Mathematical Formalization of the Messenger Paradigm[1]
Draft

Giovanna Di Marzo
CUI, Centre Universitaire d'Informatique
University of Geneva
CH-1211 Geneva 4, Switzerland
dimarzo@cui.unige.ch

November 13, 1995

[1]This work is partly supported by Swiss National Fund for Scientific Research (FNSRS) grant FNRS 20-40631.94

Abstract

This technical report presents the syntax and semantics, in terms of transition system, of the messenger paradigm as it has been presented in [8].

Messengers are mobile agents able to collaborate and to coordinate their work, but are not considered as intelligent agents. More precisely, messengers are mobile codes exchanged between messenger platforms. Arriving messengers are immediately executed (interpreted) by platforms. Messengers communicate by the means of a global store, synchronize their execution using process queues, create new messengers and move themselves across the network using appropriate instructions.

The purpose of this mathematical formalization is to give a mathematical definition to the basic ingredients of the messenger paradigm: communication through global store, synchronization through process queues, creation of new messengers, and mobility.

This formalization is neither concerned with families of messengers collaborating to solve a common goal, nor with the notion of messenger taking part to a service.

Further work will be concerned with the choice of a formalism well suited for messengers' specifications.

Contents

Chapter 1

Introduction

In order to derive properties for systems built with messengers, we need a formal specification for the messenger paradigm. This report is the first step in this direction, as it states a mathematical definition to the semantics we give to messenger systems.

Messengers are mobile codes traveling across the network between different platforms and being immediately executed (interpreted) by the platform where they arrive.

Messengers can also be seen as mobile agents, not necessarily intelligent ones, which collaborate in order to solve a common goal. We can also say that messengers are mobile threads being small parts of a whole process.

Properties we are interested to observe are among others: Is the system in a deadlock state?, What are the interactions between messengers?, How can we model a distributed algorithm ?, How can we deduce processes from the view of the messengers only?, etc.

The formal definitions we need concern the notions of *system*, *messengers*, *platforms*, etc.

This first attempt to formalization of the messenger paradigm leads to the definition of a *system state* and to a semantics expressed by the means of transition systems.

Chapter 2 presents the messenger paradigm and the basic elements of a messenger platform, chapter 3 presents the syntax and semantics of the messenger paradigm, finally chapter 4 contains some concluding remarks.

Chapter 2

The Messenger paradigm

The messenger paradigm was born as a result of a new way of thinking computer communications. As opposed to the classical sender/receiver model based on protocol entities that exchange and *interpret* specific protocol messages, Tschudin proposed in [8] a communication model based on the exchange of unspecific protocol programs, called *messengers*. Hosts receiving messengers would then *execute* the messengers' code instead of interpreting them as messages.

2.1 The Messenger Platform

All hosts involved in a messenger based communication share a common representation of the messengers and provide a local execution environment, *the messenger platform*. A messenger is executed sequentially,

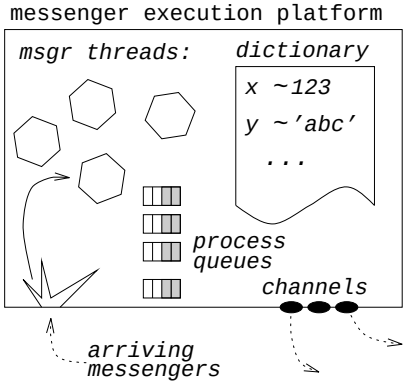

Figure 2.1: The elements of a messenger platform.

however several messengers may be executed in parallel inside the platform. Platforms are connected by unreliable channels through which messengers are sent as simple messages or data packets.
Figure 2.1 shows the basic elements of a messenger platform:

- A set of *channels* enables a messenger platform to exchange messengers with other platforms. The channels provide an unreliable datagram service which allows messengers either to reach the neighboring platform correctly, or to be discarded or lost;
- Messengers, having reached a platform in full, are turned into *independent messenger processes* or *anonymous threads of execution*, simply called *messengers*. The platform does not interfere with their execution except for identifying programming errors or unavailable resources. Nor do other messengers have the means to stop or kill a messenger process without its consent;

- The executing threads are able to share common data inside a given platform by the means of a global store, the *dictionary*, which contains pairs of keys and data values;
- *Process queues* represent the low-level mechanism offered by a platform in order to allow messengers to implement basic concurrency control functionality such as synchronization of messengers, mutual exclusion, etc. Process queues are FIFO queues of messenger processes. All messengers of a queue are blocked except the messenger at the head of the queue.

Platforms share (1) a common messenger programming language expressing messenger behavior; and (2) a common external representation of the messenger, used for the physical exchange of messengers.

2.2 Primitives of a Messenger Programming Language

Messengers are exchanged between two platforms as simple messages using a common external representation. Arriving messengers are turned into threads of execution by the corresponding platforms. A messenger is, accordingly, a sequence of instructions expressed in a *messenger programming language* common to all platforms. Beside general purpose instructions (arithmetic operations, logical operations, etc), a messenger programming language must provide a set of specialized primitives, which may appear as follows:

- Process queues, channels, as well as global variables in the dictionary of a given platform, are accessed with a key. Keys are either generated by the platform (e.g., name of a channel) or can be constructed/chosen by messenger processes using the `key()` primitive;
- Global variables in the dictionary are accessed through their key using the primitives `get(k:key)` and `set(k:key, v:value)`;
- Process queues play a central role in the synchronization of messenger processes. Primitives for handling process queues are: (1) `enter(q:key)` enables a messenger process to enter a queue (such a process will then be blocked until it reaches the head of the queue); (2) `leave` enables the messenger process at the head of the queue to remove itself from the queue; (3) `stop(q:key)` halts the queue so that when the messenger process at the head of the queue leaves the queue, the next messenger process coming at the head of the queue will remain blocked; and (4) `start(q:key)` resumes the queue previously stopped by the `stop` primitive. A referenced queue that does not yet exist is automatically created by the platform;
- A messenger is sent through a channel to another platform by `submit(k:key, m:msgr_descr)` primitive, so that a new messenger process with code `m` is ready to execute as soon as it arrives in the platform connected to the current platform through channel `k`. It is also possible to create a new local messenger process by submitting a messenger to a special "loop-back" channel – the new process is completely independent of its launching process;
- A messenger is able to alter its behavior by replacing it with another behavior using the primitive `chain(m:msgr_descr)`.

Through the use of the `submit` and `chain` primitives, messengers are able to act as mobile entities that can propel themselves across the network in order to search for given information or to perform a given task in a remote place. Alternatively, a messenger can continue its task in the local platform while the messengers it sent out will execute remotely and return with the result of some sub-task.

The possibility for a messenger process to replace its behavior with the `chain` primitive can be compared with the change of behavior of actors in the actor model of Agha [1]. The difference resides in the fact that the change of behavior is a fundamental part of the actor model: actors are seen as abstract machines dedicated to processing one incoming communication. Actors always change, explicitly or inexplicitly, their behavior before processing the subsequent communication. Messengers, however, are mobile entities that perform given tasks, not necessarily reactions to incoming communications.

Example 2.1 *Execution of Some Primitives.*
Figure 2.2 depicts two platforms P, P'. Platform P is linked to platform P' through channel c. A messenger is being executing in platform P and performs some instruction. The upper part of the figure shows the two platforms before the execution of some messenger instructions, while the lower part shows the two platforms after 3 instructions have been executed: `set(x,'beg')`, `submit(c,(...))`, `enter(q)`.

