MCTools: A Software Platform for Mobility and Timed Interaction

G. Ciobanu and C. Juravle

Approved for public release; further dissemination unlimited.

Prepared by
Formal Methods Laboratory.

Series of
Technical Reports (Institute of Computer Science Iași)

ISSN 1842 - 1490
FML is the Formal Methods Laboratory from the Institute of Computer Science of the Romanian Academy and is located in Iași. It is a research institution aimed at developing new formalisms for challenging open problems in computer science, systems biology and other emerging research fields. FML is headed by Dr. Gabriel Ciobanu.

IIT is the Institute of Computer Science of the Romanian Academy and is concerned with basic research in computer science.

Romanian Academy of Science is the supreme forum of science from Romania.
Calea Victoriei 125, Sector 1, 010071, Bucharest, ROMANIA.
**Telephone:** +40 21 2128640  
**Telefax:** +40 21 2116608  
**Web:** [http://www.academiaromana.ro/](http://www.academiaromana.ro/)

Printed in Romania and reproduced directly from the authors original copy.

---

**Copyright:** ©2009 Formal Methods Laboratory Iași (FML)  
Blvd. Carol I, nr.8, 700505, Iași, ROMANIA (RO)  
Institute of Computer Science, Romanian Academy, branch Iași.  
**Telephone/Telefax:** +40 232 241708  
**Web:** [http://iit.iit.tuiasi.ro/fml](http://iit.iit.tuiasi.ro/fml)  
**Contact:** gabriel@iit.tuiasi.ro

**Notice:** Permission is hereby granted for the redistribution of this material so long as this item is redistributed in full and with appropriate credit given to the author(s). All other rights reserved.

**Series of** Technical Reports (Institute of Computer Science Iași)  
**ISSN** 1842 - 1490
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 framework for agent migration, separating the migration mechanism such that it can be reused for other systems with mobility. We describe the frameworks architectures 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 framework for a simple problem.
1 Introduction

Mobile applications represent an important topic in distributed system field. Mobility is difficult in both the modeling part and in the implementation part, especially when time is also considered. To address the modeling part, many formalisms have been proposed over the years such as π-calculus [9], distributed π-calculus [7], TiMo [4] or mobile ambients [3]. Concerning the implementation, there are several architectures and different programming languages (Telescript [12] or Java [1]) 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 implementation is not clearly defined.

Our aim is to provide a framework for agent migration which corresponds to a formal model. Starting from TiMo, we implemented such a framework. 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 proof the extensibility of MobileCalculi framework we also implemented a software framework for dπ-calculus [7]. The whole system is named MCTools. Thus, MCTools represents a system for working with implementations of mobile calculi.

MCTools system is developed according to a choreography based distributed architecture. It is working without a central coordinator. Agents are free to roam an 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, an example is presented in Section 6.

2 TiMo

TiMo [4] 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 [9] family, close to the distributed π-calculus [7] and timed distributed π-calculus [5]. 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 time is no longer indefinite. Moreover, if an action
does not happen in a predefined time, then the process continues with a safety alternative.

The time constraints associated with input and output restricts the period in which these capabilities could be applied. After the predefined deadline is reached, another continuation for the process behaviour is chosen. For instance, if a process $a^{\Delta t} ? (u) \text{then } P \text{ else } Q$ does not receive a value on channel $c$ in the next 3 units of local time, it will behave according to process $Q$.

The time constraint associated with migration is composed from two timers. The first one is the local time, and it represents the time dedicated to internal and local actions which are executed before migration. The second one is the migration time, and it represents the maximum time dedicated to migration. For instance, a process $m[go^{\Delta 10} k \text{then } P \text{ else } Q]$ executes local actions for 4 units of time (at the current location $m$), and then it migrates in maximum 10 units of time to location $k$ where it behaves as $P$. Note that since $v$ in $go^{\Delta mt} v \text{then } P \text{ else } Q$ is a variable, migration supports a flexible movement of processes between locations.

