AN ENVIRONMENT SPECIFICATION LANGUAGE FOR MULTI-AGENT SYSTEMS

SALEM DA SILVA P

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

12/2009

Rapport de Recherche N° 1531

An Environment Specification Language for Multi-Agent Systems

Paulo Salem da Silva
salem@lri.fr
Laboratoire de Recherche en Informatique
Université Paris-Sud 11
Orsay, France

December 15, 2009

Abstract

Multi-agent systems are employed to model complex systems, which can be decomposed into several interacting pieces called agents. In such systems, agents exist, evolve and interact within an environment. In this report we present a language for the specification of such environments. This *Environment Specification Language* (ESL), as we call it, defines both structural and dynamic aspects of environments. Structurally, ESL connects agents by a social network, in which the link between agents is specified as the capability that one agent has to act upon another. Dynamically, ESL provides operations that can be composed together in order to create a number of different environmental situations and to respond appropriately to agents' actions. These features are founded on a mathematical model that we provide and that defines rigorously what constitutes an environment. Formality is achieved by employing the π-calculus process algebra in order to give the semantics of this model. This allows, in particular, a simple characterization of the evolution of the environment structure. Moreover, owing to this formal semantics, it is possible to perform formal analyses on environments thus described. For the sake of illustration, a concrete example of environment specification using ESL is also given.

Résumé

Les systèmes multi-agents sont utilisés pour modéliser des systèmes complexes, qui peuvent être décomposés en beaucoup d'entités interagissant qu'on appelle les agents. Dans ce genre de système, les agents existent, évoluent et interagissent au sein d'un environnement. Dans ce rapport, nous présentons un langage pour la spécification de ces environnements. Ce langage, que nous appelons *Environment Specification Language* (ESL), définit les aspects structurels ainsi que dynamiques des environnements. Structurellement, ESL lie les agents par un réseau social, où le lien entre les agents est la capacité que chacun a d'agir sur un autre. Dynamiquement, ESL fournit des opérations qui peuvent être composées ensemble afin de créer plusieurs situations environnementales et de répondre aux actions des agents. Ces caractéristiques sont fondées sur un modèle mathématique que nous fournissons et qui définit très précisément ce qui constitue un environnement. La base formelle est l'algèbre de processus π-calculus qui est utilisée pour donner la sémantique de ce modèle. Ceci permet, notamment, une caractérisation simple des évolutions de la structure de l'environnement. En outre, grâce à cette sémantique formelle, il est possible d'effectuer des analyses formelles sur les environnements ainsi décrits. Pour illustrer notre approche, nous donnons aussi un exemple concret d'utilisation d'ESL.

Contents

1 Introduction

Multi-agent systems (MAS) [13] are useful to model complex systems in which the entities to be studied can be decomposed into several interacting pieces called *agents*. Human societies, computer networks, neural tissue and cell biology are examples of systems that can be seen from this perspective. Given a MAS, one technique often employed to study it is simulation [4]. That is, one may implement the several agents of interest, compose them into a MAS, and then run simulations in order to analyze their dynamic behavior. In such works, the analysis method of choice is usually the collection or optimization of statistics over several runs (e.g., the mean value of a numeric variable over time). Examples of this approach include platforms such as Swarm [7], MASON [5] and Repast [8]. There are, however, other possibilities for analyzing such simulations. The crucial insight here is that simulations can be seen as incomplete

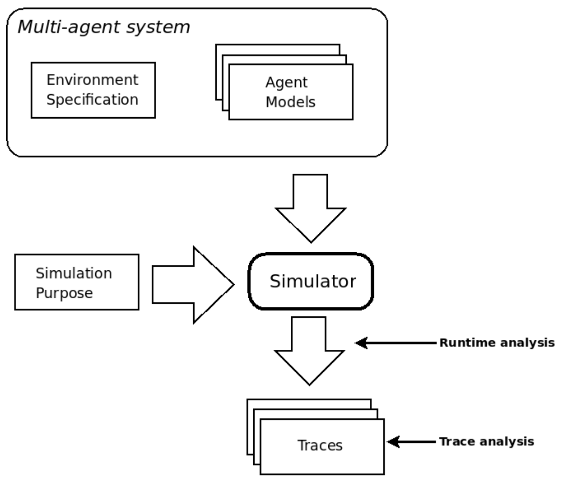

Figure 1: General architecture of our proposed tool. The simulator takes two inputs: (i) a MAS, composed by agent models and an environment specification; (ii) a *simulation purpose* to be tested. The simulator then produces traces as outputs. Verification can be done at the simulation runtime level, as well as at the trace level. The simulation purpose specifies the property to be analyzed and is used to guide the simulations (i.e., defines their purpose).

explorations of state-spaces, and thus can be subjected to some kinds of formal analyses.

A MAS can be decomposed into two aspects. The first relates to the agents. The second deals with how such agents come together and interact among themselves. The elements that form this second aspect constitute the *environment*[1] of a MAS.

That said, our work is concerned about how one can build a MAS to model a complex situation suitable for both exploratory simulation and approximate formal verification. To achieve this, we aim at providing three basic elements: (i) an agent model, which we have already described in [1]; (ii) a formal specification of the environment of these agents, so that they can be composed into a MAS; and (iii) techniques to formally analyze the resulting MAS. Our ultimate objective is to provide a tool that shall support these three elements and thus allow the specification, simulation and verification of such MASs. We shall build it on top of our existing simulation infrastructure [2]. Figure 1 outlines the architecture for such a tool.

In this report, though, we focus on the problem of defining environments. Our environments have a social network structure in which nodes are agents, and the links between them are defined by the capabilities that agents have to act upon each other. Furthermore, environments are more than a network structure, as they may change dynamically, either spontaneously or as a reaction to an agent's actions. These design choices arise from the agent model that

[1]Notice that the term "environment" is not used consistently in the MAS literature [14]. Sometimes, it is used to mean the conceptual entity in which the agents and other objects exist and that allows them to interact; sometimes, it is used to mean the computational infrastructure that supports the MAS (e.g., a simulator). We use the term in the former sense.

we consider [1]. In it, agents are described from the point of view of Behavioral Psychology [11], which suggests a number of desirable features from an environment that brings them together. For instance, great importance is placed on the possibility of performing experiments of different kinds, and of responding to agent's actions in appropriate ways. As we shall see, our approach achieves this by the *environment behaviors* it defines. Furthermore, interaction is mostly interestingly treated by abstracting physical properties away and dealing only with relationships, which we do by adopting a social network structure and operations to modify it. We believe that these characteristics already differentiate our work substantially from other existing environment description methods (see Weyns *et al.* [14] for a survey).

Here we develop a simple formal framework in which to define such environments so that they can be subject to automated analyses procedures. A high-level specification language is provided, which we call the *Environment Specification Language* (ESL), and its semantics is given in terms of the π-calculus process algebra [6, 9].

Process algebras are typically employed to describe concurrent systems. They are good at succinctly describing behaviors relevant for inter-process communication. Our particular choice of π-calculus as a theoretical foundation is motivated by a number of its distinguishing features among existing such algebras. First, it takes communication through channels as a primitive notion, which makes it a natural choice for representing networks. Second, it allows for dynamic modification, which makes the creation and destruction of connections between agents possible. Third, it provides a convenient representation for broadcast behavior through its replication operator. Finally, it has few operators and a simple operational semantics, which is attractive for implementation.