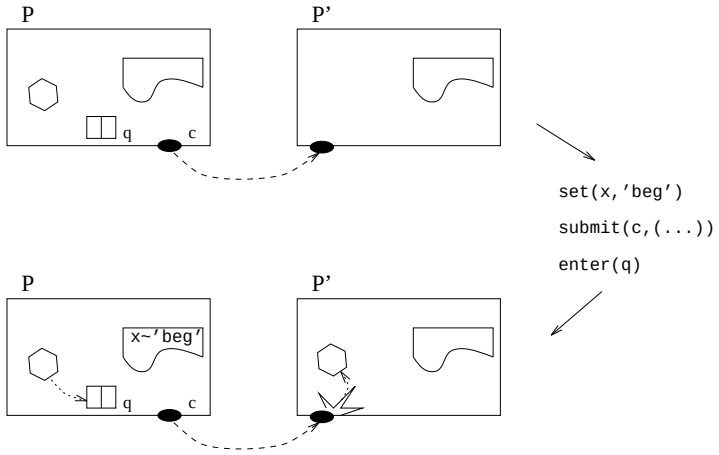

Figure 2.2: The `set, submit, enter` Primitives

These 3 instructions are executed by the messenger in platform P. The first one results in the adjunction in the dictionary of the value `'beg'` *stored under key* `x`. *The second instruction creates a new messenger in the platform P'. The code of this new messenger is not represented here. With the last instruction, the messenger executing in platform P enters process queue q.*

2.3 To Know More About Messengers

Currently two languages implement the above primitives, the `MO` language [9] with a PostScript like taste, and the MSGR-S language [12], an interpreter for an extended version of the SCHEME language.

Messengers demonstrated to be well suited for the implementation of protocol stacks [10]. Research is under work to built a distributed operating system [11], as well as for the implementation of distributed services using messengers [7].

Messengers are very close to agents [5], as they are mobile scipting code traveling across the network.

Chapter 3

Mathematical Formalization of the Messenger Paradigm

We will formalize the messenger paradigm as it has been described in [8].

We will not address, for the moment, problems like: what messengers are involved in a given process, what is a family of messengers and which messengers belong to it, etc.

The syntax and semantics, described in this chapter, concern systems of messengers executing concurrently and in parallel across different platforms. The semantics is given by the means of transition systems. Platforms are not considered to be dynamic, i.e. we have a fixed network of interconnected platforms, platforms cannot be added or removed dynamically.

3.1 Preliminary definitions

Definition 3.1 *Multiset.*
Let A be a set, a multiset over A, denoted $M(A)$ is given by the following function:

$$M(A) : A \rightarrow \mathcal{N}$$

Elements of the multiset $M(A)$ are those of the set A. In the multiset, there can be possibly multiple occurrences of a given element. For $a \in A$, $M(A)(a)$ is a natural number representing the occurrences of a in A.

3.2 Syntax of Messengers

The syntax of the messenger paradigm leads to the definition of a *system state*. A system is a set of platforms together with the state of their components: data, queues and messenger processes.

Data is a collection of pairs $(key, value)$, and a queue is an ordered list of messenger processes together with a mark indicating if the queue is stopped or not.

Messenger processes are messengers in execution. More precisely they are made of a pair of messenger codes (two ordered lists of messenger instructions), the first code standing for the already executed code and the second one for the remaining code to execute.

Messenger instructions are of different kinds. We have (1) silent instructions, i.e. internal instructions of messengers having no effect on a platform, (2) instructions concerned with the dictionary (store and retrieve of data), (3) instructions related to the synchronization of messengers using the process queues, and finally (4) instructions enabling messengers to act not only in the platform where they are executing but also in other platforms, by sending themselves or other messenger codes to other platforms.

Definition 3.2 *Universal Sets.*
We denote by:

1.	KEY,	*the set of all possible keys* k, $k \in KEY$, *i.e. the set of all possible names: variable names for data, variable names for queues, variables names for channels, names of queues, names of channels;*
2.	$Q \subseteq KEY$,	*the set of all possible queue names* q, $q \in Q$;
3.	$C \subseteq KEY$,	*the set of all possible channel names* c, $c \in C$;
4.	P, *with* $P \cap KEY = \emptyset$,	*the set of all possible platform names* p, $p \in P$;
5.	$VALUE$,	*the set of all possible value* v *of any type,* $v \in VALUE$.

In the set KEY we find all possible variable names (for data, queues and channels) and all possible names (for queues and for channels), except names for platforms. A value can be of any type including messenger codes, channel or queue names.

Messenger platforms are interconnected through (unreliable) channels. We call *network of platforms* the network resulting from the relations induced by the channel links between the platforms.

Definition 3.3 *Network of Platforms.*
A network of platforms *is a relation* N *such that:*

1. $N \subseteq P \times C \times P$
2. $\forall (p_i, c, p_j), (p'_i, c, p'_j) \in N \quad \Rightarrow p_i = p'_i, p_j = p'_j$
3. $\forall (p_i, c, p_j), (p_i, c', p_j) \in N \quad \Rightarrow c = c'$

Platforms are linked through unidirectional channels. A given channel links at most two platforms, and two platforms are linked by at most one channel in each direction.

Example 3.4 *A Network of Platforms.*
Figure 3.1 shows 3 platforms P_1, P_2, P_3 *interconnected through several channels.*

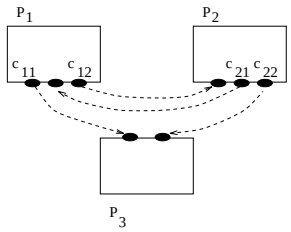

Figure 3.1: Interconnected Platforms

The resulting network of platforms N *is given by:*

1. $P = \{P_1, P_2, P_3\}, C = \{c_{11}, c_{12}, c_{21}, c_{22}\}$
2. $N \subseteq P \times C \times P$
3. $N = \{(P_1, c_{11}, P_3), (P_1, c_{12}, P_2), (P_2, c_{21}, P_2), (P_2, c_{22}, P_3)\}$

Definition 3.5 *Messenger Instructions.*

The set of all possible instructions, *I, that messengers can perform, is the least set recursively defined as:*

1. *Global Store*
   `set(k,v)` $\in I$
   `get(k)` $\in I$
2. *Silent Step: Internal Processing of data values*
   τ
3. *Synchronization*
   `enter(q)` $\in I$
   `leave` $\in I$
   `stop(q)` $\in I$
   `start(q)` $\in I$
4. *New Messenger Creation*
   $\forall i_1, \ldots, i_n \in I, n \in \mathcal{N} \Rightarrow$ `chain(`$i_1, \ldots, i_n$`)` $\in I$
   $\forall i_1, \ldots, i_n \in I, n \in \mathcal{N} \Rightarrow$ `submit(local,` $(i_1, \ldots, i_n)$`)` $\in I$
   $\forall i_1, \ldots, i_n \in I, n \in \mathcal{N} \Rightarrow$ `submit(c,`$(i_1, \ldots, i_n)$`)` $\in I$

Where `k` $\in KEY$ *stands for any key (variable name, queue name, or channel name),* `v` $\in VALUE$ *stands for any data value of any type,* `q` $\in KEY$ *stands for any queue name or queue variable,* `local` *is a reserved word, and* `c` $\in C$ *stands for any channel name.*

Instructions are of three different kinds.

Messengers perform instructions of the first kind to modify the state of data values stored in the global store of the platform, or to get values stored under a given key.

Messengers are also able to perform any instruction related to the processing of data values, e.g. arithmetic operations, use of variables, etc. However messengers are not supposed to realize input/output operations, e.g. read or write in a file. All these instructions, enable messengers to perform data processing, without altering any element in the platforms of the network. All these instructions are considered as silent steps and referenced by τ. Note that instruction `get(k)` can also be considered as a silent step, because it does not affect the state of the platform, but affects only the internal behavior of the messenger process.

Messengers perform instructions of the third kind to synchronize their execution using the mechanism of queues.

Instructions of the fourth kind are instructions a messenger perform to modify its behavior, or to create another executing messenger in the current platform or in any other platform linked by a channel to the current platform. This instruction enables a messenger to move across the network by sending its remaining code to execute to another platform.

