nomadic π-calculus [9]. Compared to these works, *MCTools* provide a flexible migration layer which could be used by several formalisms. The migration is based on the movement of the agent class and its dependencies from each location to any other location using *MCTools* lower layer.

References

1. L. Bettini, R. De Nicola, D. Falassi, M. Loreti. Implementing a Distributed Mobile Calculus Using the IMC Framework. *Electronic Notes in Theoretical Computer Science* vol.181, 63–79, 2007.
2. G. Ciobanu, C. Juravle. *MCTools: A Software Platform for Mobility and Timed Interaction*. Technical Report FML-09-01, Romanian Academy, 2009.
3. G. Ciobanu, M. Koutny. Modelling and Verification of Timed Interaction and Migration. *Lecture Notes in Computer Science* vol.4961, Springer, 215–229, 2008.
4. G. Ciobanu, C. Prisacariu. Timers for Distributed Systems. *Electronic Notes in Theoretical Computer Science* vol.164, 81–99. 2006.
5. C. Fournet, G. Gonthier, J.-J. Levy, L. Maranget, D. Remy. A Calculus of Mobile Agents. *Lecture Notes in Computer Science* vol.1119, Springer, 406-421, 1996.
6. M. Hennessy, J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation* vol.173, 82–120, 2002.
7. J. Magee, J. Krame. *Concurrency: State Models and Java Programs.* Wiley , 2006.
8. R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1991.
9. P. Sewell, P.T. Wojciechowski, B.C. Pierce. Location-independent Communication for Mobile Agents: a Two-level Architecture. *Lecture Notes in Computer Science* vol.1686, 1-31, 1999.
10. B. Thomsen, L. Leth, T.-M. Kuo. A Facile Tutorial. *Lecture Notes in Computer Science* vol.1119, Springer, 278–298, 1996.
11. J.E. White. *Telescript Technology: The Foundation for the Electronic Marketplace.* White Paper, General Magic, 1994.