It is worth to note that despite all of these qualities of process algebras in general, and of π-calculus in particular, they are not usually employed in the context of multi-agent systems simulation. One exception is the work of Wang and Wysk [12], which uses a modified π-calculus to express a certain class of agents and their environments. But their approach is not sufficient to deal with our problems, and thus we develop our own method.

We purposefully treat agents as black-boxes here. This does not mean that they have no known internal structure; it merely means that such structure is mostly irrelevant as far as their environment is concerned. We assume, thus, that those two aspects of a MAS are complementary, but separate, issues. However, there must be a way to interface the agents with their environment. This is achieved through the assumption that agents receive *stimuli* as input and that they output *actions*.

The text is organized as follows. Section 2 provides the preliminary definitions in order to give an appropriate semantics to ESL. Then, Section 3 presents the language itself. The language's semantics, in turn, is given in Section 4. For the sake of clarity, Section 5 provides a concrete example of environment built in ESL. At last, Section 6 summarizes the main points presented and considers the new perspectives that ESL brings. A brief overview of the π-calculus is also provided in Appendix A.

For the sake of readability, we have omitted π-calculus input and output parameters when such parameters are not relevant (e.g., we write $\overline{a}$ instead of $\overline{a}(x)$ if x is not used later). Moreover, some elements are colored in a different manner (i.e., preliminary definitions; expressions, sets and logical formulas of

ESL; and the $[\ ]_\pi$ translation function that we shall shortly introduce).

2 Preliminary Definitions

The environment formalization presented in this report must have a way to effectively interact with the agents of a MAS. Agents may trigger events that have a meaning in the environment specification (e.g., the performance of an action). Conversely, the environment specification may request the performance of an operation (e.g., to stimulate an agent). It is necessary, therefore, to have a mathematical foundation that formally defines how to accomplish this. We fulfill this requirement by providing both a *vocabulary* in which a few primitives are defined and a definition for what constitutes an *environment status* with respect to these primitives.

Both of these definitions emanate from a particular agent model [1] that we are considering. This model establishes how agents are stimulated and how they produce behavioral responses. And since it is precisely through stimulation and behavior that agents interface with their environment, the assumptions about these aspects must be made explicit here. In other words, the definitions below can be seen as interfaces that allow an environment to communicate with its agents.

Definition 1 (Vocabulary). *A* vocabulary *is a tuple*

$$\langle Stimuli, Actions, AgentIDs \rangle$$

such that:

- $Stimuli$ *is a finite set of* stimuli*;*
- $Actions$ *is a finite set of* actions*;*
- $AgentIDs$ *is a finite set of* agent identifiers*;*

The sets $Stimuli$, $Actions$ and $AgentIDs$ define, respectively, all available stimuli, actions and agent identifiers. These are sets containing primitive, unstructured, elements.

Definition 2 (Environment Status). *An* environment status *is a pair*

$$\langle Stimulation, Response \rangle$$

such that:

- $Stimulation : AgentIDs \times Stimuli \to \{Beginning, Stable, Ending, Absent\}$*;*
- $Response : AgentIDs \times Actions \to \{Emitting, NotEmitting\}$*.*

The $Stimulation$ function controls the stimulation of a particular agent by a particular stimulus. Notice that agent stimulation is not an instantaneous operation. We assume the agent differentiates the beginning, the stable phase, the ending, and the absence of a particular stimulation. Hence, we provide the appropriate elements in the function's range.

The *Response* function keeps track of the actions being emitted by the agents. We assume that actions begin and end instantaneously, and therefore we define only two elements in the function's range. Notice that this difference in relation to the *Stimulation* function arises from the underlying agent model [1], which expects things to work like this.

Moreover, we denote by ES the set of all such environment status.

Later on we will show how to relate an environment's π-calculus process with such an environment status. By this provision, we will be able interpret the π-calculus LTS in terms of MAS primitives. Roughly, this is how the semantics of our environment formalization shall be given in Section 4.

3 Environment Specification Language

The environment must be described in a formal way, and to this end we have chosen to give its semantics using the π-calculus. However, in order to hide the complexities of the underlying process algebra, it is convenient to provide a higher-level language to write environment descriptions. We do this by defining the *Environment Specification Language* (ESL). Expressions in this language can then be automatically translated into π-calculus. This is achieved by using a translation function to map constructs of this language into π-calculus (i.e., a construct C is translated to $[C]_\pi$).

Definition 3 (Translation function)**.** *The* translation function $[\ ]_\pi$ *maps constructs of ESL into π-calculus expressions.*

The full definition of this function is given as new constructs are introduced.

The constructs of ESL can be roughly divided into *structures* and *operations*. The former define the elements that exist and how they interact. The later account for the manipulation of these structures.

The text below is organized as follows. Section 3.1 defines the fundamental π-calculus events upon which our formalization is built. Section 3.2 describes the main structures that form an environment. Then, Section 3.3 introduces some core operations and functions to manipulate these structures and which provide the basis for the environment's dynamic aspect. Finally, Section 3.4 provides convenience operations built on top of these core ones.

3.1 Underlying Elementary π-Calculus Events

A π-calculus specification can be divided into two parts. First, and most fundamentally, it is necessary to specify the set of events that are particular to that specification. Second, it is necessary to specify processes built using those events. In this section we account for this first part.

Input and output events are all made from basic names. Hence, we first formally define a set of names in order to have the corresponding events. The definition below define such names, and Table 1 presents an informal description of the events that arise. The formal description of their meaning, however, shall be given later on, in Section 4, by characterizing environment status based on the events that can be performed.

Definition 4 (Environment Names)**.** *The* environment names *are defined by the following set:*

$$\begin{aligned} ENames = \quad & \{\overline{emit}_a^n, \overline{stop}_a^n, beginning_s^n, stable_s^n, absent_s^n, \\ & \overline{destroy}_{a,n}^{s,m}, \overline{ccn}, \overline{done} \mid \\ & a \in Actions, s \in Stimuli, m, n \in AgentIDs\} \end{aligned}$$

Notice that names are primitive entities, even though they are denoted here with subscripts and superscripts, which could suggest some sort of parametrization. This writing style is merely for readability's sake.

With these names, we may now establish the set of events relevant to ESL.

Definition 5 (Environment Events)**.** *The* environment events *are defined by the following set:*

$$EEvents = \quad \{\overline{e}(x), e\langle x\rangle \mid e, x \in ENames\} \cup \{\tau\}$$

As a technicality, it is sometimes convenient to be able to translate π-calculus processes and events using the $[\ ]_\pi$ function. The result of such translation is, of course, the process or event itself. Thus we extend the domain of $[\ ]_\pi$ to include π-calculus and give the following definition.

Definition 6. *Let P be an arbitrary π-calculus process or prefix. Then,*

$$[P]_\pi = P$$

A corollary of this definition is that the $[\ ]_\pi$ function is idempotent (i.e., $[[C]_\pi]_\pi = [C]_\pi$).

Event	**Informal description**
	Agent to environment
$\overline{emit}_a^n$	Agent identified by n performs action a.
$\overline{stop}_a^n$	Agent identified by n stops performing action a.
	Environment to agent
$beginning_s^n$	Delivery of stimulus s to the agent identified by n is beginning.
$stable_s^n$	Delivery of stimulus s to the agent identified by n is stable.
$ending_s^n$	Delivery of stimulus s to the agent identified by n is ending.
$absent_s^n$	Delivery of stimulus s to the agent identified by n becomes absent.
	Environment to environment
$\overline{destroy}_{a,n}^{s,m}$	Requests the destruction of an action transformer that converts action a from agent identified by n into stimulus s accepted by the agent identified by m.
$\overline{ccn}$	Requests the creation of a new action transformer.
$\overline{done}$	Signals that an operation has terminated.