Definition 3.6 *Messenger Codes.*
The set of all possible messenger codes, *$Code$, is the set of all finite (possibly empty) ordered sequences of instructions:*

$$Code = \{\emptyset\} \cup \{(i_1, \ldots, i_n) \mid n \in \mathcal{N}, i_j \in I, j = \{1, \ldots, n\}\}$$

Example 3.7 *Messenger Codes.*

Some examples of messenger codes can be given by:

$$
\begin{array}{lll}
code_1 & = (& \texttt{stop(q)}, \\
 & & \texttt{submit(c, (set(x,'beg'), submit(c', (start(q)))))}, \\
 & & \texttt{enter(q)}, \\
 & & \texttt{leave)}
\end{array}
$$

$$
\begin{array}{lll}
code_2 & = (& \texttt{set(my\_res, 0)}, \\
 & & \texttt{set(my\_code, (set(my\_res, get(my\_res) + 1),} \\
 & & \qquad\qquad \texttt{my\_chain := get(my\_code)}, \\
 & & \qquad\qquad \texttt{chain(my\_chain)))}, \\
 & & \texttt{my\_chain := get(my\_code)}, \\
 & & \texttt{chain(my\_chain))}
\end{array}
$$

The messenger process whose code is the first messenger code, $code_1 \in Code$, firstly stops a queue named q, it then submits through a channel named c a messenger code, enters the previously stopped queue q waiting to be awakened by another messenger process. Once it has been awakened, it then leaves the queue. The messenger process whose code has been submitted through channel c, as it arrives, in the platform linked to the current platform with channel c, will `set` the key x with value $'beg'$, and then sends through channel c' a messenger whose code is to start a queue named q. If c, c' are two channels linking two platforms in two diverse directions, then the code $start(q)$ will restart the queue where the first messenger is waiting. $code_1$ shows an example of a messenger process submitting a messenger to another platform in order to perform a given task and waiting in a queue, for the result of this messenger.

The messenger process whose code is the second messenger code, $code_2 \in Code$, performs a loop to increment the value stored under key `my_res`. The messenger process firstly initializes with value 0 the key `my_res`, it then stores a messenger code in the global store under the key `my_code`, finally it takes this code from the global store and changes its behavior with this code. Note that the instruction `my_chain:=get(my_code)` is a silent step. The code stored under key `my_code` is dedicated to increment the value of `my_res`. This is realized by an infinite sequence of two instructions: (1) to increment the value, (2) a change of behavior, the new behavior will be the previous behavior. This continuous change of behavior realizes the loop.

Definition 3.8 *Messenger Processes.*
A messenger process *is a triple*

$$msgr = (code_{beg}, code_{end}, \texttt{st}) \text{ where } code_{beg}, code_{end} \in Code, \texttt{st} \in \{\texttt{stopped}, \texttt{idle}\}$$

The set of all messenger processes is denoted by $MSGR \subseteq Code \times Code \times \{\texttt{stopped}, \texttt{idle}\}$.

A messenger process is a messenger in execution, where $code_{beg}$ stands for the portion of code already executed by the messenger, and $code_{end}$ stands for the remaining code to execute. Both portions of code $code_{beg}$ or $code_{end}$ can be the empty code $\emptyset$. The mark `st` is used when the messenger process is at the head of a queue. It means that the messenger process is idle and ready to execute if `st` is `idle` and that the messenger process is stopped if `st` is `stopped`.

Example 3.9 *Messenger Processes.*

Let $code_1$ be the messenger code of example 3.7. Consider the following messenger processes:

1. $msgr_1 = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{beg} = (\emptyset)$
   $code_{end} = (code_1)$

2. $msgr_2 = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{beg} = (code_1)$
   $code_{end} = (\emptyset)$

3. $msgr_2 = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{beg} = (\texttt{stop(q)}, \texttt{submit(c,(set(x,'beg'),submit(c',(start(q))))))}$
   $code_{end} = (\texttt{enter(q)}, \texttt{leave})$

The first messenger process, $msgr_1$, is `idle`, i.e. ready to execute. It has an empty $code_{beg}$ part which means that the messenger process has executed no instruction yet. It is at the beginning of its execution. The next instruction to execute will be `stop(q)`.

The second messenger process, $msgr_2$, stands for a messenger process, having the same code than $msgr_1$ but it has completely ended its execution, as its $code_{end}$ part is empty.

The third messenger process, $msgr_3$, is in the course of its execution. It has begun but not yet finished and the next instruction to perform will be `enter(q)`.

Definition 3.10 *Data state.*
The state of a data $d \in KEY$ *is given by* $s_d = (d, state)$ *where* $d \in KEY$ *is the data name and* $state \in VALUE$ *can be any value.*

A data state is given by the name of the data, i. e. the key d, and by the value associated to the key.

Definition 3.11 *Queue state.*
The state of a queue $q \in Q$ *is given by* $s_q = (q, state)$ *where* $q \in Q$ *is the queue name, and* $state \in MSGR^* \times \{\texttt{stopped}, \texttt{idle}\}$.

A queue state is given by the name of the queue, the key q, by its content, i.e. an ordered list (possibly infinite or empty) of messenger processes and by the mention `stopped` or `idle`.

Definition 3.12 *Platform state.*
The state of a platform $p \in P$ *is given by* $s_p = (p, data, queues, m)$ *where:*

1. $p \in P$ *is the platform name*
2. *data is a set of data states s.t.*
   $\forall (d, state), (d, state') \in data \Rightarrow state = state'$
3. *queues is a set of queue states s.t.*
   $\forall (q, state), (q, state') \in queues \Rightarrow state = state'$
4. $m \in M(MSGR)$ *is a multiset of mesenger processes.*

A platform state is given by the name of the platform, the set of all data names together with their value, present in the platform, i.e. appearing in the global store, the set of all queue names together with their state, present in the platform, and the multiset of messenger processes being executed by the platform. The multiset of messenger processes m is the multiset of messenger processes currently being executed in the platform. Messenger processes inserted in queues are not present in m but are in the queue states *queues*.

We use multisets of messengers, instead of simple sets. The reason for this is that messengers are *anonymous*, i.e. there is no naming of messengers, therefore it is possible to have in the same platform two messenger processes made of the same code and being at the same point of their execution.

A system is given by the state of all the involved platforms, i.e. the state of all data, queues appearing in the platforms and all messenger processes executing inside each platform. We can see the network of platforms into two different manners: either we assume that the network of platforms is fixed and does not change during the whole life of the system, or we assume that new platforms can be added or removed to the network during the life of the system. In the second case, the network of platforms itself is a part of the system. For the moment, we assume that we are in the first case, where all platforms are well known from the beginning and that their interconnections are fixed. Therefore, we suppose no dynamicity at the level of the platforms.

Definition 3.13 *System State.*
Let $N \subseteq P_N \times C_N \times P_N$ be a network of platforms, with $P_N \subseteq P$ and $C_N \subseteq C$ be a set of platform names and a set of channel names respectively, a system state for N is given by S_N, a set of platform states, such that:

1. $(p, data, queues, m) \in S_N \Leftrightarrow p \in P_N$
2. $\forall (p, data, queues, m), (p, data', queues', m') \in S_N \Rightarrow data = data', queues = queues', m = m'$

A system state is a set of platform states, *exactly one* for each platform $p \in P_N$. The system state gives for each platform the state of its components: the data, the queues, the messenger processes executing outside any queues.

Example 3.14 *Data, Queue, Platform and System States.*
Let $N = \{(p, c, p'), (p', c', p)\}$ be a network of platforms made of two platforms, p, p', connected through two channels: c from p to p', and c' from p' to p.

Examples of states for data, queues and systems can be:

1. *Examples of Data States*
   $(code\_to\_use, (stop(q)))$
   $(int\_to\_use, 127)$
   $(channel\_to\_use, c)$
   $(queue\_to\_use, q')$
2. *Examples of Queue States*
   $(q, ((msgr_a, msgrb), \texttt{stop}))$
   $(q', (\emptyset, \texttt{idle}))$
3. *Examples of Platform States*
   $(p, data, queues, m)$ *where :*
   $data = \{(code\_to\_use, (stop(q))), (channel\_to\_use, c)\}$
   $queues = \{(q, ((msgr_a, msgrb), \texttt{stopped}))\}$
   $m = (msgr_a, msgr_c, msgr_c)$
   $(p', data', queues', m')$ *where*
   $data' = \{(int\_to\_use, 127), (queue\_to\_use, q')\}$
   $queues' = \{(q, (msgr_b, \texttt{stopped}))\}$
   $m' = (msgr_a)$
4. *Examples of System States*
   $S_N = \{(p, data, queues, m), (p', data', queues', m')\}$

The system state is made of two tuples, one for each platform of the network N.

For platform state of platform p, the *data* part contains two values in the global store: a messenger code ($stop(q)$) under key *code_to_use* and a channel name c under key *channel_to_use*; the queues part contains only one queue state, for queue q, a stopped queue, containing two messenger processes named $msgr_a$, $msgr_b$, with $msgr_a$ at the head of the queue. The multiset of messenger processes currently executing in platform p consists in one occurrence of $msgr_a$, and two occurrences of a messenger process $msgr_c$. It is to note that the occurrence of $msgr_a$ belonging to multiset m and the occurrence of $msgr_a$

lying at the head of queue q are two different messenger processes, with the same messenger code, the same mark, and being at the same point of their execution.

