A Coordination Model to Specify Systems Including Mobile Agents [*]

P. Ciancarini, F. Franzè and C. Mascolo
Dip. di Scienze dell'Informazione, University of Bologna
Mura Anteo Zamboni, 7, I-40127 Bologna, Italy
phone: +39 51 354506, fax: +39 51 354510
e-mail: {ciancarini,franze,mascolo}@cs.unibo.it

Abstract

A coordination model provides a formal framework in which the interaction of active entities that we call agents can be expressed. A coordination model deals with the creation and destruction of agents, their communication activities, their distribution and mobility in space, as well as the synchronization and distribution of their actions over time. We show how a coordination model called PoliS offers a flexible basis for the description and the analysis of architectures of systems including mobile agents. We have developed a model checking technique for the automatic analysis of PoliS specifications.

1 Introduction

New computing paradigms based on code mobility need specification methods able to deal with the features of agent-based architectures [20]. For instance, since the original WWW architecture supports very limited forms of distributed programming, it is being extended with specific programming languages, like Java, which extends the functionality of WWW browsers to deal with agents called *applets*. Some multiuser applications need migratory agents, which need to be coordinated in their travelling over the network.

A coordination model provides a formal framework in which the interaction of software agents can be expressed [5]. A coordination model deals with the creation and destruction of agents, their communication activities, their distribution and mobility in space, as well as the synchronization and distribution of their actions over time.

In this paper we show how a coordination model called PoliS [7] can be used to specify and analyse software architectures including mobile agents: code mobility is represented in the coordination model and reasoning on formal properties can be supported by model checking.

Research in the field of software architecture has led to the definition of architecture description languages (ADL) exploiting well-known formalisms like for instance CSP [1] and the π-calculus [16]. In [14] also the Chemical Abstract Machine (CHAM), a well known coordination model, has been used as an ADL. Using the CHAM the state of a system is represented by a chemical solution (a multi-set of terms of a word algebra) whose transformation is operationally defined by the application of multi-set rewriting (reaction) rules. The CHAM was originally introduced for representing concurrent computations [3]. In fact, the CHAM is also a simple coordination model to describe and control coordination and interaction among agents [2]. In our knowledge these notations do not support code mobility. A paper which analizes a number of formal models suitable for mobility is [10].

The paper is organized as follows: Section 2 introduces PoliS; Section 3 studies how PoliS can be used to describe software architectures including mobility. We describe the architecture of the "Meeting Scheduler System" case study as a system including mobile agents. In Section 4 we analyse the case study using a model checker. Finally, in Section 5 we outline our future work.

2 Overview of PoliS

PoliS is a coordination model based on multiple tuple spaces [13]. A tuple space, or *space* for short, includes both tuples and other spaces. PoliS specifications are hierarchically structured: a PoliS specifica-

[*]Copyright 1998 IEEE. Published in the Proceedings of IWSSD9, April 1998 Isobe, Japan. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

tion denotes a tree of nested spaces that dynamically evolves in time.

A PoliS space can contain both other spaces and tuples of two types: *ordinary tuples*, which are ordered sequences of values, and *program tuples*, which contain the coordination rules that manage activities inside the space they belong to. The execution of a program tuple is an action, which can modify a space tree removing tuples and adding tuples and spaces. However, an action can only handle the tuples of the space it belongs to *and* the tuples of its parent space. This constraint precisely defines both the "input" and the "output" environment of any action, as represented by a program tuple.

The typical structure of a nested multiple tuple space is graphically shown in Figure 1. In such a figure any ellipse represents a tuple space, any ordered sequence of values (for example $(5, 6)$) is an ordinary tuple and any tuple $(''r'' : R)$ is a program tuple; nested ellipses represent nested spaces.

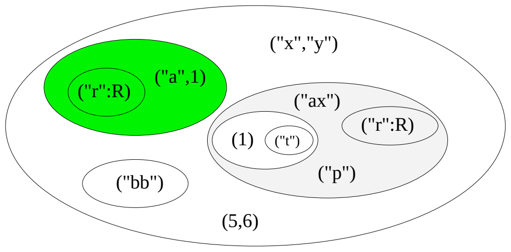

Figure 1: A PoliS space tree

A space is a multi-set of tuples. A space is modified by reactions that transform multi-sets of tuples in multi-sets of tuples (this is multi-set rewriting, and is common to most coordination models based on *generative communication* [5]). The *rule* is the construct that defines which reactions can take place. A rule can act on the tuples of the space in which it resides and in the tuples of the parent space of this space: we will call this spaces the rule scope. A rule defines a reaction that reads and consumes tuples in its scope, performs a sequential computation, produces new tuples in its scope and creates new subspaces.