2.1 Syntax

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 $m_I \geq 0$). TriMO is presented in Table 1.

$$
P, Q ::= \ 0
\quad | \quad a^{\Delta t} ! \langle v \rangle \text{then } P \text{ else } Q
\quad | \quad a^{\Delta t} ? (u) \text{then } P \text{ else } Q
\quad | \quad go^{\Delta mt} v \text{then } P \text{ else } Q
\quad | \quad I(v_1, \ldots, v_{m_I})
\quad | \quad P | Q
\quad | \quad \#P
\]

$$

Table 1: TriMO syntax

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 $a^{\Delta t} ! \langle v \rangle \text{then } P \text{ else } Q$ 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 $a^{\Delta t} ? \langle u \rangle \text{then } P \text{ else } Q$ has a similar behaviour.
Process $\text{go}^{lt} v \text{then } P \text{else } Q$ implements mobility. It first waits $lt$ units of time which represents the local time dedicated to local/internal 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 $P | Q$. 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.

2.2 Operational Semantics

TiMo operational semantics has two parts. The first one is the structural equivalence on networks which is the smallest congruence defined by the equalities of Table 2.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M</td>
<td>N \equiv N</td>
</tr>
<tr>
<td>$(M</td>
<td>N)</td>
</tr>
<tr>
<td>$k[P] \equiv k[P</td>
<td>0]$</td>
</tr>
<tr>
<td>$k[I(l_1, ..., l_m)] \equiv k[I(l_1/u_1, ..., l_m/u_m)]$</td>
<td></td>
</tr>
<tr>
<td>$k[P</td>
<td>Q] \equiv k[P]</td>
</tr>
</tbody>
</table>

Table 2: Rules of structural equivalence for TiMo

The operational semantics is given by the rules presented in Table 3. Looking to the labels of the transitions, there are two kinds of transition rules: $M \xrightarrow{\beta} N$ and $M \xrightarrow{\varphi} N$. The first one corresponds to the execution of an action $\beta$, while the second one represents a timing tick. The action $\beta$ 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.

$\phi(N)$ is the time-passing function, and it is described in the following consecutive stages:

- each top-level expression $I(l_1, ..., l_m)$ is replaced by the corresponding definition $\{l_1/u_1, ..., l_m/u_m\} P_i$;
- each top-level expression of the form $a^{\Delta}... \text{then } P \text{else } Q$ or $\text{go}^{0\Delta}... \text{then } P \text{else } Q$ is replaced by $\#Q$;
- each top-level expression of the form $a^{\Delta}! (l)$ then $P$ else $Q$ is replaced by $a^{\Delta(t-1)}! (l)$ then $P$ else $Q$;
- each top-level expression of the form $a^{\Delta}? (u)$ then $P$ else $Q$ is replaced by $a^{\Delta(t-1)}? (u)$ then $P$ else $Q$;
\[
\begin{align*}
\text{MOVE:} & \quad k[\text{go}^{0 \Delta m t} l \text{ then } P \text{ else } Q] \xrightarrow{k:l} l[#P]
\end{align*}
\]
\[
\begin{align*}
\text{COM:} & \quad l[a^{\Delta t} ! \langle l \rangle \text{ then } P \text{ else } Q] \xrightarrow{k:a(l)}
\left[ a^{\Delta t'} ? (u) \text{ then } P' \text{ else } Q' \right]
\xrightarrow{l[#P | #\{l/u\}P']}
\end{align*}
\]
\[
\begin{align*}
\text{PAR:} & \quad N \xrightarrow{\beta} N'
\end{align*}
\]
\[
\begin{align*}
\text{STRUC:} & \quad N \equiv N' \quad N \xrightarrow{\beta} M \quad M \equiv M'
\end{align*}
\]
\[
\begin{align*}
\text{TIME:} & \quad N \not\xrightarrow{\sqrt{\phi(N)}}
\end{align*}
\]

Table 3: TiMO operational semantics

- each top-level expression of the form \( \text{go}^{lt \Delta m t} \text{ then } P \text{ else } Q \) is replaced by \( \text{go}^{lt' \Delta (mt')} \text{ then } P \text{ else } Q \) where \( lt' = \max\{0, lt - 1\} \) and

\[
mt' = \begin{cases} 
mt & \text{if } lt > 0 \\
\max\{0, mt - 1\} & \text{if } lt = 0
\end{cases}
\]

- all occurrences of the special symbol \# are deleted,

where \( P \) is a top-level expression not containing a symbol \#. Note that only after the \( lt \) timer, decremented by \( \phi \), reaches 0, the process migrates to the desire location.