For platform state of platform p', the $data'$ part contains an integer 127 stored under key int_to_use, and a queue name q' stored under key $queue\_to\_use$; the $queues'$ part contains only one stopped queue q with messenger process $msgr_b$ at the head of the queue; the multiset of messenger processes executing in the platform p' contains only one messenger process $msgr_a$.

3.3 Operational Semantics of Messengers

The semantics is given by the means of transition systems, which are triples made of two system states and one next possible instruction.

Definition 3.15 *Transition System.*
Let $N = P_N \times C_N \times P_N$ be a network of platforms, a transition system for N, is a set of triples:

$$TRS_N \subseteq S \times I \times S$$

where S is a set of system states for N:

$$S = \{S_N \,|\, S_N \text{ is a system state for } N\}$$

A transition in the transition system is a triple (a, i, b), made of two system states and a messenger instruction. The system state changes, when an instruction is executed by a messenger in one of the platforms of the network.

Definition 3.16 *Next Possible Messenger Instruction.*
Let $N = P_N \times C_N \times P_N$ be a network of platforms, let a be a system state, let i be a messenger instruction, i is a next possible *messenger instruction that can be executed from a, iff:*

$$\begin{aligned}
&\exists msgr = (code_{beg}, code_{end}, \texttt{idle}) \in MSGR \text{ with}\\
&\quad 1.\quad code_{end} = (i, code'_{end}) \text{ and}\\
&\quad 2.\quad \exists p \in P_N,\ s.t.\ s_p = (p, data, queues, m) \in a \text{ and}\\
&\qquad a)\quad msgr \in m \text{ or}\\
&\qquad b)\quad \exists (q, state) \in queues \text{ s.t.}\\
&\qquad\qquad state = ((msgr, \ldots, msgr_i, \ldots), \texttt{st}), \texttt{st} \in \{\texttt{stopped}, \texttt{idle}\}
\end{aligned}$$

A next possible messenger instruction that the system is able to execute, is the next instruction that an idle messenger process has to execute. This messenger process is currently executing in one of the platforms either outside any queue or at the head of a `stopped` or `idle` queue. Such a messenger cannot be in a queue without being at the head of the queue, because in this case, the messenger would be blocked, and it could not execute the next instruction of its remaining code. It goes the same for a stopped messenger at the head of a queue: as it is stopped it cannot execute any instruction.

Given i a next possible messenger instruction, there are one or more messenger processes that are idle and ready to execute instruction i.

Example 3.17 *Next Possible Messenger Instructions.*
Consider the system state S_N of example 3.14. Next possible instructions are all the next instructions that `idle` *messenger processes are ready to execute, i.e. the first instruction in the $code_{end}$ of (1) all messengers belonging to m or m' and of (2) all idle messengers being at the head of queues (stopped or not).*

If we have the following messenger processes:

1. $msgr_a = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{end} = (set(int\_to\_use, 0), start(q))$
2. $msgr_b = (code_{beg}, code_{end}, \texttt{stopped})$ *where:*
   $code_{end} = (leave)$
3. $msgr_c = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{end} = (my\_code := get(code\_to\_use), submit(c, my\_code))$

All the next possible instructions of system state S_N are:

$set(int\_to\_use, 0)$, *from* $msgr_a$ *in* m
$set(int\_to\_use, 0)$, *from* $msgr_a$ *at the head of queue* q
$set(int\_to\_use, 0)$, *from* $msgr_a$ *in* m'
$my\_code := get(code\_to\_use)$ *from* $msgr_c$ *in* m
$my\_code := get(code\_to\_use)$ *from the other* $msgr_c$ *in* m

We have 2 different messenger processes executing inside the platforms p, p': $msgr_a$ with one occurrence in m and one in m', $msgr_c$ with two occurrences in m. We have also 2 different messenger processes inserted in the queues: $msgr_a$ at the head of queue q in platform p, $msgr_b$ in the same queue q, and $msgr_b$ at the head of queue q in platform p'.

The next possible instructions can be taken only in `idle` messenger processes. For that reason, the instruction *leave* of messenger process $msgr_b$ cannot be a next possible instruction as the occurrence of $msgr_b$ in queue q of platform p is not at the head of the queue, and the occurrence of $msgr_b$ at the head of queue q in platform p' has its mark set to `stopped`. The other messenger processes $msgr_a$ and $msgr_c$ are `idle`, the first instruction of their remaining code to execute is respectively $set(int\_to\_use, 0)$, and $my\_code := get(code\_to\_use)$. As we have 3 occurrences of $msgr_a$, there are 3 possibilities to have $set(int\_to\_use, 0)$ as next possible instruction, and as we have 2 occurrences of $msgr_c$, there are 2 possibilities to have the silent step $my\_code := get(code\_to\_use)$ as next possible instruction.

Definition 3.18 *New System State after the execution of a next possible messenger instruction.*
Let a be a system state, let i be a next possible messenger instruction and $msgr$ be a messenger process $msgr = (code_{beg}, (i, code_{end}), \texttt{idle})$ currently executing in a platform p of state $(p, data, queues, m)$, with $msgr \in m$ or $msgr$ at the head of a queue q with state $(q, ((msgr, m^*), \texttt{st})) \in queues$. After the execution of i from state a, and depending on the nature of instruction i, the new system state, denoted $b_{a,i}$, is obtained from a by modifying elements of a according to one of the following rules:

Global Store

$$\frac{i = \texttt{set(k,v)}, msgr \in m}{m \rightarrow m', data \rightarrow data \setminus \{(k, v')\} \cup \{(k, v)\}} \quad (1)$$

$$\frac{i = \texttt{set(k,v)}, (q, ((msgr, m^*), \texttt{st})) \in queues}{(q, ((msgr, m^*), \texttt{st})) \rightarrow (q, ((msgr', m^*), \texttt{st})), data \rightarrow data \setminus \{(k, v')\} \cup \{(k, v)\}} \quad (2)$$

$$\frac{i = \texttt{get(k)}, msgr \in m}{m \rightarrow m'} \quad (3)$$

$$\frac{i = \texttt{get(k)}, (q, ((msgr, m^*), \texttt{st})) \in queues}{(q, ((msgr, m^*), \texttt{st})) \rightarrow (q, ((msgr', m^*), \texttt{st}))} \quad (4)$$

Silent Step

$$\frac{i = \tau, msgr \in m}{m \rightarrow m'} \quad (5)$$

$$\frac{i = \tau, (q, ((msgr, m^*), \mathtt{st})) \in queues}{(q, ((msgr, m^*), \mathtt{st})) \rightarrow (q, ((msgr', m^*), \mathtt{st}))} \quad (6)$$

Synchronization: Entering in an existing queue

$$\frac{i = \mathtt{enter(q')}, msgr \in m, \exists (q', (m'^*, \mathtt{st})) \in queues}{m \rightarrow m \setminus \{msgr\}, (q', (m'^*, \mathtt{st})) \rightarrow (q', ((m'^*, msgr''), \mathtt{st}))} \quad (7)$$

$$\frac{\begin{array}{c} i = \mathtt{enter(q')}, (q, ((msgr, (code_{beg1}, code_{end1}, \mathtt{st}_1), m^*), \mathtt{st})) \in queues, \\ \exists (q', (m'^*, \mathtt{st})) \in queues \end{array}}{\begin{array}{c} (q, ((msgr, (code_{beg1}, code_{end1}, \mathtt{st}_1), m^*), \mathtt{st})) \rightarrow (q, ((code_{beg1}, code_{end1}, \mathtt{st}), m^*, \mathtt{st})), \\ (q', (m'^*, \mathtt{st})) \rightarrow (q', ((m'^*, msgr''), \mathtt{st})) \end{array}} \quad (8a)$$

$$\frac{i = \mathtt{enter(q')}, (q, (msgr, \mathtt{st})) \in queues, \exists (q', (m'^*, \mathtt{st})) \in queues}{(q, (msgr, \mathtt{st})) \rightarrow (q, (\emptyset, \mathtt{st})), (q', (m'^*, \mathtt{st})) \rightarrow (q', ((m'^*, msgr''), \mathtt{st}))} \quad (8b)$$