Table 1: Informal description of events, divided in three categories according to their origin and destination. The corresponding output or input events not shown merely allow the ones described to work properly.

3.2 Main Environment Structures

The *environment* is the central structure of specifications. It defines which agents are present, how they are initially connected, and what dynamic behaviors exist in the environment itself.

Definition 7 (Environment)**.** *An* environment *is a tuple* $\langle AG, AT, EB\rangle$ *such that:*

- $AG = \{ag_1 \dots ag_l\}$ *is a set of* agent profiles*;*
- $AT = \{t_1 \dots t_m\}$ *is a set of* action transformers*;*
- $EB = \{eb_1 \dots eb_n\}$ *is a set of* environment behaviors*.*

Moreover, let $ENames = \{en_1, \dots, en_o\}$*. Then the corresponding* π*-calculus expression for the environment is defined as:*

$$\begin{aligned} [\langle AG, AT, EB\rangle]_\pi = \;& (\nu\; en_1, \dots, en_o) \\ & ([ag_1]_\pi | [ag_2]_\pi | \dots | [ag_l]_\pi | \\ & [t_1]_\pi | [t_2]_\pi | \dots | [t_m]_\pi | \\ & [eb_1]_\pi | [eb_2]_\pi | \dots | [eb_n]_\pi | \\ & !NewAT) \end{aligned}$$

where

$$\begin{aligned} NewAT = \;& ccn\langle emit, stop, absent, beginning, stable, ending, destroy\rangle. \\ & T(emit, stop, absent, beginning, stable, ending, destroy) \end{aligned}$$

and T *is given in Definition 9.*

This definition merits a few comments. First, all names from $ENames$ are restricted to the environment. Second, the set of action transformers provide the network structure that connects the agents. Third, the environment behaviors, as the name implies, specifies behaviors that belong to the environment itself. This is useful to model reactions to agent's actions, as well as to capture ways in which the environment may evolve. In the first case the behavior is specified as an *environment response* (Definition 10 below), while in the second case the behavior is simply an ESL operation. Finally, the component $NewAT$ allows the creation of new action transformers. In order to do so, it receives a message $\overline{ccn}$ ("create connection"), whose parameters initialize the rest of the expression. To see this more clearly, suppose that $NewAT$ is in parallel composition as follows:

$$\overline{ccn}(emit^n_a, stop^n_a, absent^m_s, beginning^m_s, stable^m_s, ending^m_s, destroy^{s,m}_{a,n})|NewAT$$

Then $\overline{ccn}$ will react with ccn in $NewAT$, and the resulting expression will be the following:

$$T(emit^n_a, stop^n_a, absent^m_s, beginning^m_s, stable^m_s, ending^m_s, destroy^{s,m}_{a,n})$$

This expression corresponds to the definition of an action transformer, as we shall shortly see (Definition 9). Furthermore, notice that in the environment definition there is a parallel replication operator on $!NewAT$. This ensures that the creation of action transformers can happen as many times as needed to produce reactions[2], owing to the following structural congruence rule:

$$!NewAT \equiv NewAT | !NewAT$$

Environments exist in order to allow agents to interact. As we remarked earlier, the internal structure of these agents, as complex as it may be, is mostly irrelevant for their interaction model. Thus, we have abstracted it away as much as possible. What is left are the interfaces that allow agents to interact with each other and with the environment itself, which we call *agent profiles*. Hence, we have the following definition.

Definition 8 (Agent Profile)**.** *An* agent profile *is a triple $\langle n, S, A \rangle$ such that:*

- *$n \in AgentIDs$ is a unique identifier for the agent;*
- *$A = \{a_1 \ldots a_i\} \subseteq Actions$ is a set of actions;*
- *$S = \{s_1 \ldots s_j\} \subseteq Stimuli$ is a set of stimuli.*

Moreover,

$$\begin{aligned} [\langle n, S, A \rangle]_\pi = \quad & ([Act(a_1, n)]_\pi | [Act(a_2, n)]_\pi | \ldots | [Act(a_i, n)]_\pi) | \\ & ([Stim(s_1, n)]_\pi | [Stim(s_2, n)]_\pi | \ldots | [Stim(s_j, n)]_\pi) \end{aligned}$$

such that, for all $a \in A$ and $s \in S$, we have:

$$[Act(a, n)]_\pi = !(\overline{emit^n_a}.\overline{stop^n_a})$$

$$[Stim(s, n)]_\pi = \quad piStim(absent^n_s, beginning^n_s, stable^n_s, ending^n_s)$$

where

$$\begin{aligned} & piStim(absent, beginning, stable, ending) = \\ & \quad absent.beginning.stable.ending.piStim(absent, beginning, stable, ending) \end{aligned}$$

In this definition, it is clear that agents have several components, each responsible for controlling one particular action or stimulus. $Act(a, n)$ defines that the agent identified by n can start emitting an action a and can then stop such emission. The replication operator ensures that this sequence can be carried out an unbounded number of times. $Stim(s, n)$, in turn, defines that the agent identified by n can be stimulated by s, and that this stimulation follows four

[2]For the reader familiar with the π-calculus, it is worth to note that because all environment names are restricted, the only way for the system to progress is by performing reactions by the application of the COM rule found on the operational semantics (see Definition 34). Moreover, the rule $STRUCT$ together with the structural congruence relation ensures that COM will be applied as long as there are $\overline{ccn}$ events to react with $NewAT$.

steps. The recursive call ensures that this stimulation sequence can start again as soon as it finishes the last step. Notice that these definitions reflect the assumptions about the agent model we consider [1].

Agents interact by stimulating each other. But to have this capability, it is first necessary to define that an agent's action causes a stimulation in another agent. This is done through *action transformers*.

Definition 9 (Action Transformer)**.** *An* action transformer *is a tuple* $\langle ag_1, a, s, ag_2 \rangle$ *such that:*

- ag_1 *is an agent profile* $\langle n, S_1, A_1 \rangle$*;*
- ag_2 *is an agent profile* $\langle m, S_2, A_2 \rangle$*;*
- a *is an action such that* $a \in A_1$*;*
- s *is a stimulus such that* $s \in S_2$*;*

Moreover, the corresponding π*-calculus expression for the action transformer is defined as:*

$$[\langle ag_1, a, s, ag_2 \rangle]_\pi = \\ T(emit_a^n, stop_a^n, absent_s^m, beginning_s^m, stable_s^m, ending_s^m, destroy_{a,n}^{s,m})$$

where

$$T(emit, stop, absent, beginning, stable, ending, destroy) = \\ (\overbrace{emit.\overline{beginning}.\overline{stable}.stop.\overline{ending}.\overline{absent}}^{Normal\ behavior}.T(emit, stop, absent, beginning, stable, ending, destroy)) + \\ \underbrace{destroy}_{To\ disable\ the\ action\ transformer}$$

The above definition can be divided in two parts. First, there is its normal behavior, which merely defines the correct sequence through which an action is transformed in a stimulus. Once such a sequence is completed, a recursive call to the process definition restarts the action transformer. Second, there is the part that allows the transformer to be destroyed. By performing *destroy*, the action transformer disapears, since this event is not followed by anything.