### 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 \( \pi \)-calculus [9] family) and the practical part which deals with mobile code and facilitate the implementation of mobile systems. 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 the 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 TiMO framework 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.

To ease the user interaction with this framework we also developed a generic purpose GUI. 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 system.

We describe the framework implementation from a functional viewpoint. A global view of the framework 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.

3.1 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 3.2, by managing the low level details and acting as a mobility facade. After processing a migration request, a formalism finally
passes it to the core module. At this level the request contains only the agent which wants to migrate and the location name of the destination. The core module will handles such a request by:

- resolving the name to a real address with the help of a location mapping;
- obtaining a connection with the remote host according to a specific protocol (low level communication);
- telling the mobility mechanism to migrate the agent using this connection.

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.

### 3.2 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. It is solely defined by means of interfaces, allowing anyone to easily add, modify or replace the migration mechanism. It is based only on four interfaces: for migrating code, for marshallers, for unmarshallers and for an abstract factory which create these entities.

Here we describe the default migration mechanism. There are many ways in which a migration mechanism can be implemented (with respect to Java), and all have to take into account that an agent is a class and so it is not expected that the destination location knows its definition. Thus, besides its data, it is also necessary to transfer not only its definition but also its dependencies. For example if agent $A1$ depends on agent $A2$ (let’s say by a spawn primitive), then agent $A2$ definition should migrate together with $A1$. 
Among the possible alternatives we consider the solution based on bytecode migration. The implementation is described below. It ensure the dependencies migration by using special class loaders.

The main idea is to retain the bytecode of agents in a static 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. The steps are described below:

1. When an agent is created, it is loaded with a special class loader $MClassLoader$.

2. This extracts the class bytecode and stores it in a static local repository. The bytecode is extracted from the compiled file of the agent. The repository is common for all the $MClassLoader$ on the same machine.

3. At migration, the agent definition is searched in this repository.

4. The class dependencies are extracted through reflection. The base classes and framework classes are excluded from these dependency. Note that this way of extracting dependencies has a limitation; it cannot reveal dependencies hidden only as local variables inside a method body.

5. The final dependencies are searched in the same repository as the agent code.

6. The bytecode of agent and its dependencies are sent first.

7. The agent object is sent through a standard output stream.

8. At destination, a special stream ($MObjectInputStream$) is used. This specialized variant can load the object it received with another class loader than the default one. This ensures that the agent object is loaded with the class loader that loaded its definition and its dependencies definition.

9. With the help of the $MClassLoader$, the agent class is reconstructed and loaded. The same works for agents dependencies.

10. The new bytecode is added to the local repository.

11. Now the agent object can be received. Since its definition is already loaded, a new object with same data can be created.

12. The agent is added and started at the destination environment.
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 allow 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

Here we present some details on the software framework inspired by the TiMo model. The framework is built upon the MobileCalculi framework, and offers an environment for creating and executing TiMo agents. A TiMo agent is an agent whose actions are based on primitives from the TiMo formalism. Thus the basic actions of such an agent are migration and communication on channels. It is difficult to write useful/practical agents using only these primitives, and for this reason we let the agents code to be completed with standard Java code. In this way we obtain agents which can work effectively over a network, and interact between them in correspondence with the TiMo paradigm.

In order to help writing the TiMo agents, we develop a language called TLang to inter-
mediate between the high-level TiMo and the low-level Java code. 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. Figure 2 presents the transition from TiMo formalism to a MobileCalculi framework.

![Figure 2: From TiMo to MobileCalculi Framework](image)

The main features of TiMo implementation are:

- the possibility of 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.

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.

4.1 Entities and Primitives

We first describe how specific entities are represented.

Agent A generic agent is represented by a TimoProcess class which extends the execution pattern from MobileCalculi framework. Every specific agent must extend this class and provides implementation for the body() method which is actually the agent body to be executed. The execution model is completed by implementation as final methods of T\textsc{m} primitives: in() - input on channel; out() - output on channel; go() - migration. Each agent runs in its own execution thread inside a T\textsc{m} environment represented by a TimoLocation.

Location A T\textsc{m} location is implemented as TimoLocation which extends a basic execution context from MobileCalculi framework. It keeps two lists: one for the available resources (channels in our case), and one for the current agents requests. The resources are handled by a ChannelHandler object, and the request by a ResourceHandler object. More about these handlers is discussed in Section 4.1.