Synchronization: Entering in a non-existing queue

$$\frac{i = \mathtt{enter(q')}, msgr \in m, \nexists (q', (m'^*, \mathtt{st})) \in queues}{m \rightarrow m \setminus \{msgr\}, queues \rightarrow queues \cup \{(q', (msgr', \mathtt{idle}))\}} \quad (9)$$

$$\frac{\begin{array}{c} i = \mathtt{enter(q')}, (q, ((msgr, (code_{beg1}, code_{end1}, \mathtt{st}_1), m^*), \mathtt{st})) \in queues, \\ \nexists (q', (m'^*, \mathtt{st})) \in queues \end{array}}{\begin{array}{ll} queues \rightarrow queues \setminus & \{(q, ((msgr, (code_{beg1}, code_{end1}, \mathtt{st}_1), m^*), \mathtt{st}))\} \cup \\ & \{(q, ((code_{beg1}, code_{end1}, \mathtt{st}), m^*), \mathtt{st}), (q', (msgr', \mathtt{idle}))\} \end{array}} \quad (10a)$$

$$\frac{i = \mathtt{enter(q')}, (q, (msgr, \mathtt{st})) \in queues, \nexists (q', (m'^*, \mathtt{st})) \in queues}{queues \rightarrow queues \setminus \{(q, (msgr, \mathtt{st}))\} \cup \{(q, (\emptyset, \mathtt{st})), (q', (msgr', \mathtt{idle}))\}} \quad (10b)$$

Synchronization: Entering the queue the messenger is in

$$\frac{i = \mathtt{enter(q)}, (q, ((msgr, (code_{beg1}, code_{end1}, \mathtt{st}_1), m^*), \mathtt{st})) \in queues}{(q, ((msgr, (code_{beg1}, code_{end1}, \mathtt{st}_1), m^*), \mathtt{st})) \rightarrow (q, (((code_{beg1}, code_{end1}, \mathtt{st}), m^*, msgr''), \mathtt{st}))} \quad (11a)$$

$$\frac{i = \mathtt{enter(q)}, (q, (msgr, \mathtt{st})) \in queues}{(q, (msgr, \mathtt{st})) \rightarrow (q, (msgr'', \mathtt{st}))} \quad (11b)$$

Synchronization: Leaving a queue

$$\frac{i = \mathtt{leave}, msgr \in m}{m \rightarrow m'} \quad (12)$$

$$\frac{i = \mathtt{leave}, (q, ((msgr, (code_{beg1}, code_{end1}, \mathtt{st}_1), m^*), \mathtt{st})) \in queues}{\begin{array}{c} m \rightarrow m \cup \{msgr'\}, \\ (q, ((msgr, (code_{beg1}, code_{end1}, \mathtt{st}_1), m^*), \mathtt{st})) \rightarrow (q, (((code_{beg1}, code_{end1}, \mathtt{st}), m^*), \mathtt{st})) \end{array}} \quad (13a)$$

$$\frac{i = \mathtt{leave}, (q, (msgr, \mathtt{st})) \in queues}{\begin{array}{c} m \rightarrow m \cup \{msgr'\}, \\ (q, (msgr, \mathtt{st})) \rightarrow (q, (\emptyset, \mathtt{st})) \end{array}} \quad (13b)$$

Synchronization: Stopping an existing queue

$$\frac{i = \texttt{stop(q')}, msgr \in m, \exists (q', (m'^*, \texttt{st})) \in queues}{m \rightarrow m', (q', (m'^*, \texttt{st})) \rightarrow \{(q', (m'^*, \texttt{stopped}))} \quad (14)$$

$$\frac{i = \texttt{stop(q')}, (q, ((msgr, m^*), \texttt{st})) \in queues, \exists (q', (m'^*, \texttt{st})) \in queues}{(q, ((msgr, m^*), \texttt{st})) \rightarrow (q, ((msgr', m^*), \texttt{st})), (q', (m'^*, \texttt{st})) \rightarrow (q', (m'^*, \texttt{stopped}))\}} \quad (15)$$

Synchronization: Stopping a non-existing queue

$$\frac{i = \texttt{stop(q')}, msgr \in m, \nexists (q', (m'^*, \texttt{st})) \in queues}{m \rightarrow m', queues \rightarrow queues \cup \{(q', (\emptyset, \texttt{stopped}))\}} \quad (16)$$

$$\frac{i = \texttt{stop(q')}, (q, ((msgr, m^*), \texttt{st})) \in queues, \nexists (q', (m'^*, \texttt{st})) \in queues}{queues \rightarrow queues \setminus \{(q, ((msgr, m^*), \texttt{st}))\} \cup \{(q, (msgr', m^*, \texttt{st})), (q', (\emptyset, \texttt{stopped}))\}} \quad (17)$$

Synchronization: Stopping the queue the messenger is in

$$\frac{i = \texttt{stop(q)}, (q, ((msgr, m^*), \texttt{st})) \in queues}{(q, ((msgr, m^*), \texttt{st})) \rightarrow (q, (msgr', m^*, \texttt{stopped}))} \quad (18)$$

Synchronization: Starting an existing queue

$$\frac{i = \texttt{start(q')}, msgr \in m, \exists (q', ((code_{beg1}, code_{end1}, \texttt{st}_1, m'^*), \texttt{st})) \in queues}{m \rightarrow m', (q', ((code_{beg1}, code_{end1}, \texttt{st}_1, m'^*), \texttt{st}))\} \rightarrow (q', ((code_{beg1}, code_{end1}, \texttt{idle}, m'^*), \texttt{idle}))} \quad (19a)$$

$$\frac{i = \texttt{start(q')}, msgr \in m, \exists (q', (\emptyset, \texttt{st})) \in queues}{m \rightarrow m', (q', (\emptyset, \texttt{st}))\} \rightarrow (q', (\emptyset, \texttt{idle}))} \quad (19b)$$

$$\frac{\begin{array}{c} i = \texttt{start(q')}, (q, ((msgr, m^*), \texttt{st})) \in queue, \\ \exists (q', ((code_{beg1}, code_{end1}, \texttt{st}_1, m'^*), \texttt{st})) \in queues \end{array}}{\begin{array}{c} (q, ((msgr, m^*), \texttt{st})) \rightarrow (q, (msgr', m^*, \texttt{st})), \\ (q', ((code_{beg1}, code_{end1}, \texttt{st}_1, m'^*), \texttt{st}))\} \rightarrow (q', ((code_{beg1}, code_{end1}, \texttt{idle}, m'^*), \texttt{idle})) \end{array}} \quad (20a)$$

$$\frac{\begin{array}{c} i = \texttt{start(q')}, (q, ((msgr, m^*), \texttt{st})) \in queue, \\ \exists (q', (\emptyset, \texttt{st})) \in queues \end{array}}{\begin{array}{c} (q, ((msgr, m^*), \texttt{st})) \rightarrow (q, (msgr', m^*, \texttt{st})), \\ (q', (\emptyset, \texttt{st}))\} \rightarrow (q', (\emptyset, \texttt{idle})) \end{array}} \quad (20b)$$

Synchronization: Starting an non-existing queue

$$\frac{i = \texttt{start(q')}, msgr \in m, \nexists (q', (m'^*, \texttt{st})) \in queues}{m \rightarrow m', queues \rightarrow queues \cup \{(q', (\emptyset, \texttt{idle}))\}} \quad (21)$$

$$\frac{i = \texttt{start(q')}, (q, ((msgr, m^*), \texttt{st})) \in queue, \nexists (q', (m^*, \texttt{st})) \in queues}{queues \rightarrow queues \setminus \{(q, ((msgr, m^*), \texttt{st}))\} \cup \{(q, (msgr', m^*, \texttt{st})), (q', (\emptyset, \texttt{idle}))\}} \quad (22)$$

Synchronization: Starting the queue the messenger is in

$$\frac{i = \texttt{start(q)}, (q, ((msgr, m^*), \texttt{st})) \in queues}{(q, ((msgr, m^*), \texttt{st})) \rightarrow (q, (msgr', m^*, \texttt{idle}))} \quad (23)$$