More precisely, a rule is made up of a *preactivation*, a *local computation*, and a *postactivation*. The preactivation is a multi-set of tuples to be found in its scope; the local computation is any sequential computation which does not modify the tuple space; the postactivation is made up of a multi-set of tuples to be produced in its scope and of a set of spaces to be created. Notice that this is a very general definition; actually rules need not to be made up of all the admitted components: a rule can have an empty preactivation, it can involve no local computation, it can produce no tuples and it can create no spaces.

The preactivation can include *formal tuples*, that are tuples whose fields can be identifiers; moreover, it includes the primitive **ask**, that permits to check the values that are assigned to the identifiers of a formal tuple matched against a tuple in the space.

The semantics of a program tuple PT is that a reaction takes place in a space if the space itself includes both PT and a multi-set of tuples matching the preactivation of PT. A *match* relation checks whether a multi-set of formal tuples M_{ft} can be instantiated by a multi-set M_{gt} of ground tuples. Consequently, such a *match* relation is defined between pairs of multi-sets of tuples and not between pairs of tuples.

The tuples of the preactivation must be read or consumed in the rule scope. When a rule can be activated in a space, the reaction can takes place: the tuples to be consumed locally are removed from the space where the reaction takes place, the tuples to be consumed externally are removed from the parent space of the space where the reaction takes place, the local computation is performed, the tuples and the new spaces of the postactivation are created.

A program tuple is a multi-set rewriting rule: preactivation and postactivation are multi-sets and the local computation is written as annotation on the arrow between preactivation and postactivation. A tuple in the preactivation must be read if the symbol "?" is put in front of it and must be consumed otherwise; a read or consume operation involves the parent space if the symbol "↑" is put in front of a tuple and involves the local space if the symbol is missing; a tuple in the postactivation must be produced in the parent space if the symbol " ↑" is put in front of it and must be produced locally otherwise.

Rules are first class entities in PoliS: in fact, they are themselves part of spaces as (program) tuples that can be read, consumed or produced just like ordinary tuples. A program tuple has the form (*rule_id*: *rule*) where *rule_id* is a rule identifier and *rule* is a PoliS rule. A program tuple has an identifier which simplifies reading or consuming program tuples. Whenever disjoint multi-sets of tuples satisfy the activation preconditions of a set of rules, such rules can be executed independently and simultaneously: every rule modifies only the portion of space containing the tuples that must be read or consumed and therefore other rules can modify other tuples in the space or other spaces. In PoliS rules specification is written below the specification of the space containing it.

A simple example helps in explaining both syntax and semantics of PoliS. Let us consider a producer-consumer system. Such a system can be described by a space tree where the producer and the consumer are associated to two distinct spaces both included in the main space containing also the buffer represented by tuples generated by the producer. Such a system is graphically shown in Figure 2.

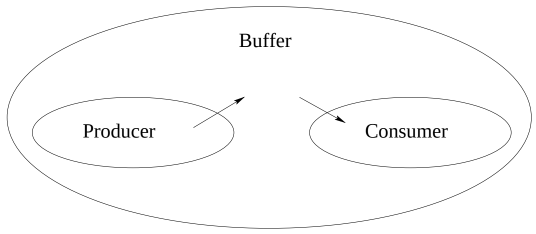

Figure 2: Producer-consumer: spaces topology

Table 1 contains the specification of the producer-consumer system. The *StartContext* space is the main space, that contains the initial tuple s and rules presents. The program tuple (“r_g” : R_g) indicates that the rule R_G, specified below in Table 1, is contained in the main space.

A key feature in PoliS is that a space tree can evolve dynamically: a new space is created by the primitive **tsc** (for *tuple space create*) and any space can be removed because of the execution of a special rule named *invariant* that terminates the space where it is executed. The execution of a rule containing a $\mathbf{tsc}(M)$ operation in its postactivation causes the multi-set M to be added as a child space of the space where the rule was executed.