Channel It is represented by Channel class. It is a generic class, parametrized with the type of messages the channel can handle. This class offers synchronizing mechanisms for input and output actions.

Name The implementation of names is mainly the one from the MobileCalculi framework, only the name for channels being extended with information about the channel type.
Primitives

We describe the implementation of TiMo primitives, namely migration and communication. 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 analyses 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).

The precise steps of migration are presented bellow; a successful migration can be visualized in Figure 3.

1. The agent calls the method $go(locationName, localTime, migrationTime)$ inherited from TimoProcess. $localTime$ corresponds to the first timer in the formalism and $migrationTime$ corresponds to the second timer (also referred as migration time).

2. The method simply translates the call at the current location, adding a reference to the agent. The request is encapsulated in a GoRequest object, and sent to the RequestHandler. The object comprises of the destination and the associated timers. The agent execution is suspended until the request is marked as finished.

3. At every tick of the local clock the RequestHandler tries to resolve the requests it has received. In case of GoRequest, the ResourceHandler decrements the localTime by one at every tick. The request is marked as finished when the localTime reaches 0. After that the migrationTime starts to be decremented.

4. When the request is finished, the migration call is transformed into a request to the core module of MobileCalculi framework. The request contains a reference to the agent and the location name. This part is synchronized, so only a single process can migrate at one time.

5. The core module tries to resolve the location name by using its private repository.
6. If the location name corresponds to a real location, then a communication channel is opened with the remote location. Otherwise, an UnknownLocation exception is thrown, and the migration fails. If an error appears during communication with the remote location, then a CommunicationException is thrown. In case of exception jump at step 13.

7. The core module calls the loaded migration mechanism which migrates the agent to the destination. In case of errors, a MigrationException is thrown (jump at step 13).

8. At destination, the agent is received through the loaded migration mechanism.

9. The destination sends a receive message to the agent initial location telling that the agent was received. The message is sent using the core module of MobileCalculi framework.

10. If the message is received before the migrationTime reaches 0, then a confirmation message is sent to the destination.

11. At the initial location, the agent thread is stopped and the agent is deleted from the list of active processes. At destination, after receiving the confirmation message, the agent is added to the local environment and it is started by calling the method body(). The synchronization for restarting the execution from the same point is solved by the programmer. The default behaviour if no confirmation message is received at destination is to remove the agent.

12. If migrationTime expires, and no receive message was received, a TimeOutException is thrown. The safety process is activated and the agents continue its execution at the current location.

13. In case of exception, the migration fails and the agent continues its execution at the current location by using the safety process.

**Communication** between agents takes place on channels, and it is based on the rendezvous mechanism [8]. For an agent, receiving an information on a channel follows the steps described below:

1. The agent calls in(channelName, messageType, waitingTime) inherited from TimoProcess; channelName is the name of channel on which the communication takes place, messageType is the type of the message to be received, and waitingTime corresponds to channel timer from TiMO formalism.

2. The request is sent further to the current location where it is encapsulated in a Request object, and then sent to the channels manager ChannelHandler and to the requests manager RequestHandler. The agent is locked until the request is finished with either success or failure.
3. At every tick the RequestHandler tries to resolve the requests as follows:

   (a) it decrements the associated timer by one;

   (b) then it calls the ChannelHandler which looks for the channel with the specified name. If the channel is not found, it means that we should deal with a new one, and so a new channel with the same type and name is added to the resource list. Next, the request is sent to this new channel. If the channel is found but it does not have the specified type, an exception (ChannelTypeException) is thrown and the communication fails. If the channel is found and has the same type, the request is forwarded to the channel.

   (c) at the channel level, the request is handled as follows. The channel has two lists: one for reading requests, and one for writing requests. When it is called to resolve a request, it looks in these lists and makes pairs (input - output) until one list...
becomes empty. After the matching step, it transfers data from the output to the input end, and mark the requests as being finished and satisfied.

(d) If the timer reaches 0 and no communication partner has been found, the request is marked as finished, but not satisfied.

4. After the request is finished, there are two possibilities. If it is satisfied (meaning that a communication with other agent occurred) then the received data is simply returned. If such thing did not happen, then a TimeOut exception is thrown.