New Messenger Creation: The messenger changes its code

$$\frac{i = \texttt{chain(code)}, msgr \in m}{msgr \rightarrow (\emptyset, code, \texttt{idle})} \quad (24)$$

$$\frac{i = \texttt{chain(code)}, (q, ((msgr, m^*), \texttt{st})) \in queues}{(q, ((msgr, m^*), \texttt{st})) \rightarrow (q, (((\emptyset, code, \texttt{idle}), m^*), \texttt{st}))\}} \quad (25)$$

New Messenger Creation: The messenger creates locally a new messenger process

$$\frac{i = \texttt{submit(local, code)}, msgr \in m}{m \rightarrow m' \cup \{(\emptyset, code, \texttt{idle})\}} \quad (26)$$

$$\frac{i = \texttt{submit(local, code)}, (q, ((msgr, m^*), \texttt{st})) \in queues}{m \rightarrow m \cup \{(\emptyset, code, \texttt{idle})\}, (q, ((msgr, m^*), \texttt{st})) \rightarrow (q, ((msgr', m^*), \texttt{idle}))} \quad (27)$$

New Messenger Creation: The messenger creates remotely a new messenger process over an unreliable channel

$$\frac{i = \texttt{submit(c,code)}, msgr \in m, \exists (p, c, p_c) \in N, (p_c, data_c, queues_c, m_c) \in a}{m \rightarrow m', m_c \rightarrow m_c \cup \{(\emptyset, code, \texttt{idle})\}} \quad (28)$$

$$\frac{i = \texttt{submit(c,code)}, (q, ((msgr, m^*), \texttt{st})) \in queues, \exists (p, c, p_c) \in N, (p_c, data_c, queues_c, m_c) \in a}{\begin{array}{c}(q, ((msgr, m^*), \texttt{st})) \rightarrow (q, (msgr', m^*, \texttt{idle})), \\ m_c \rightarrow m_c \cup \{(\emptyset, code, \texttt{idle})\}\end{array}} \quad (29)$$

$$\frac{i = \texttt{submit(c,code)}, msgr \in m}{m \rightarrow m'} \quad (30)$$

$$\frac{i = \texttt{submit(c,code)}, (q, ((msgr, m^*), \texttt{st})) \in queues}{(q, ((msgr, m^*), \texttt{st})) \rightarrow (q, (msgr', m^*, \texttt{idle}))} \quad (31)$$

Where

$$\begin{array}{ll} msgr & = (code_{beg}, (i, code_{end}), \texttt{idle}) \\ msgr' & = ((code_{beg}, i), code_{end}, \texttt{idle}) \\ msgr'' & = ((code_{beg}, i), code_{end}, \texttt{st}) \\ m' & = m \setminus \{msgr\} \cup \{msgr'\} \\ m^* & \in MSGR^* \\ \texttt{st} & \in \{\texttt{stopped}, \texttt{idle}\} \end{array}$$

By applying one of the rules (1) *to* (31)*, we obtain an intermediary system state* $tmp_{a,i}$*. The final system state,* $b_{a,i}$*, is obtained by applying to all* $(p, data_p, queues_p, m_p) \in tmp_{a,i}$*, as long as it is possible, the three following rules:*

$$\frac{msgr = (code_{beg}, \emptyset, \texttt{idle}), msgr \in m_p}{m_p \rightarrow m_p \setminus \{msgr\}} \quad (32)$$

$$\frac{msgr = (code_{beg}, \emptyset, \texttt{idle}), (q, ((msgr, (code_{beg1}, code_{end1}, \texttt{st}_1), m^*), \texttt{st})) \in queues_p}{(q, ((msgr, (code_{beg1}, code_{end1}, \texttt{st}_1), m^*), \texttt{st})) \rightarrow (q, (((code_{beg1}, code_{end1}, \texttt{st}), m^*), \texttt{st}))} \quad (33a)$$

$$\frac{msgr = (code_{beg}, \emptyset, \texttt{idle}), (q, (msgr, \texttt{st})) \in queues_p}{(q, (msgr, \texttt{st})) \rightarrow (q, (\emptyset, \texttt{st}))} \quad (33b)$$

The new system state depends on the next possible instruction i to execute and on the messenger process $msgr$ that will execute instruction i. Two cases occur, either $msgr$ is in the multiset m of messenger processes executing in the platforms, or $msgr$ is at the head of a queue q and $msgr$ is idle. Notice that stopped messenger processes at the head of a queue cannot be ready to execute instruction i, while idle messenger processes at the head of stopped queues are ready to execute instruction i.

For any instruction i, and for both cases of $msgr$ being at the head of a queue or not, the messenger process $msgr$ will change. Most of the time, messenger process $msgr = (code_{beg}, (i, code_{end}), \texttt{idle})$ becomes $msgr' = ((code_{beg}, i), code_{end}, \texttt{idle})$ or $msgr'' = ((code_{beg}, i), code_{end}, \texttt{st})$ where instruction i moves from the portion of code to execute to the portion of code that has already been executed and the mark of the messenger process depends on the queue where it has moved. The exception comes from the `chain(code)` instruction, whose effect is to replace the code of $msgr$ by `code`, $msgr$ becomes the messenger process $(\emptyset, code, \texttt{idle})$.

As the executing messenger process $msgr$ changes, it results that (1) if $msgr$ is in m, then the multiset m of messengers will also change; and (2) if $msgr$ is at the head of a queue q, then the state of this queue will change.

The `set(k,v)` instruction changes the state of the global store $data$ by creating a new entry with key k and value v, if the key k was not known in the global store, or simply by changing the value associated to k, if k already exists in $data$ (rules $(1), (2)$).

The `get(k)` instruction does not imply any additional change to the system state as it gives $msgr$ the knowledge of the value stored under the key k (rules $(3), (4)$). If k is not known, nothing happens.

The silent step, τ, has exactly the same effect on the system state as the `get(k)` instruction, because a silent instruction stands for an internal processing of variables by the messenger $msgr$(rules $(5), (6)$).

The `enter(q')` instruction depends on whether the queue q' already exists (rules $(7), (8)$) in the platform or not (rules $(9), (10)$), and on the possibility for $msgr$ to enter the queue it is already in (rules (11)). In rule (7), $msgr$ disappears from the multiset m and appears at the end of queue q' with its new remaining code to execute and with the same mark as the mark of the queue. In rule (8), $msgr$ moves from the head of queue q to the end of queue q'. Rule $(9), (10)$ are similar to rules $(7), (8)$ respectively, the queue q' is created and $msgr$ moves to the newly created queue, where it is the unique messenger. The new created queue is `idle` by default. The last case is treated by rule (11), $msgr$ enters the queue where it is currently executing: $msgr$ will disappear from the head of the queue and appear to the end of the same queue. In rules $(8a), (10a), (11a)$, $msgr$ leaves a queue, and the next messenger process coming at the head of this queue takes the mark of the queue, i.e. if the queue is stopped (or idle) the messenger process coming at the head of the queue becomes stopped (or idle). Rules $(8b), (10b), (11b)$ treat the case of $msgr$ leaving a queue, where it is the unique messenger process. In the 5 rules (7) to (11) the messenger process inserted at the end of the queue takes the mark of the queue, i.e. if the queue q' is `idle`, then the inserted messenger process will be `idle` too, otherwise it will be `stopped`.

The `leave` instruction moves $msgr$ from the queue where it is to the multiset of messenger m (rule (13)). The messenger coming at the head of the queue, after $msgr$ has leaved, takes the mark of the queue (rule $(13a)$). If $msgr$ is not in a queue (rule (12)) nothing happens.

The `stop(q')` instruction depends on whether or not the queue q' exists and on the possibility for $msgr$ to stop the queue it is already in. If the queue q' exists, it is only stopped: the mark `st` standing for `idle` or `stopped` is replaced by `stopped` (rule $(14), (15)$). If the queue q' doesn't exist, it is created as a `stopped` queue with no messenger processes in (rules $(16), (17)$). Finally, if the stopped queue is the queue where $msgr$ is in, its mark is set to `stopped` (rule (18)). It is to notice here, that the messenger process at the head of the stopped queue doesn't change its mark, e.g. the messenger process remains `idle` if the queue was previously `idle`, it is not affected by the stop of the queue.

It goes the same for the `start(q')` instruction, except that the mark is set to `idle` instead of `stopped`, and that the messenger process at the head of the queue changes its mark to `idle` (rules $(19), (20)$).