For instance, the rule R_g of Table 1, contained in the main space, creates a space tree representing the producer-consumer system. Such a rule creates the spaces S_p and S_c that contain the tuples describing the producer and the consumer, respectively.

S_p is the producer space and contains the tuples ("$next_p$", i), that is a counter, and the rule R_p.

R_p emits a new produced item (tuple) in the main space: ($\uparrow$“*prod*”, i, p) and updates the counter increasing it by one.

S_c is the consumer space. It contains a counter and the rule R_c. R_c consumes a tuple from the main space and increases the counter by one.

In order to partially constrain activities inside a tuple space we can define one or more *invariants*, namely constraints that must hold for all the tuple space lifetime. Whenever an invariant is violated, the tuple space terminates and disappears. A PoliS invariant is a condition on the tuple space contents: it asserts that the space will never contain a given multi-set of tuples. Invariant rules can only read tuples locally (the tuples that must not belong to the tuple space) and produce tuples in the parent space. When the tuples to be read are in the space, the reaction specified by the invariant takes place in the usual way. Local computation and tuple production are used to communicate possible results to the parent space and then the space dies. Invariants are given by means of special program tuples whose names are replaced by the keyword **invariant**.

Going back to our example, if we want the consumer computation to terminate as soon as it receives an item containing the value 0, we put the invariant shown in Table 1 in the consumer space. The invariant fires when the consumer space contains a tuple ("*prod*", i, 0). Tuple ("*done*") represents a termination signal sent by the consumer to the parent space.

Communication is inspired by Linda [12]: tuples representing messages are put in a space shared by agents which have to communicate. Hence, communication is decoupled because agents do not know each other, since they access tuples by pattern matching. Since messages have no destination address, their contents determine the set of possible receivers, (communication is property driven).

In summary, a space represents at the same time both an agent performing a (chemical) computation and a persistent, multicast channel supporting communication among agents it contains.

3 The specification of an architecture with mobile agents

We use mobile agents to specify the “Meeting Scheduler System” (case study for IWSSD9, url: `http://salab-www.cs.titech.ac.jp/iwssd9.html`).

We first give an informal description (Sect. 3.1), then a PoliS specification (Sect. 3.2).

3.1 The Meeting Scheduler System: an informal description

An organization manages meetings as follows. A meeting initiator asks all potential attendees for the following information included in their personal agendas:

- a set of dates on which they cannot attend the meeting (exclusion set);
- a set of dates on which they would prefer the meeting to take place (preference set). For simplicity, and withous loss of generality, we assume that all days outside the exclusion set and not yet fixed for a meeting are free and represent the preference set.

$StartContext$

$$StartContext = \{\!|\ (``r_g" : R_g)\ |\!\}$$

$$R_g = \{\!|\ (''r_g'' : R_g)\ |\!\} \longrightarrow \{\!|\ \mathbf{tsc}(S_p), \mathbf{tsc}(S_c)\ |\!\}$$

S_p

$$S_p = \{\!|\ (''next_p'', 0), (''r_p'' : R_p)\ |\!\}$$

$$R_p = \{\!|\ (''next_p'', i_1)\ |\!\} \xrightarrow{(i_2)\leftarrow f(i_1)} \{\!|\ \uparrow(''prod'', i_2, p), (''next_p'', i_2)\ |\!\}$$
$$\textbf{where } f(i_1) = (i_1 + 1)$$

S_c

$$S_c = \{\!|\ (''next_c'', 0), (''r_c'' : R_c), (\mathbf{invariant} : R_{inv})\ |\!\}$$

$$R_c = \{\!|\ (''next_c'', i_1), \uparrow(''prod'', i_1, p)\ |\!\} \xrightarrow{(i_2)\leftarrow f(i_1)} \{\!|\ (''prod'', i_1, p), (''next_c'', i_2)\ |\!\}$$
$$\textbf{where } f(x) = (x + 1)$$

$$R_{inv} = \{\!|\ ?(''prod'', i, 0)\ |\!\} \longrightarrow \{\!|\ \uparrow(''done'')\ |\!\}$$

Table 1: Specification of the Producer-Consumer System

The proposed meeting date should belong to none of the exclusion sets and to as many preference sets as possible. A date conflict occurs when no date can be found. Conflicts can be resolved in two ways:

- some participants remove some dates from their exclusion set;
- some participants withdraw from the meeting.

The system should assist users in the following activities.