We choose to have an intermediate structure such as the action transformer between the agents instead of allowing a direct communication because an agent's actions may have other effects besides stimulation. In particular, the environment can also respond to such actions in custom ways. This is done by expressions of *environment response laws*, which the user is supposed to define, and are part of the environment behavior.

Definition 10 (Environment Response)**.** *Let* $\langle n, S, A \rangle$ *be an agent profile, and* $a \in A$ *an action. Then the* environment response *function* $ER()$ *for this action and agent is defined as follows:*

$$ER(a, ag) = Op(a, ag)$$

where $Op(a, ag)$ *is some ESL operation in which* a *and* ag *are free variables.*

Moreover, the corresponding π-calculus expression is as follows:

$$[ER(a, ag)]_\pi = emit_a^n.[Op(a, ag); ER(a, ag)]_\pi$$

In this definition, $Op(a, ag)$ must be given by the user. As an example of such an environment response, we may cite the classical notion of reinforcement from behavioral psychology. When an agent performs a desirable action, the environment may be designed so that the agent receives a reward in order to reinforce this behavior. This relation between the agent's action and an associate reward can be elegantly modeled in a process algebraic way according to the above definition of environment response.

3.3 Core Environment Operations and Functions

ESL provides a core of definitions upon which others can be built. In particular, a number of *operations* are defined. The meaning of such operations is given as π-calculus expressions with the particularity of signaling their own termination by the $\overline{done}$ event. This detail is important because it will allow their sequential composition, as we shall see in Section 3.3.6.

Below we present these core definitions according to their purpose.

3.3.1 Agent Stimulation

The following operations are provided to control the stimulation of agents.

Definition 11 (Begin stimulation operation). *Let $ag = \langle n, S, A \rangle$ be an agent profile, and $s \in S$ be a stimulus. Then the* begin stimulation *operation is writen as:*

$$BeginStimulation(s, ag)$$

Moreover,

$$[BeginStimulation(s, ag)]_\pi = \overline{beginning_s^n}.\overline{stable_s^n}.\overline{done}$$

Definition 12 (End stimulation operation). *Let $ag = \langle n, S, A \rangle$ be an agent profile, and $s \in S$ be a stimulus. Then the* end stimulation *operation is writen as:*

$$EndStimulation(s, ag)$$

Moreover,

$$[EndStimulation(s, ag)]_\pi = \overline{ending_s^n}.\overline{absent_s^n}.\overline{done}$$

Definition 13 (Stimulate operation). *Let $ag = \langle n, S, A \rangle$ be an agent profile, and $s \in S$ be a stimulus. Then the* stimulate *operation is defined as:*

$$Stimulate(s, ag) = BeginStimulation(s, ag); EndStimulation(s, ag)$$

3.3.2 Action Transformers

The following operations are provided to manipulate action transformers.

Then we have the following operations.

Definition 14 (Create action transformer operation)**.** *Let* $ag_1 = \langle n, S_1, A_1 \rangle$ *be an agent profile,* $ag_2 = \langle m, S_2, A_2 \rangle$ *be another agent profile,* $a \in A_1$ *be an action, and* $s \in S_2$ *be a stimulus. Then the* create action transformer *operation is writen as:*

$$Create(ag_1, a, s, ag_2)$$

Moreover,

$$[Create(ag_1, a, s, ag_2)]_\pi = \overline{ccn}(emit^n_a, stop^n_a, absent^m_s, beginning^m_s, stable^m_s, ending^m_s, destroy^{s,m}_{a,n}).\overline{done}$$

In the above definition, notice that $\overline{ccn}$ is crafted to react with the component $NewAT$ given in Definition 7. Since operations will ultimately be put together with parallel composition in the environment, it follows that the $Create(ag_1, a, s, ag_2)$ operation will be able to react with $NewAT$ and originate a new action transformer.

Definition 15 (Destroy action transformer operation)**.** *Let* $ag_1 = \langle n, S_1, A_1 \rangle$ *be an agent profile,* $ag_2 = \langle m, S_2, A_2 \rangle$ *be another agent profile,* $a \in A_1$ *be an action, and* $s \in S_2$ *be a stimulus. Then the* destroy action transformer *operation is writen as:*

$$Destroy(n, a, s, m)$$

Moreover,

$$[Destroy(n, a, s, m)]_\pi = \overline{destroy^{s,m}_{a,n}}.\overline{done}$$

3.3.3 Sets

Certain sets of elements are particularly useful for modeling. The core language provides functions that allow one to access them.

Definition 16. *Let* X *be any set,* $S \subseteq Stimuli$*,* $A \subseteq Actions$*,* $ag = \langle n, S, A \rangle$ *be an agent profile,* i, j *be natural numbers and* $I \subseteq AgentIDs$*. Then we have the following special sets:*

- $\emptyset$*: The empty set.*
- $\mathbb{P}(X)$*: The set of all subsets of* X *(i.e., its power set).*
- $canReceive(n) = S$
- $canEmit(n) = A$
- $i..j = \{k \mid i \leq k \leq j\}$.
- $\langle I, S, A \rangle = \{\langle id, S, A \rangle \mid id \in I\}$

The $\langle I, S, A \rangle$ construction allows the concise specification of large sets of similar agents. It is especially useful if the agent identifiers are natural numbers, because in this case it can be used in association with the $i..j$ construction. For example, if we know that agent identified by 1 up to 100 are all similar, we can specify all of their profiles at once by writing $\langle 1..100, S, A \rangle$.

Composite sets can be obtained by the usual operators of $\cup$ (union), $\cap$ (intersection) and $\setminus$ (subtraction).

3.3.4 Predicates and Logical Formulas

Primitive predicates are necessary to specify conditions. Below we define relevant predicates for ESL.

Definition 17. *Let X be a set, ag_1 and ag_2 be agent profiles, a be an action, s be a stimulus, and x and y be agents, stimuli or actions. Then we have the following predicates:*

- $isConnected(ag_1, a, s, ag_2)$*: True if, and only if, there exists an action transformer that takes action a from agent ag_1 and transforms it in stimulus s delivered to agent ag_2.*
- $x \in X$*: True if, and only if, x is in set X.*
- $x \notin X$*: True if, and only if, x is not in the set X.*
- $x = y$*: True if, and only if, x and y refer to the same entity.*
- $x \neq y$*: True if, and only if, x and y refer to different entities.*

Formulas can be obtained by using the usual logical connectives $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction) and $\rightarrow$ (implication).

3.3.5 Quantification

In order to succinctly express arbitrary number either of choices or of concurrent execution, it is convenient to define two special quantification operators. Given a set of possible parameters and a parameterized expression, these operators generate a new expression that corresponds to a composition of the several instantiations that the given expression might have with respect to the specified set of possible parameters.

Definition 18 (Universal quantification with sum)**.** *Let Y be a finite set, $Exp()$ be an arbitrary expression, and $Formula$ be a logic formula that is obeyed by the elements $y_1, y_2, \ldots y_n \in Y$. Then the* universal quantification with sum *is defined as:*

$$\forall_{+} y : Y | Formula \bullet Exp(y) = Exp(y_1) + Exp(y_2) + \ldots + Exp(y_n)$$

Definition 19 (Universal quantification with parallel composition)**.** *Let Y be a finite set, $Exp()$ be an arbitrary expression, and $Formula$ be a logic formula that is obeyed by the elements $y_1, y_2, \ldots y_n \in Y$. Then the* universal quantification with parallel composition *is defined as:*

$$\forall_{|} y : Y | Formula \bullet Exp(y) = Exp(y_1) | Exp(y_2) | \ldots | Exp(y_n)$$

3.3.6 Composition Operators

In order to build complex operations on top of the basic ones, ESL provides composition operators. Some of these can be mapped directly to π-calculus operators, but others require more sophistication.