5. If an exception was thrown, then it is caught in the agent body, case in which the safety process should be executed.

The sending procedure is similar; the only difference is that if the request is finished successfully, the call does not return a value.

4.2 TiMo Language and Compiler

TLang represents an intermediate language which links the high-level TiMo model with the low-level of implementation in Java. With an intuitive syntax close to both TiMo and conventional programming, TLang is created to facilitate the development of mobile agents and specification of distributed systems.

| go[l^\Delta mt] l then P else Q | try (go[l,mt] l) {P} else {Q} |
| a^\Delta t ? (u) then P else Q | try (on C read[t] u) {P} else {Q} |
| a^\Delta t ! (v) then P else Q | try (on C write[t] v) {P} else {Q} |
| k[P | Q] | system sys-name @location k |
| | P | Q endlocation |
| | endsystem |

Table 4: TiMo to TLang translation

TLang uses only a limited set of Java types and instructions. The allowed types are Integer, Boolean and String (together with location and parametrized channels). The instructions which can be used, besides TiMo primitives, are if-then-else and while. An important feature of TLang is that it permits to embed native Java code into agents code. This code is directly translated into agent Java definition. Of course, we can write the agents directly into Java using its full features; however, by using TLang it is easier to develop the desired system. Even if TLang does not include all Java functionalities, it provides several important advantages.

First of all, TLang syntax is close to TiMo syntax, so it is easier to implement agents having a TiMo description. It provides a way to specify an entire system, including real addresses of machines, preferred communication ports and agents distribution map. Since
try (go[l,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 5: TLang - Implementation encoding

it has features of the conventional programing style, it is easier to share data (such as a channel name) between agents through global variables. For example, in case of channel name sharing, if it is written directly in Java code, one have to take care to encode the same channel in both agent classes. In TLang, we only have to declare a global channel and then use it, the compiler ensures that it is correctly encoded. Moreover, TLang provides a simple way of writing (private) channels, known only by some agents. On the other hand, even if the provided migration mechanism implements a weak mobility, TLang simulates a strong mobility by forcing any go primitives to execute only once. However, the programmer still has to do some work to ensure that the agent starts from the correct execution point. Another advantage is that the language is well typed, any type inconsistency resulting in a compiling error. For example, if an agent is coded to receive a String over a channel which is dedicated to location variables, then an error is triggered at compilation. It is hard to ensure such a condition in Java. This reduces the number of failed agents due to type errors.
The result of compilation is, besides the Java code of agents, an object containing information about the specified distributed system. This object can be directly loaded into the low level framework which can then start the distributed environment.

We present how primitives of TiMo are encoded into TLang in Table 4. The translation from TLang into Java code is summarized in Table 5.

5 Implementation Correctness

In this section we show that our implementation corresponds with TiMo high-level semantics. We first define a formal notion of configuration. Using this notion and, we reason about the implementation correctness (with respect to TiMo semantics).

**DEFINITION 5.1** Given a process \( R \) specified in TiMo, we define the process stack \( S(R) \), or simply \( S \), as in Table 6.

\[
\begin{align*}
R & \quad \mapsto \quad S(R) \\
\text{go}^\Delta \text{lm} \ 	ext{then} P \ 	ext{else} Q & \quad \mapsto \quad \{ \text{go}, (l, m), l \} \\
\text{a}^\Delta t \ ?(v) \ 	ext{then} P \ 	ext{else} Q & \quad \mapsto \quad \{ \text{in c, t}, v \} \\
\text{a}^\Delta t \ !\langle u \rangle \ 	ext{then} P \ 	ext{else} Q & \quad \mapsto \quad \{ \text{out c, t, u} \} \\
P \mid Q & \quad \mapsto \quad S(P) \ 	ext{and} \ S(Q) \ 	ext{distinct stacks}
\end{align*}
\]

Table 6: Process Stack Definition

**REMARK 5.1** This definition is consistent with both the theoretical view which presents the process as a sequence of actions, and the practical view where each process has a stack from where the next action is executed. Moreover, it is consistent with the TiMo framework. 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 $S_1^n$, $S_2^n$, ..., $S_n^n$ of the processes which run at that location $l$, and it is written as $l[S_1^n, S_2^n, ..., S_n^n]$. 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 $l_1[S_1^n, ..., S_{n_1}^n] \times ... \times l_n[S_1^n, ..., S_{n_n}^n]$ a network with $n$ locations, where for each location $l_i$ the number of processes is provided by $n_i$.