- Plan meetings consistently, using the constraints expressed by participants.
- Re-plan a meeting dynamically (to offer flexibility). Participants should be allowed to modify their exclusion and preference sets before a meeting date is decided. A meeting date initially found may need to be modified; sometimes the meeting may even be cancelled.
- Support conflict resolution according to some arbitrary resolution policies.

The meeting scheduler system must in general handle several meeting requests in parallel. Meeting requests can compete by overlapping in time: concurrency must thus be managed.

3.2 Our specification using PoliS

The "Meeting Scheduler System" specification document in PoliS is organized as follows: every initiator of a meeting is associated to a multi-set of tuples representing a mobile agent. Several agents (one for each meeting) can run in parallel.

Each initiator agent moves among the sites of participants collecting free dates and trying to decide a date (see Fig. 3). For simplicity we assume that a meeting can take place only if all potential attendees will participate.

An agent collects information inside a participant space, then it is frozen and moved outside: we call this phase *serialization*, because it is similar to what happens to a Java object which moves over the Internet [15].

Table 2 shows our specification, that includes three kinds of spaces. The $StartContext$ is the initial space: it includes p participants and n agents, one for each meeting. Each *participant* space has an initial state consisting of tuples representing its agenda: some days are marked "free" and other are marked "exclusion", meaning that these dates are in the participant exclusion set (we implicitly assume that the number of meetings (n) is less or equal to the number of days in the agenda (m)). Agendas are represented by multisets after $\oplus$ operator in the $StartContext$ definition.

StartContext

$$StartContext = \left\{\!\left|\begin{array}{l} Participant_1 \oplus \{\!|(\text{“}day\text{”}, 1, \text{“}free\text{”}), \ldots, (\text{“}day\text{”}, m, \text{“}exclusion\text{”})|\!\}, \\ Participant_2 \oplus \{\!| \ldots |\!\}, \ldots \\ Participant_p \oplus \{\!|(\text{“}day\text{”}, 1, \text{“}exclusion\text{”}), \ldots, (\text{“}day\text{”}, m, \text{“}free\text{”})|\!\}, \\ Agent_1, \ldots, Agent_n, \\ (\text{“}start\text{”}), \ldots, (\text{“}start\text{”}), \\ (\text{“}end\text{”} : END) \end{array}\right|\!\right\}$$

$$END = \{\!| \; (\text{“}frozen\text{”}, k, s_k) \; |\!\} \xrightarrow{(day) \leftarrow f(s_k)} \{\!| \; (\text{“}end\text{”}, day, k) \; |\!\}$$
where $f(x) = (f_{end}(x))$

Participant$_i$

$$Participant_i = \left\{\!\left|\begin{array}{l} (\text{“}get\text{”} : GETAG), (\text{“}push\text{”} : PUSHAG), (\text{“}extend\text{”} : EXTEND), \\ (\text{“}accept\text{”}) \end{array}\right|\!\right\}$$

$$GETAG = \left\{\!\left|\begin{array}{l} \uparrow(\text{“}frozen\text{”}, h, s_h), \\ \uparrow(\text{“}agent\text{”} : a), (\text{“}accept\text{”}) \end{array}\right|\!\right\} \longrightarrow \left\{\!\left|\begin{array}{l} (\text{“}frozen\text{”}, h, s_h), \\ (\text{“}agent\text{”} : a), (\text{“}agent\text{”}) \end{array}\right|\!\right\}$$

$$PUSHAG = \left\{\!\left|\begin{array}{l} (\text{“}frozen\text{”}, h, s_h), \\ (\text{“}agent\text{”} : a), (\text{“}go\text{”}) \end{array}\right|\!\right\} \longrightarrow \left\{\!\left|\begin{array}{l} \uparrow(\text{“}frozen\text{”}, h, s_h), \\ \uparrow(\text{“}agent\text{”} : a), (\text{“}accept\text{”}) \end{array}\right|\!\right\}$$

$$EXTEND = \left\{\!\left|\begin{array}{l} (\text{“}day\text{”}, d, \text{“}exclusion\text{”}), \\ ?(\text{“}accept\text{”}) \end{array}\right|\!\right\} \longrightarrow \{\!| \; (\text{“}day\text{”}, d, \text{“}free\text{”}) \; |\!\}$$