Definition 20 (Sequential Composition)**.** *Let Op_1 and Op_2 be operations. Then their* sequential composition *is also an operation and is written as:*

$$Op_1; Op_2$$

Moreover,

$$[Op_1; Op_2]_\pi = (\nu\ start)[Op_1]_\pi\{start/done\}|start.[Op_2]_\pi$$

The above translation aims at accounting for the intuition that Op_1 must take place before Op_2. However, we cannot translate $Op_1; Op_2$ immediatly as $[Op_1]_\pi.[Op_2]_\pi$, because in general π-calculus would not allow the resulting syntax (e.g., $(P + Q).R$ would not be a valid expression). Therefore, we adapt the suggestion offered by Milner [6] (in Example 5.27), which works as follows. We assume that every operation signals its own termination using the $\overline{done}$ event. Then, when composing Op_1 and Op_2, we: (i) create a new event, *start*; (ii) rename the *done* event in Op_1 to *start*; (iii) make *start* guard Op_2; (iv) put the two resulting processes in parallel. Notice that, by this construction, the only way that Op_2 can be performed is after $\overline{start}$ is performed, which can only happen when Op_1 terminates.

Definition 21 (Sequence)**.** *Let Op be an operation and n be an integer such that $n \geq 1$. Then a* sequence *of n compositions of Op is defined as:*

$$Seq(Op, n) = \begin{cases} Op; Seq(Op, n-1) & n > 1 \\ Op & n = 1 \end{cases}$$

Definition 22 (Unbounded Sequence)**.** *Let Op an operation. Then an* unbounded sequence *of compositions of Op is defined as:*

$$Forever(Op) = Op; Forever(Op)$$

The translation of these two kinds of sequences to π-calculus follows, of course, from the translation of the sequential composition operator.

Definition 23 (Choice)**.** *Let Op_1 and Op_2 be operations. Then their composition as a* choice *is also an operation and is written as:*

$$Op_1 + Op_2$$

Moreover,

$$[Op_1 + Op_2]_\pi = [Op_1]_\pi + [Op_2]_\pi$$

Definition 24 (Parallel Composition)**.** *Let Op_1, Op_2, ..., Op_n be n operations. Then their* parallel composition *is also an operation and is written as:*

$$Op_1|Op_2|\ldots|Op_n$$

Moreover,

$$\begin{aligned}[Op_1|Op_2|\dots|Op_n]_\pi = \quad & (\nu\ start)[Op_1]_\pi\{start/done\}|[Op_2]_\pi\{start/done\}|\dots| \\ & [Op_n]_\pi\{start/done\}|\underbrace{start.start.\dots.start}_{n\ times}.\overline{done}\end{aligned}$$

The translation for the parallel composition is not straightforward because it is necessary to ensure that $\overline{done}$ is sent only once in the composed operation. That is to say, the parallel composition of n operations[3] is an operation itself, and it only terminates when each of its components terminates. If this care is not taken, later sequential compositions will not work as expected. Our definition ensures the correct translation by: (i) creating a new name, *start*, restricted to the composition; (ii) renaming *done* to *start* in Op_1, Op_2, ..., Op_n ; (iii) creating a new component that waits for n *start* events before sending one $\overline{done}$. By this construction, the only way that a $\overline{done}$ event can be sent is by first producing n *start* events, which can only happen if each operation terminates individually.

3.4 Convenience Environment Operations

Using the operations defined above, we may define a number of other convenience operations. There are many possibilities for such operations, but below we give some examples that seem useful. We employ polymorphism where appropriate to avoid creating new names and to show possible variations of an operation.

Let $S \subseteq Stimuli$ be a set of stimuli, $s \in Stimuli$ be a stimulus, $A \subseteq Actions$ be a set of actions, and AG, AG_1 and AG_2 be sets of agent profiles. Then we have the following operations.

Stimulate several agents. A stimulus is delivered to the agents.

$$Stimulate(s, AG) = \forall_| ag : AG | s \in canReceive(ag) \bullet Stimulate(s, ag)$$

Stimulate several agents with several stimuli. Several stimuli are delivered to the agents.

$$Stimulate(S, AG) = \forall_| s : S \bullet Stimulate(s, A)$$

Connect two sets of agents. Allows the creation of action transformers between two specified sets of agents using the specified sets of actions and stimuli. Notice that this does not mandate that the action transformers should actually be created. Rather, it specifies that it is possible for them to be created. This allows one to consider all the possibilities of connections between the two sets.

$$\begin{aligned}Connect(AG_1, AG_2, A, S) = \quad & \forall_| ag_1 : AG_1 \bullet \forall_| ag_2 : AG_2 \bullet \forall_| a : A \bullet \forall_| s : S| \\ & AG_1 \cap AG_2 = \emptyset \wedge a \in canEmit(ag_1) \wedge \\ & s \in canReceive(ag_2) \bullet \\ & Create(ag_1, a, s, ag_2)\end{aligned}$$

[3]We define the operator for n operations instead of just two because this avoids the problem of establishing its associativity properties.

Connect agents in set. Similarly, allows the creation of action transformers between the agents of a specified set using the specified sets of actions and stimuli.

$$\begin{aligned} Connect(AG, A, S) = \; & \forall_{|} ag_1 : AG \bullet \forall_{|} ag_2 : AG \bullet \forall_{|} a : A \bullet \forall_{|} s : S| \\ & ag_1 \neq ag_2 \wedge a \in canEmit(ag_1) \wedge s \in canReceive(ag_2) \bullet \\ & Create(ag_1, a, s, ag_2) \end{aligned}$$

Disconnect agent in a set. Destroys the action transformers between the agents in the specified set.

$$\begin{aligned} Disconnect(AG) = \; & \forall_{|} ag_1 : AG \bullet \forall_{|} ag_2 : AG \bullet \forall_{|} a : canEmit(ag_1) \bullet \\ & \forall_{|} s : canReceive(ag_2)| \\ & ag_1 \neq ag_2 \wedge isConnected(ag_1, a, s, ag_2) \bullet \\ & Destroy(ag_1, a, s, ag_2) \end{aligned}$$

4 Language Semantics

The semantics of ESL is given by considering: (i) a syntactical translation of ESL into π-calculus expressions; (ii) a mathematical foundation which relates π-calculus events to the stimuli and actions of agents. The π-calculus translation of (i), through its operational semantics (Definition 34), provides an over-approximation of the desired behavior, which is then made precise using the restrictions provided by (ii). By this method, we shall be able to build an LTS that defines the possible states and transitions for any particular environment specification.

More precisely, given an environment E, we shall build an *annotated environment LTS* by considering the LTS induced by $[E]_\pi$, whose states shall be annotated with our environment status (Definition 2), and whose structure shall be subject to some restrictions based on the possible values for an environment status. Then we shall then have an LTS whose states have the following form.