We denote by $\text{Config}$ 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 5.2** Over the configuration set, we define the transition function $\delta : \text{Config} \times \text{CT} \rightarrow \text{Config}$, where $\text{CT} = \{\text{tick}, \text{subst}, \text{go}, \text{fail}_\text{com}, \text{fail}_\text{go}\}$.

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

- $l[S_1, ..., (\#(\text{chact} a, t, x), ..., S_n)] \xrightarrow{\text{tick}} l[S_1, ..., (\text{chact} a, t - 1, x), ..., S_n]$ where $\text{chact} \in \{\text{in}, \text{out}\}$ and $x \in \text{Val} \cup \text{Var}$.

- $l[S_1, ..., (\#(\text{go}, (l t, m t), l), ..., S_n)] \xrightarrow{\text{tick}} l[S_1, ..., \text{(go}, lt - 1, m t), l), ..., S_n]$ provided that $lt > 0$ and $l \in \text{Loc}$.

- $l[S_1, ..., (\#(\text{go}, (0, m t), l), ..., S_n)] \xrightarrow{\text{tick}} l[S_1, ..., \text{(go}, 0, m t), l), ..., S_n]$ provided that $mt \geq 0$ and $l \in \text{Loc}$.

- $l[S_1, ..., (\text{in} a, t, v), ..., (\text{out} a, t', u), ..., S_n] \xrightarrow{\text{subst}} l[S_1, ..., \#S(\{u/v\}P), ..., \#S(P'), ..., S_n]$ provided that $\min(t, t') \geq 0$, the stack of in action is $[(\text{in} a, t, v), P, Q]$ and that of out action is $[(\text{out} a, t', u), P', Q']$.

- $l[S_1, ..., (\text{in} a, t, x), ..., S_n] \xrightarrow{\text{fail}_\text{com}} l[S_1, ..., \#S(Q), ..., S_n]$ provided that $t < 0$ and the stack of in action is $[(\text{in} a, t, v), P, Q]$.

- $l[S_1, ..., (\text{out} a, t', u), ..., S_n] \xrightarrow{\text{fail}_\text{com}} l[S_1, ..., \#S(Q), ..., S_n]$ provided that $t < 0$ and the stack of out action is $[(\text{out} a, t, u), P, Q]$.

- $l[S_1', ..., \text{(go}, (l t, m t), k), ..., S_{n_l}'] \times k[S_1^k, ..., S_{n_k}^k] \xrightarrow{\text{go}} l[S_1', ..., 0, ..., S_{n_l}'] \times k[S_1^k, ..., S_{n_k}^k, \#S(P)]$ provided that $lt = 0$, $mt = 0$ and the stack of go at is $[(\text{go}, (l t, m t), l), P, Q]$.

- $l[S_1', ..., \text{(go}, (l t, m t), k), ..., S_{n_l}'] \xrightarrow{\text{fail}_\text{go}} l[S_1', ..., \#S(Q), ..., S_{n_l}']$ provided that $lt = 0$, $mt = 0$, location $k$ is unreachable and the stack of go at $l$ is $[(\text{go}, (l t, m t), l), P, Q]$.

If none of the above rules can be applied, we apply one of the following rules:
− l[S₁,...,(chact a, t, x),..., Sₙ] \xrightarrow{\text{tick}} l[S₁,...,(chact a, t − 1, x),..., Sₙ]
where \( \text{chact} \in \{\text{in}, \text{out}\} \) and \( x \in \text{Val} \cup \text{Var} \).

− l[S₁,...,(go, (lt, mt), l),..., Sₙ] \xrightarrow{\text{tick}} l[S₁,...,(go, (lt − 1, mt), l),..., Sₙ]
provided that \( \text{lt} > 0 \) and \( l \in \text{Loc} \).

− l[S₁,...,(go, (0, mt), l),..., Sₙ] \xrightarrow{\text{tick}} l[S₁,...,(go, (0, mt), l),..., Sₙ]
provided that \( \text{mt} \geq 0 \) and \( l \in \text{Loc} \).

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

It is worth to note that \# is overloaded, being used for both processes and configurations. In this way we emphasize they have the same meaning related to the time-passing function.