The `chain(code)` instruction replaces $msgr$ by $(\emptyset, code, \texttt{idle})$ either in the multiset m (rule (24)) or in the queue q (rule (25)).

The `submit(local,code)` instruction adds a new messenger process $(\emptyset, code\texttt{idle})$ to the multiset of messenger processes m (rules $(26), (27)$).

The `submit(c,code)` instruction adds a new messenger process $(\emptyset, code\texttt{idle})$ to the multiset of messenger processes m_c of the platform p_c connected to p through channel c (rules (28), (29)). If such a platform does not exist, or a failure has occurred through channel c so that *code* never reaches platform p_c, nothing happens (rules (30), (31)).

The last two rules (32), (33) are used to throw out of the system, messenger processes that have ended their code, i.e. messenger processes of the form $(code_{beg}, \emptyset, \texttt{st})$. These two rules have to be applied after one of the rules (1) to (31) have been applied, and as long as they are applyable. Rule (32) removes all messenger processes that have ended their code from the multiset of messenger processes m , while rule (33) takes away from the head of each queue q messenger processes that have ended their code.

Remark 3.19 *It is possible, that two or more messenger processes are ready and idle to execute instruction i, the new system state $b_{a,i.}$ is, in a non deterministic way, one of the new system states obtained from a after the execution of instruction i by one of these messengers. Two messenger processes, ready and idle to execute i, can either have same or different codes. The new system state obtained by the execution of i from messenger processes having the same code and being outside any queues will be the same for all of these messenger processes, because messenger processes are* anonymous. *The new system state obtained by the execution of i from messenger processes with the same code and being at the head of different queues will be different, because queues are named queues.*

Example 3.20 *Next Possible Instruction and New System State.*
Let $N = \{(p, c, p'), (p', c', p)\}$ be a network of platforms made of two platforms p, p' connected through two channels: c from p to p', and c' from p' to p.

Consider the following system state $S_N = a$, $a = \{(p, data, queues, m), (p', data', queues', m')\}$ where:

$$
\begin{array}{l}
(p, data, queues, m) \text{ where :} \\
\qquad data \quad = \{(code\_to\_use, start(q))\} \\
\qquad queues \quad = \{(q, ((msgr_a, msgr_b), \texttt{stopped}))\} \\
\qquad m \quad = (msgr_a, msgr_c, msgr_c, msgr_d, msgr_f) \\
(p', data', queues', m') \text{ where} \\
\qquad data' \quad = \{(int\_to\_use, 127), (queue\_to\_use, q')\} \\
\qquad queues' \quad = \{(q, (msgr_b, \texttt{stopped}))\} \\
\qquad m' \quad = (msgr_a, msgr_e)
\end{array}
$$

The remaining codes of these messenger processes is the following:

1. $msgr_a = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{end} = (set(int\_to\_use, 0), start(q))$
2. $msgr_b = (code_{beg}, code_{end}, \texttt{stopped})$ *where:*
   $code_{end} = (leave)$
3. $msgr_c = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{end} = (my\_code := get(code\_to\_use), submit(c, my\_code))$
4. $msgr_d = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{end} = (enter(q''))$
5. $msgr_e = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{end} = (start(q))$
6. $msgr_f = (code_{beg}, code_{end}, \texttt{idle})$ *where:*
   $code_{end} = (submit(c, (start(q))))$

Global Store

If we take as next possible instruction i_1 from a, the instruction $set(int\_to\_use, 0)$ from $msgr_a$ in m, the new system state b_{a,i_1} is given by: $b_{a,i_1} = \{(p, data, queues, m), (p', data', queues', m')\}$ where:

$(p, data, queues, m)$ *where* :

$$\begin{aligned} data &= \{(code\_to\_use, start(q)), (int\_to\_use, 0)\} \\ queues &= \{(q, ((msgr_a, msgr_b), \texttt{stopped}))\} \\ m &= (msgr'_a, msgr_c, msgr_c, msgr_d, msgr_f) \end{aligned}$$

$(p', data', queues', m')$ *where*

$$\begin{aligned} data' &= \{(int\_to\_use, 127), (queue\_to\_use, q')\} \\ queues' &= \{(q, (msgr_b, \texttt{stopped}))\} \\ m' &= (msgr_a, msgr_e) \end{aligned}$$

Where $code_{end} = start(q)$ *for* $msgr'_a$.

The effect of instruction i_1 *adds a new entry in the global store,* int_to_use *with value* 0. *The remaining code of messenger process* $msgr_a$ *changes leading to a new messenger process* $msgr'_a$ *instead of* $msgr_a$ *in* m. *Note that the occurrence of* $msgr_a$ *at the head of queue* q *remains unchanged as this messenger process has executed no instruction.*

If we take as next possible instruction i_2 *from* a, *the instruction* $set(int\_to\_use, 0)$ *but from* $msgr_a$ *in* q, *instead of* $msgr_a$ *in* m, *the new system state is given by* $b_{a,i_2} = \{(p, data, queues, m), (p', data', queues', m')\}$ *where:*

$(p, data, queues, m)$ *where* :

$$\begin{aligned} data &= \{(code\_to\_use, start(q)), (int\_to\_use, 0)\} \\ queues &= \{(q, ((msgr'_a, msgr_b), \texttt{stopped}))\} \\ m &= (msgr_a, msgr_c, msgr_c, msgr_d, msgr_f) \end{aligned}$$

$(p', data', queues', m')$ *where*

$$\begin{aligned} data' &= \{(int\_to\_use, 127), (queue\_to\_use, q')\} \\ queues' &= \{(q, (msgr_b, \texttt{stopped}))\} \\ m' &= (msgr_a, msgr_e) \end{aligned}$$

Where $code_{end} = start(q)$ *for* $msgr'_a$.

Here the situation is similar to that of the previous example, except that it is the messenger process $msgr_a$ *at the head of queue* q *which changes its remaining code becoming messenger process* $msgr'_a$.

Silent Step

If we take as next possible instruction i_3 *from* a, *the instruction* $my\_code := get(code\_to\_use)$ *from* $msgr_c$ *in* m, *the new system state* b_{a,i_3} *is given by:* $b_{a,i_3} = \{(p, data, queues, m), (p', data', queues', m')\}$ *where:*

$(p, data, queues, m)$ *where* :

$$\begin{aligned} data &= \{(code\_to\_use, start(q))\} \\ queues &= \{(q, ((msgr_a, msgr_b), \texttt{stopped}))\} \\ m &= (msgr_a, msgr'_c, msgr_c, msgr_d, msgr_f) \end{aligned}$$

$(p', data', queues', m')$ *where*

$$\begin{aligned} data' &= \{(int\_to\_use, 127), (queue\_to\_use, q')\} \\ queues' &= \{(q, (msgr_b, \texttt{stopped}))\} \\ m' &= (msgr_a) \end{aligned}$$

Where $code_{end} = submit(c, my\_code)$ *for* $msgr'_c$.

The silent step changes in the system state only the messenger process that has executed the instruction, here it is one of the occurrences of $msgr_c$ *which becomes* $msgr'_c$.