Definition 25 (State)**.** *Let E be an environment and P be a π-calculus process obtained by applying π-calculus operational semantics rules to $[E]_\pi$. Moreover, let $\langle Stimulation, Response\rangle$ be an environment status. Then a* state *is defined as the following pair:*

$$(P, \langle Stimulation, Response\rangle)$$

By this construction, at any point of the LTS we shall be able to know both what is the current situation of the agents (because of the added environment status) and what are the possible changes from that point (because of the π-calculus operational semantics).

To proceed with this construction, we need a number of definitions. Let us begin by providing a way to observe the internal transitions of an environment, which is a fundamental capability that we need before proceeding. Recall from Definition 7 that an environment's π-calculus process has a number of restrictions that would prevent such observations (i.e., the transitions would be internal to the process and not discernible in the LTS). It is, however, possible to characterize these restrictions syntactically, and thus we may provide a simple method to remove them when needed. This is accomplished by the following *environment unrestriction function unr*.

Definition 26 (Environment Unrestriction Function)**.** *Let P and Q be π-calculus processes such that*

$$P = (\nu\ en_1, \ldots, en_o)Q$$

where $\{en_1, \ldots, en_o\} = ENames$*. Then the* environment unrestriction function *is defined as* $unr(P) = Q$*.*

We may now define the *Stimulation* function present in each state as follows.

Definition 27 (Stimulation)**.** *Let* $(P, \langle Stimulation, Response\rangle)$ *be a state. Moreover, let* $\rightarrow$ *be the transition relation induced by the π-calculus operational semantics. Then, for all* $s \in Stimuli$ *and* $n \in AgentIDs$*, we have:*

$$Stimulation(n, s) = \begin{cases} Absent & \textit{if there exists a } P' \textit{such that } unr(P) \overset{beginning_s^n}{\rightarrow} P' \\ Beginning & \textit{if there exists a } P' \textit{such that } unr(P) \overset{stable_s^n}{\rightarrow} P' \\ Stable & \textit{if there exists a } P' \textit{such that } unr(P) \overset{ending_s^n}{\rightarrow} P' \\ Ending & \textit{if there exists a } P' \textit{such that } unr(P) \overset{absent_s^n}{\rightarrow} P' \end{cases}$$

The *Stimulation* definition establishes the status of a particular stimulation based on the order that stimulations must change (see Definition 8). For instance, if a process is capable of receiving a $beginning_s^n$ event, it must be the case that stimulus s is currently absent in agent identified by n. The *Stimulation* function, therefore, merely gives a way of reading the π-calculus LTS in order to have this information explicitly for every agent and stimulus in any given process.

The *Response* function, on the other hand, is assumed as given. Thus, we do not define it. However, it imposes some constraints on the LTS, which we must specify and take in account. As we shall see shortly, these constraints turn the π-calculus over-approximation into an exact description of the transition system's structure that we wish to assign to ESL.

Definition 28 (Transition constraints)**.** *Let* $s_1 = (P_1, \langle Stimulation_1, Response_1\rangle)$ *and* $s_2 = (P_2, \langle Stimulation_2, Response_2\rangle)$ *be states in an annotated environment LTS* $\langle S, L, \rightsquigarrow\rangle$*. Moreover, let* $\rightarrow$ *be the transition relation induced by the π-calculus operational semantics. Then the transition* $s_1 \overset{l}{\rightsquigarrow} s_2$ *is forbidden if one of the cases hold:*

- *there exists* $a \in Actions$ *and* $n \in AgentIDs$ *such that:*
  - $Response_1(n, a) = Emitting$*;*
  - P_2 *was obtained by internally producing the event* $\overline{stop_a^n}$ *in* P_1*.*
- *there exists* $a \in Actions$ *and* $n \in AgentIDs$ *such that:*
  - $Response_1(n, a) = NotEmitting$*;*

- *P_2 was obtained by internally producing the event $\overline{emit_a^n}$ in P_1.*

- *there exists $a \in Actions$ and $n \in AgentIDs$ such that:*
  - *$Response_1(n, a) = Emitting$;*
  - *$Response_2(n, a) = NotEmitting$;*
  - *there exists a P' such that $unr(P_1) \xrightarrow{emit_a^n} P'$.*

- *there exists $a \in Actions$ and $n \in AgentIDs$ such that:*
  - *$Response_1(n, a) = NotEmitting$;*
  - *$Response_2(n, a) = Emitting$;*
  - *there exists a P' such that $unr(P_1) \xrightarrow{stop_a^n} P'$.*

The first constraint asserts that if an agent identified by n is emitting an action a, then it cannot produce the $\overline{stop_a^n}$ event to proceed to a new state. Conversely, the second constraint states that if the agent is not emitting such an action, then it cannot produce the $\overline{emit_a^n}$ event. The third constraint asserts that if the agent is emitting the action in a given state, and it proceeds to a state in which it is no longer emitting such an action, then it must not be the case that some process was still ready to receive that action (i.e., by producing the input event $emit_a^n$). This means that it can only stop emitting an action when the action has already produced all of its effects. The final constraint is the counterpart for stopping an emission. Hence, if an agent is not emitting some action, and then it starts emitting it, it must not be the case that some process was still ready to receive the stop signal (i.e., by producing the input event $stop_a^n$).

At last, we may define the annotated environment LTS as follows.

Definition 29 (Annotated Environment LTS)**.** *Let E be an environment (Definition 7), and let $\rightarrow$ be the transition relation induced by the π-calculus operational semantics (Definition 34). Then an* annotated environment LTS *is an LTS $\langle S, L, \rightsquigarrow \rangle$ such that:*

- *$L = EEvents$ (see Definitions 5);*
- *S and $\rightsquigarrow$ are constructed inductively as follows:*
  - ***Initial state**. $([E]_\pi, es) \in S$, where $es = \langle Stimulation, Response \rangle$ such that for all $a \in Actions$, $s \in Stimuli$, and $n \in AgentIDs$ we have $Stimulation(n, s) = Absent$ and $Response(n, a) = NotEmitting$.*
  - ***Other states and transitions**.*
    If $s_1 = (P_1, \langle Stimulation_1, Response_1 \rangle) \in S$,
    then $s_2 = (P_2, \langle Stimulation_2, Response_2 \rangle) \in S$ and $s_1 \overset{l}{\rightsquigarrow} s_2$ if and only if:
    * *$P_1 \xrightarrow{l} P_2$;*
    * *$Stimulation_2$ is defined w.r.t. P_2 according to Definition 27;*
    * *$s_1 \overset{l}{\rightsquigarrow} s_2$ is not forbidden by Definition 28.*

This definition can be summarized as follows. The LTS has an initial state, which is made of the π-calculus process of some environment, as well as an environment status that says that all actions are not being emitted, and that all stimuli are absent in every agent. From this initial state we begin the construction of the remaining states and of the transition relation. This is accomplished by using the π-calculus operational semantics to know the available transitions at any given state, and then by applying the definitions and constraints we gave previously to prune the possible transitions and augment the reachable states with environment status. This procedure is repeated to every new state introduced until there are no new transitions possible.

5 Example

Let us consider a concrete example in order to see how the previous constructs can be effectively used. We shall model an hierarchical business structure in order to simulate it. In the *Business* environment we shall have several workers, who have powers over one another. The purpose of the system is to produce a certain amount of the *Work* action. The problem is that the workers are also humans and behave differently according to their work load. In particular, some will be more prone to stress than others. The simulation aims at discovering if the particular environment being modeled is capable of achieving the desired amount of work. Below we construct the model step by step.