**PROPOSITION 5.1** 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 \text{com-go} \ 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 \( k : a(l) \).

\[
\begin{array}{c}
R \\
\downarrow \beta \\
\text{config} \\
\downarrow \\
R' \\
\downarrow \text{config'}
\end{array}
\]

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

- **The communication case:** \( k[[\text{a}^{\Delta t} \ ? \ (v) \ \text{then} \ P \ \text{else} \ Q] \ | \ a^{\Delta t'} ! \ (u) \ \text{then} \ P' \ \text{else} \ Q'] \xrightarrow{k:a(u)} k[\#\{u/v\}P] \ | \ #P'] \). The configuration \( k[(\text{in} a, t, v), (\text{out} a, t', u)] \) corresponds to process \( R \). We apply subst rule: \( k[(\text{in} a, t, v), (\text{out} a, t', u)] \xrightarrow{\text{subst}} k[\#S(P), #S(P')] \). It is easy to see that the resulting configuration corresponds to the process \( R' \).

- **The migration case:** \( k[\text{go}^{t, \Delta t'} \ m \ \text{then} \ P \ \text{else} \ Q] \xrightarrow{k:\text{com}} m[#P] \). The corresponding configuration of the left-hand side is \( k[(\text{go}, (t, t'), m)] \). We apply go rule and get: \( k[(\text{go}, (t, t'), m)] \times m[0] \xrightarrow{\text{go}} k[0] \times m[#P] \). The resulting configuration is the corresponding one for the right-hand side which proves this case.

- **The failure cases** are treated in a similar manner, thus we do not present the details.

- **When an action comes after a #,** we should add an extra tick transition in order to keep the consistency between processes and configurations. Suppose that we have the following case: \( k[\text{go}^{t, \Delta t'} \ m \ \text{then} \ P \ \text{else} \ Q] \xrightarrow{k:\text{com}} m[#P] \) with \( P = a^{\Delta t} \ ? \ (v) \ \text{then} \ P_1 \ \text{else} \ Q_1 \ | \ a^{\Delta t'} ! \ (u) \ \text{then} \ P'_1 \ \text{else} \ Q'_1 \). The evolution goes further by applying the tick rule:
\[ m[#P] \xrightarrow{\text{tick}} m[a^{\Delta t-1}] \ ?(v) \text{ then } P_1 \text{ else } Q_1 | a^{\Delta t'-1} ! (u) \text{ then } P'_1 \text{ else } Q'_1 \]

The corresponding configuration transitions are: \( k[(\text{go}, (t, t'), m)] \times m[0] \xrightarrow{\text{go}} k[0] \times m[#S(P)] \). Expanding \( S(P) \) we obtain: \( m[#S(P)] = m[#(\text{in } a, t, v)] \xrightarrow{\text{subst}} m[#S(P_1), #S(P'_1)] \) which corresponds to the resulting process.

- The other cases are similarly treated, and not presented here.

**REMARK 5.2** Each syntactic structure from T1MO (with the mentioned restrictions) can be represented in TLang (the compiler language) which then can be translated into a Java implementation.

We show how a high-level structure from T1MO becomes a low-level implementation by defining two functions, \( \text{timo2lang} \) and \( \text{lang2impl} \), which translate a process expression into a TLang program, and then a TLang program into a Java implementation.

Let \( \text{TimoProc} \) be the set of all TiMO processes, \( \text{TLangProg} \) the set of all programs/specifications which can be written in TLang language, and \( \text{JavaCode} \) the set of correct Java programs. The functions are defined as follows:

\[
\text{timo2lang} : \text{TimoProc} \rightarrow \text{TLangProg}
\]

\[
\text{lang2impl} : \text{TLangProg} \rightarrow \text{JavaCode}
\]

Since the TiMO processes are built structurally, it is enough to show how the basic syntactic structures are handled. For the basic cases, function \( \text{timo2lang} \) is presented in Table 4 and function \( \text{lang2impl} \) in Table 5. The left column represents the argument, and the right one is the result of function application.

The TiMO processes can be encoded directly in Java without using the intermediate language TLang. This can easily be proved by composing the functions \( \text{lang2impl} \) and \( \text{timo2lang} \). The function \( \text{lang2impl} \circ \text{timo2lang} \) takes a process expression and returns its Java program. Note that \( \text{timo2lang}(\text{TimoProc}) \subset \text{TLangProg} \) and \( \text{timo2lang}(\text{TLangProg}) \subset \text{JavaCode} \), and so not every program written in TLang or Java encodes a TiMO process.