Synchronization: Entering in an existing queue

If we take as next possible instruction i_4 *from* a, *the instruction* $enter(q'')$ *from* $msgr_d$ *in* m, *the new system state* b_{a,i_4} *is given by:* $b_{a,i_4} = \{(p, data, queues, m), (p', data', queues', m')\}$ *where:*

$$
\begin{array}{l}
(p, data, queues, m) \ where: \\
\qquad data \quad = \{(code\_to\_use, start(q))\} \\
\qquad queues \quad = \{(q, ((msgr_a, msgr_b), \texttt{stopped})), \\
\qquad\qquad\quad (q'', ((msgr'_d, \texttt{idle})))\} \\
\qquad m \quad = (msgr_a, msgr_c, msgr_c, msgr_f) \\
(p', data', queues', m') \ where \\
\qquad data' \quad = \{(int\_to\_use, 127), (queue\_to\_use, q')\} \\
\qquad queues' \quad = \{(q, (msgr_b, \texttt{stopped}))\} \\
\qquad m' \quad = (msgr_a, msgr_e)
\end{array}
$$

Where $code_{end} = \emptyset$ *for* $msgr'_d$*. We observe that messenger process* $msgr_d$ *disappears from multiset* m *and appears in a newly created idle queue* q''*. Note that as* $msgr_d$ *has performed its last instruction, and as it is at the head of queue, according to rule* (33b) *we have not* $(q'', ((msgr'_d, \texttt{idle})))$ *in queues but* $(q'', ((\emptyset, \texttt{idle})))$*.*

Synchronization: Starting an existing queue

If we take as next possible instruction i_5 *from* a*, the instruction* $start(q)$ *from* $msgr_e$ *in* m'*, the new system state* b_{a,i_5} *is given by:* $b_{a,i_5} = \{(p, data, queues, m), (p', data', queues', m')\}$ *where:*

$$
\begin{array}{l}
(p, data, queues, m) \ where: \\
\qquad data \quad = \{(code\_to\_use, start(q))\} \\
\qquad queues \quad = \{(q, ((msgr_a, msgr_b), \texttt{stopped}))\} \\
\qquad m \quad = (msgr_a, msgr_c, msgr_c, msgr_d, msgr_f) \\
(p', data', queues', m') \ where \\
\qquad data' \quad = \{(int\_to\_use, 127), (queue\_to\_use, q')\} \\
\qquad queues' \quad = \{(q, (msgr'_b, \texttt{idle}))\} \\
\qquad m' \quad = (msgr_a)
\end{array}
$$

Where $msgr'_b$ *has the same messenger code as* $msgr_b$*, but with mark* `idle` *instead of mark* `stopped`*. Messenger process* $msgr_e$ *disappears from* m' *according to rule* (32)*, because* $msgr_e$ *has just performed its last instruction.*

New Messenger Creation: The messenger creates remotely a new messenger process over an unreliable channel

If we take as next possible instruction i_6 *from* a*, the instruction* $submit(c, (start(q)))$ *from* $msgr_f$ *in* m*, the new system state* b_{a,i_6} *is given by:* $b_{a,i_6} = \{(p, data, queues, m), (p', data', queues', m')\}$ *where:*

$$
\begin{array}{l}
(p, data, queues, m) \ where: \\
\qquad data \quad = \{(code\_to\_use, start(q))\} \\
\qquad queues \quad = \{(q, ((msgr_a, msgr_b), \texttt{stopped}))\} \\
\qquad m \quad = (msgr_a, msgr_c, msgr_c, msgr_d) \\
(p', data', queues', m') \ where \\
\qquad data' \quad = \{(int\_to\_use, 127), (queue\_to\_use, q')\} \\
\qquad queues' \quad = \{(q, (msgr_b, \texttt{stopped}))\} \\
\qquad m' \quad = (msgr_a, msgr_e, (\emptyset, (start(q)), \texttt{idle}))
\end{array}
$$

In multiset m'*, we have one new messenger process with remaining code to execute is* $(start(q))$*, and which is at the beginning of its execution (it has executed no instruction yet).* $msgr_f$ *disappears from* m *because it has no more instructions to execute.*

If we take as next possible instruction i_6 from a, the same instruction $submit(c, my\_code)$ from $msgr_f$ in m, the new system state b_{a,i_6} can also be given by: $b_{a,i_6} = \{(p, data, queues, m), (p', data', queues', m')\}$ where:

$$
\begin{array}{l}
(p, data, queues, m) \text{ where :} \\
\qquad \begin{array}{ll}
data & = \{(code\_to\_use, start(q))\} \\
queues & = \{(q, ((msgr_a, msgr_b), \texttt{stopped}))\} \\
m & = (msgr_a, msgr_c, msgr_c, msgr_d)
\end{array} \\
(p', data', queues', m') \text{ where} \\
\qquad \begin{array}{ll}
data' & = \{(int\_to\_use, 127), (queue\_to\_use, q')\} \\
queues' & = \{(q, (msgr_b, \texttt{stopped}))\} \\
m' & = (msgr_a, msgr_e)
\end{array}
\end{array}
$$

In this case, the messenger code sent through channel c has been lost, so that nothing changes in platform p'. The $msgr_f$ ends its execution and disappears from m.

Definition 3.21 *Operational Semantics (Model).*
Let $N = P_N \times C_N \times P_N$ be a network of platforms, a transition system TRS_N is a model for N iff:

$\forall (a, i, b) \in TRS_N$

1. *i is a next possible instruction from a*
2. $b = b_{a,i}$

A transition system is a model if for each triple in the transition system, (1) the instruction part is a next possible instruction that can be executed from the initial state of the system and (2) the final system state is obtained from the initial one after the execution of the instruction i.

Example 3.22 *Models of Messenger Systems.*
Consider system state a of example 3.20, triples that can be part of a model for the network N are:

1. (a, i_1, b_{a,i_1})
2. $(a, i_2, b_{a,i_2}), etc.$

Chapter 4

Conclusion

This report states a mathematical definition of the messenger paradigm. The syntax gives the definition of a system state and the semantics is given in terms of transition system.

We have defined how messengers can (1) act locally on their own messenger platforms by changing values in the global store or by synchronizing their execution by the means of process queues, and (2) act on remote messenger platforms by sending messenger codes.

Further work will be concerned firstly with the formalization of the messenger paradigm using several well known formalisms related to mobility or to communication in distributed systems, in order to derive a specification language well suited to express messenger systems and their properties.

The formalisms we will use are among others: the CO-OPN2 [2] formalism, an extension of CO-OPN [3], which uses high-level algebraic Petri Nets, where tokens are not simple algebraic terms but whole Petri Nets, realizing in that way the mobility of processes; the π-calculus [6] derived from CCS and enabling to model mobile processes; the Actors [1] model, which is very similar to messenger as messenger can be seen as actors; and the POLIS [4] programming model of distributed systems with communication based on a shared dataspace.

The second focus of our further work is concerned with the extension of this mathematical definition to systems of messengers including the notions of families of messengers, services realized by the means of several messengers, mobility of messengers among services, systems of processes built with mobile threads, messengers collaborating to solve a common goal, exploration of the platform, interaction between messengers (even if it is a lack of interaction).

To realize this we will base our work on existing formalizations of processes built with threads and extending them to mobile threads, as well as on software agents, i.e. pieces of software that communicates and agent-based software engineering.

Bibliography

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Olivier Biberstein and Didier Buchs. Structured Algebraic Nets with Object-Orientation. In *Proceedings of the "Application and Theory of Petri Nets 1995" workshop on "Object-Oriented Programming and Models of Concurrency"*, Torino, Italy, June 1995.

[3] Didier Buchs and Nicolas Guelfi. A concurrent object oriented Petri nets approach for system specification. In M. Silva, editor, *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Aahrus, Denmark, June 1991.

[4] P. Ciancarini. Distributed Programming with Logic Tuple Spaces. Technical Report UBLCS-93-7, University of Bologna, 1993.

[5] G. Di Marzo and M. Muhugusa and C. F. Tshudin and J. Harms. The Messenger Paradigm and its Impact on Distributed Systems. In *ICC'95 workshop on Intelligent Computer Communication*, 1995.

[6] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes I and II. *Journal of Information and computation*, 100(1):1–40,41–77, 1992.

[7] M. Muhugusa, G. Di Marzo, C. Tschudin, and J. Harms. Distributed Semaphore in a Messenger Environment. In *Decentralized Intelligent and Multi-Agent Systems DIMAS 95.* Institute of Computer Science, AGH - Technical University of Mining and Metallurgy, Krakow, Poland, November 1995.

[8] C. F. Tschudin. *On the Structuring of Computer Communications.* PhD thesis, Université de Genève, 1993. Thèse No 2632.

[9] C. F. Tschudin. An Introduction to the M0 Messenger Language. Technical Report No 86 (Cahier du CUI), University of Geneva, 1994.

[10] C. F. Tschudin. Protokollimplementierung mit Kommunikationsboten. In *KiVS'95-Tagung, Chemnitz*, 1995.

[11] C. F. Tschudin, G. Di Marzo, M. Muhugusa, and J. Harms. Messenger-based Operating Systems. Technical Report No 90 (Cahier du CUI), University of Geneva, 1994.

[12] R. Lino Valverde. MSGR-S: Un environnement d'exécution de messagers basé sur un interpréteur Scheme parallèle. Diploma thesis, University of Geneva, 1994.