Actions definition. We begin by defining the possible actions. Employees may work and rest, whereas the president just works.

$$EmployeeActions = \{Work, Rest\}$$
$$PresidentActions = \{Work\}$$

Stimuli definitions. Similarly, we may define the existing stimuli.

$$EmployeeStimuli = \{Money, Stress, Order, Entertainment\}$$
$$PresidentStimuli = \{Stress, Order\}$$

Agent profiles definitions. Agent profiles can then be specified in terms of the previous actions and stimuli. Our example shall have a president, three top directors and 100 other employees. To specify their profiles, we shall use individual agent declarations for the president and the directors, as well as a group declaration for the remaining employees. Furthermore, we assume that the agent identifiers are natural numbers.

$$president = \langle 0, PresidentStimuli, PresidentActions\rangle$$
$$director_1 = \langle 1, EmployeeStimuli, EmployeeActions\rangle$$
$$director_2 = \langle 2, EmployeeStimuli, EmployeeActions\rangle$$
$$director_3 = \langle 3, EmployeeStimuli, EmployeeActions\rangle$$
$$Others = \langle 4..103, EmployeeStimuli, EmployeeActions\rangle$$
$$AG = \{president, director_1, director_2, director_3\} \cup Others$$

Action transformers. Once we have defined agent profiles, we may specify how they relate to each other through action transformers. In our case, we shall begin by just specifying the fact that the business in question has a president and some directors that receives orders from him. That is to say, the work of the president is to give orders, as follows.

$$at_1 = \langle president, Work, Order, director_1 \rangle$$
$$at_2 = \langle president, Work, Order, director_2 \rangle$$
$$at_3 = \langle president, Work, Order, director_3 \rangle$$
$$AT = \{at_1, at_2, at_3\}$$

Further relations are given later on.

Environment response laws. Some characteristics of the model can be better captured through our environment response laws. The employee's work produces money for him, but also some stress. So for all $ag \in AG$ we have:

$$ER(Work, ag) = Stimulate(Money, ag)|Stimulate(Stress, ag)$$

Hence, an employee also needs time off to avoid stress. We may model this by specifying that when an agent $ag \in AG$ rests, he entertains himself:

$$ER(Rest, ag) = Stimulate(Entertainment, ag)$$

The fact that this causes the employee's stress to be reduced is hidden withing the agent. By stating this environment response law, we are actually using this knowlege of the internal behavior of the agent, which is not given in the environment specification.

Initial environment structure. With the previous elements we may define the initial structure of an environment.

$$Business = \langle AG, AT, EB \rangle$$

Enrichment of environment structure. There are many ways in which a business hierarchy can be designed. Thus, the problem of choosing an appropriate structure arises. Our formalism allows for the specification of a number of possibilities at once, in order to capture a whole proposed "architecture". This frees the modeler from having to define each possibility individually and then simulating it.

In our example, we have already defined a relation between the president and the top directors. Let us now specify the rest of the company's hierarchy in a less rigid form. Again, the fundamental idea is that some work is done and then this results in orders for one or more subordinates. So first we define the actions and stimuli that matter for this purpose.

$$A = \{Work\}$$
$$S = \{Order\}$$

Then we specify the possible candidates for each hierarchical layer. If we suppose that employees

$$\text{Best} = \{\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23\ \}$$

are known to be more competent, and that we want three layers below the president, we can get the following ones:

$$L_1 = \{director_1, director_2, director_3\}$$
$$L_2 = \langle Best, EmployeeStimuli, EmployeeActions\rangle$$
$$L_3 = \langle Others \backslash Best, EmployeeStimuli, EmployeeActions\rangle$$

Finally, we state explicitly that these agents sets should be part of a layered structure. To do so, we include the following environment behaviors.

$$Connect(L_1, L_2, A, S) \in EB$$
$$Connect(L_2, L_3, A, S) \in EB$$

This specifies that any connections between adjacent layers are possible. So, for instance, an employee may end up receiving orders from more than one superior.

Notice that while all the employees look similar (i.e., are capable of similar actions and stimulation), it is not at all the case that they really act in the same way. For instance, the stress tolerance of the several agents may be very different, and therefore some of them might be better suitable for certain positions. But these characteristics are not exposed in the environment's specification, which has access only to the agent profiles (i.e., restrictions about which interactions are possible for the agent).

6 Conclusion

In this report we have presented a formalization for environments of MASs. We provided a high-level description for this formalization, with a semantics given in π-calculus. We also provided a vocabulary in which to describe primitives that are necessary for the understanding of some crucial π-calculus events.

The presented environments have both structural and operational aspects. That is to say, they represent certain structures, which can then be changed by certain operations. These operations serve to two purposes. First, they provide a way to specify behaviors of the environments themselves (e.g., environment responses to the actions of agents). Second, they allow the succint specification of several possible scenarios for an environment (e.g., several possible network structures).

With the presented formalization, we will now consider concrete means for analyzing our MASs. To this end, a number of questions can be formulated, namely:

- Since the semantics of ESL is given as an LTS, it follows that now we we need criteria for selecting paths in it. With such paths, we shall be able to perform concrete simulations.

- The available formal elements suggest logical predicates of interest. For instance, a predicate which asserts that an agent is beginning to be stimulated (e.g., for an agent identified by n and a stimulus s, $stimulationBeginning(n, s)$) is a natural choice, since we can obtain this information by observing the annotated environment LTS.

- We may also consider how to implement the proposed language. In this respect, we believe that the π-calculus base can be particularly useful, since we could implement its few elements in order to have our whole language on top of it. A similar approach for simulation is taken by Wang and Wysk [12]. More generally, there are programming languages based on π-calculus, such as the Join-Calculus [3] and Pict [10].

With all of these artifacts, we shall then be able to assemble an actual tool, which is our final objective.

Acknowledgments

The author would like to thank Prof. Dr. Marie-Claude Gaudel (Laboratoire de Recherche en Informatique, Université Paris-Sud 11) for her numerous comments and suggestions during the preparation of this work, as well as Prof. Dr. Ana C. V. de Melo (Institute of Mathematics and Statistics, University of São Paulo) for her remarks on the last versions of the text.

This work benefited from the financial support of *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES) and *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq).

A Brief Overview of the π-Calculus

This section presents a brief account of the π-calculus. Our objective is not to teach the calculus, but merely to quickly recall the notions that we employ to accomplish our work. The definitions we present are adapted from Parrow [9], which the reader might also find useful as an introduction to the calculus.

The π-calculus is a process algebra designed to model interaction and mobility of *processes*[4]. To do so, it provides an algebraic language in which to write such processes, as well as a mathematical framework that interprets them in terms of Labeled Transition Systems (LTS). Let us then begin by defining what an LTS is.

Definition 30 (Labeled Transition System). *A* labeled transition system *is a tuple $\langle S, L, \rightarrow \rangle$ such that:*

- *S is a set of* states*;*
- *L is a set of* labels*;*
- *$\rightarrow \in S \times L \times S$ is a* transition *relation.*

[4]In the literature, π-calculus processes are often called "agents". We avoid using this terminology in order to don't confuse it with our own notion of agents.

Moreover, let $s_1, s_2 \in S$ and $l \in L$. Then, if $\langle s_1, l, s_2 \rangle \in \rightarrow$, we also denote this fact by writing $s_1 \xrightarrow{l} s_2$. The opposite fact, in turn, is denoted by $\neg(s_1 \xrightarrow{l} s_2)$.