Proposition 5.1 and Remark 5.2 show a sound way of deriving Java code for mobility starting from TiMO specification. Thus we conclude with the following statement.

**REMARK 5.3** Each agent specified in TiMO (with the mentioned restrictions) can be implemented by the software platform defined by MCTools, and its execution reflects the operational semantics of TiMO.
6 Example

Here we present a simple problem which demonstrates the usability of the migration framework 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:

\[
\text{Client} = \text{go}^{2\Delta^3} \text{info then} (a^{\Delta^2} ? (u) \text{then go}^{0\Delta^0} u \text{else go}^{0\Delta^3} \text{home})
\]

\[
\text{Broker} = a^{\Delta^4} ! (\text{shopA}) \text{then 0 else a}^{\Delta^7} ! (\text{shopB})
\]

\[
\text{Shops} = \text{home[Client]} | \text{info[Broker]}
\]

Minimally, the Shops system may be encoded in TLang as below.

```plaintext
#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;
if (go[2,3] info){
  if (on A read [2] shop) {
    if (go[0,0] shop){}
  } else {
    if (go[0,3] home){}
  }
}
endagent

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

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

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.

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 location 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)] $\xrightarrow{\text{tick} \ 5} go$ info[#(in A, 2 shop)] $\xrightarrow{\text{tick}} info[(in A, 2, shop)]$. 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 obtain the following evolution which abstracts the system execution and follows the TiMo semantics:

$\xrightarrow{\text{tick}}$ info[(out A, 4, shopA)] $\xrightarrow{\text{com fail}}$ home[(go, (0, 1), info)] $\xrightarrow{\text{com}}$ info[#(out A, 7, shopB)] $\xrightarrow{\text{com}}$ info[(out A, 6, shopB), (in A, 2, shop)] $\xrightarrow{\text{com}}$ info[(go, (0, 0), shopB), 0_B] $\xrightarrow{\text{com}}$ info[0_B] $\xrightarrow{\text{com}}$ shopB[0_C]. By 0_C we denote the terminated Client process and by 0_B the terminated Broker process. We omit the empty
7 Related Work and Concluding Remarks

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

We design this framework 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 location configuration.
algebra with communication, migration and temporal aspects. An intermediate language called \textit{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 d\(\pi\)-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.

Several formalisms and implementations have been proposed in the recent years. Among them, we mention Facile [11], distributed join-calculus [6] and nomadic pi-calculus [10].

Facile provides mainly a local concurrency with message passing, but not distributed concurrency. It is influenced by \(\pi\)-calculus and higher-order calculi, and models process mobility via higher-order communication and remote process execution. There is no explicit notion of locality (for potential failures), and it is not possible for a running process to be migrated to a different locality, unless the process has been explicitly defined to allow such a migration.

A further step related to migration and location failures is given by the distributed join-calculus and in D\(\pi\) which have introduced process migration and access control. Distributed join-calculus takes a tree of locations, and migration primitive \textit{go} allows any location to move in order to become a child of any non-descendant.

Nomadic Pict is a distributed version of Pict language, focusing on distributed communication infrastructure for mobility. There are two kinds of communication: low-level location-dependent primitives, and high-level location independent primitives. Nomadic Pict works with trees of sites and agents; agents are located at sites, and are dynamically generated. Migration allows any agent \(a\) to move to become a child of any sites; migration is based on recording the current sites of agents, agent creation, and then recording the new site.

In distributed join and nomadic \(\pi\)-calculus, more exactly in the implementations derived from these formal models, a single execution of a program could become distributed. However it is mainly based on communication primitives, and different applications need wildly different communication infrastructure, with different synchronization, security, and performance.

Compared to these works, \textit{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 \textit{MCTools} lower layer.

A similar framework called IMC is presented in [2]. Based on this platform, the authors have implemented the distributed \(\pi\)-calculus. \textit{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, MCTools implements a handling mechanism of various formalism which is not available within IMC.

As further work, we plan to extend the framework by adding security to agent migration. We also plan to develop and implement a strong mobility mechanism at least for some specific class of agents.

References