Processes are written by using *names* to create prefixes and by using several operators to combine such prefixes. These prefixes represent *events*[5].

Definition 31 (π-calculus Process)**.** *Let a, x, y, $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ be names. Then a* π-calculus process *is an expression defined by the following syntax.*

Prefixes				
α	$::=$	$\overline{a}(x)$	*Output*	
		$a\langle x \rangle$	*Input*	
		τ	*Internal*	
Processes				
P	$::=$	$\mathbf{0}$	*Nil*	
		$\alpha.P$	*Prefix*	
		$P + P$	*Choice*	
		$P \vert P$	*Parallel composition*	
		$(\nu x)P$	*Restriction*	
		$[x = y]P$	*Match*	
		$[x \neq y]P$	*Mismatch*	
		$!P$	*Parallel replication*	
		$A(y_1, \ldots, y_n)$	*Identifier*	
Definitions				
		$A(x_1, \ldots, x_n) = P$	*Process definition*	$(i \neq j \Rightarrow x_i \neq x_j)$

Given a a process P, we denote the set of its *bound names* by $bn(P)$, and of its *free names* by $fn(P)$. Moreover, we denote by $\mathscr{P}^{\pi}$ the set of all processes.

Names often need to change over the course of a process execution. This is achieved using *substitution functions.*

Definition 32 (Substitution Function)**.** *Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ be names, and P be a process. Then a* substitution function

$$\{x_1, \ldots, x_n / y_1, \ldots, y_n\}$$

maps process P into P', such that in P':

- *For all $y_i \in fn(P)$, x_i will substitute y_i in P';*
- *Alpha-conversion is performed in order to prevent that $x_i \in bn(P')$.*

Moreover, we denote the application of the substitution function on P as:

$$P\{x_1, \ldots, x_n / y_1, \ldots, y_n\}$$

It is often the case that processes which are syntactically different should have the same behavior. To model this, the calculus provides a *structural congruence* relation, which defines equivalences that can be determined by syntax alone. This is useful, in particular, to fully define the replication operator.

[5]In the literature, such events are often called "actions". Again, we avoid using this terminology in order to prevent confusion with our own notion of actions.

Definition 33 (π-calculus Structural Congruence)**.** *Let P, Q and R be arbitrary π-calculus processes. Then the* structural congruence *relation $\equiv$ is the smallest relation that satisfies the following axioms.*

- *If P and Q are variants by alpha-conversion, then $P \equiv Q$*
- $P|Q \equiv Q|P$, $(P|Q)|R \equiv P|(Q|R)$, $P|\boldsymbol{0} \equiv P$
- $P + Q \equiv Q + P$, $(P + Q) + R \equiv P + (Q + R)$, $P + \boldsymbol{0} \equiv P$
- $!P \equiv P|!P$
- *Scope extension laws:*
  - $(\nu x)\boldsymbol{0} \equiv \boldsymbol{0}$
  - $(\nu\ x)(P|Q) \equiv P|(\nu\ x)Q$ *if* $x \notin fn(P)$
  - $(\nu\ x)(P + Q) \equiv P + (\nu\ x)Q$ *if* $x \notin fn(P)$
  - $(\nu\ x)(\nu\ y)P \equiv (\nu\ y)(\nu\ x)P$
  - $(\nu\ x)[u = v]P \equiv [u = v](\nu\ x)P$ *if* $x \neq u$ *and* $x \neq v$
  - $(\nu\ x)[u \neq v]P \equiv [u \neq v](\nu\ x)P$ *if* $x \neq u$ *and* $x \neq v$

The behavior of processes is given by an operational semantics. That is to say, a number of rules that define how algebraic expressions should be translated to LTSs.

Definition 34 (π-calculus Operational Semantics)**.** *Let P, P', Q and Q' be processes, α be a prefix, and a, x and u be names. Then the* operational semantics of the π-calculus *is given by the following rules.*

$$\frac{P' \equiv P \quad P \xrightarrow{\alpha} Q \quad Q' \equiv Q}{P' \xrightarrow{\alpha} Q'}\ \text{STRUCT} \qquad \frac{}{\alpha.P \xrightarrow{\alpha} P}\ \text{PREFIX}$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}\ \text{SUM} \qquad \frac{P \xrightarrow{\alpha} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{P|Q \xrightarrow{\alpha} P'|Q}\ \text{PAR}$$

$$\frac{P \xrightarrow{a\langle x\rangle} P' \quad Q \xrightarrow{\overline{a}(u)} Q'}{P|Q \xrightarrow{\tau} P'\{u/x\}|Q'}\ \text{COM} \qquad \frac{P \xrightarrow{\alpha} P' \quad x \notin \alpha}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}\ \text{RES}$$

$$\frac{P \xrightarrow{\overline{a}(x)} P' \quad a \neq x}{(\nu x)P \xrightarrow{\overline{a}\nu x} P'}\ \text{OPEN} \qquad \frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}\ \text{MATCH}$$

$$\frac{P \xrightarrow{\alpha} P' \quad x \neq y}{[x \neq y]P \xrightarrow{\alpha} P'}\ \text{MISMATCH}$$

Notice that on all of these definitions, prefixes can have only one parameter. It is possible, however, to have prefixes with multiple parameters (the so called *polyadic* π-calculus) and define them in terms of these simple ones. It is this polyadic notation that we use in this report.

References

[1] Paulo Salem da Silva and Ana C. V. de Melo. A simulation-oriented formalization for a psychological theory. In Matthew B. Dwyer and Antonia Lopes, editors, *FASE 2007 Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 42–56. Springer-Verlag, 2007.

[2] Paulo Salem da Silva and Ana C. V. de Melo. Reusing models in multi-agent simulation with software components. In Müller Padgham, Parkes and Parsons, editors, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, volume 2, pages 1137 – 1144. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[3] Cedric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha*, pages 268–332. Springer-Verlag, 2000.

[4] Nigel Gilbert and Steven Bankers. Platforms and methods for agent-based modeling. *Proceedings of the National Academy of Sciences of the United States*, 99(Supplement 3), 2002.

[5] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. MASON: A new multi-agent simulation toolkit. 2004. http://cs.gmu.edu/ eclab/projects/mason/.

[6] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.

[7] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. 1996. Working Paper 96-06-042.

[8] Michael North, Nick Collier, and Jerry R. Vos. Experiences creating three implementations of the Repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation*, 16(1):1–25, 2006. http://repast.sourceforge.net/.

[9] Joachim Parrow. An introduction to the pi-calculus. In J. A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.

[10] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 1997.

[11] Burrhus Frederic Skinner. *Science and Human Behavior.* The Free Press, 1953.

[12] Jianrui Wang and Richard A. Wysk. A pi-calculus formalism for discrete event simulation. In *WSC '08: Proceedings of the 40th Conference on Winter Simulation*, pages 703–711. Winter Simulation Conference, 2008.

[13] Gerhard Weiss, editor. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, MA, USA, 1999.

[14] Danny Weyns, H. Van Dyke Parunak, Fabien Michel, Tom Holvoet, and Jacques Ferber. Environments for multiagent systems: State-of-the-art and research challenges. In Danny Weyns *et al.*, editor, *Proceedings of the 1st International Workshop on Environments for Multi-agent Systems (Lecture Notes in Computer Science, 3374)*, pages 1–47. Springer, 2005.