Agent$_i$

$$Agent_i = \left\{\!\left|\begin{array}{l} (\text{“}resume\text{”}), (\text{“}start\text{”} : START), (\text{“}resume\text{”} : RESUME), (\text{”}update\text{”} : U) \\ (\textbf{invariant} : EXIT), (\text{“}withdraw\text{”} : WITHDRAW), (\text{“}agent\text{”} : AGENT) \end{array}\right|\!\right\}$$

$$START = \{\!| \; \uparrow(\text{“}start\text{”}), (\text{“}resume\text{”}) \; |\!\} \longrightarrow \left\{\!\left|\begin{array}{l} (\text{“}done\text{”}), \\ (\text{“}M\text{”}, 1, 0), \ldots, (\text{“}M\text{”}, m, 0) \end{array}\right|\!\right\}$$

$$AGENT = \left\{\!\left|\begin{array}{l} (\text{“}agent\text{”} : AGENT), ?(\text{“}frozen\text{”}, k, s_k), \\ ?(\text{“}agent\text{”}) \end{array}\right|\!\right\} \longrightarrow \{\!| \; \textbf{tsc}(Agent_k) \; |\!\}$$

$$RESUME = \left\{\!\left|\begin{array}{l} \uparrow(\text{“}frozen\text{”}, i, s_i), \\ (\text{“}resume\text{”}), \uparrow(\text{“}agent\text{”}) \end{array}\right|\!\right\} \xrightarrow{(d_1, \ldots, d_m) \leftarrow f(s_i)} \left\{\!\left|\begin{array}{l} (\text{“}M\text{”}, 1, d_1), \ldots, (\text{“}M\text{”}, m, d_m), \\ (\text{“}self\text{”}, i), \uparrow(\text{“}go\text{”}) \end{array}\right|\!\right\}$$
where $f(x) = (deserialize_1(x), \ldots, deserialize_m(x))$

$$U = \left\{\!\left|\begin{array}{l} \uparrow(\text{“}day\text{”}, 1, d_1), \ldots, \uparrow(\text{“}day\text{”}, m, d_m), \\ (\text{“}M\text{”}, 1, v_1), \ldots, (\text{“}M\text{”}, m, v_m), \\ ?(\text{“}self\text{”}, me) \end{array}\right|\!\right\} \xrightarrow{(\overline{e}, \overline{w}) \leftarrow f(\overline{d}, \overline{v}, me)} \left\{\!\left|\begin{array}{l} \uparrow(\text{“}day\text{”}, 1, e_1), \ldots, \uparrow(\text{“}day\text{”}, m, e_m), \\ (\text{“}M\text{”}, 1, w_1), \ldots, (\text{“}M\text{”}, m, w_m), \\ (\text{“}done\text{”}) \end{array}\right|\!\right\}$$
where $f(\overline{x}, \overline{y}, z) = (if\ (\forall j x_j \neq z \wedge k = min\{j | x_j = \text{“}free\text{”}\}) then\ (\overline{x}_{|x_k = z}, \overline{y}_{|y_k = y_k + 1})\ else\ (\overline{x}, \overline{y}))$

$$WITHDRAW = \left\{\!\left|\begin{array}{l} \uparrow(\text{“}day\text{”}, h, me), (\text{“}M\text{”}, h, n_1), \\ ?(\text{“}self\text{”}, me), \textbf{ask}(n_1 > 0) \end{array}\right|\!\right\} \xrightarrow{(n_2) \leftarrow f(n_1)} \left\{\!\left|\begin{array}{l} \uparrow(\text{“}day\text{”}, h, \text{“}free\text{”}), \\ (\text{“}M\text{”}, h, n_2), (\text{“}done\text{”}) \end{array}\right|\!\right\}$$
where $f(x) = (x - 1)$

$$EXIT = \left\{\!\left|\begin{array}{l} ?(\text{“}M\text{”}, 1, v_1), \ldots, ?(\text{“}M\text{”}, m, v_m), \\ ?(\text{“}done\text{”}) \end{array}\right|\!\right\} \xrightarrow{(s) \leftarrow f(v_1, \ldots, v_m)} \left\{\!\left|\begin{array}{l} \uparrow(\text{“}frozen\text{”}, i, s), \\ \uparrow(\text{“}agent\text{”} : AGENT) \end{array}\right|\!\right\}$$
where $f(x_1, \ldots, x_m) = (serialize(x_1, \ldots, x_m))$

Table 2: PoliS specification of the system

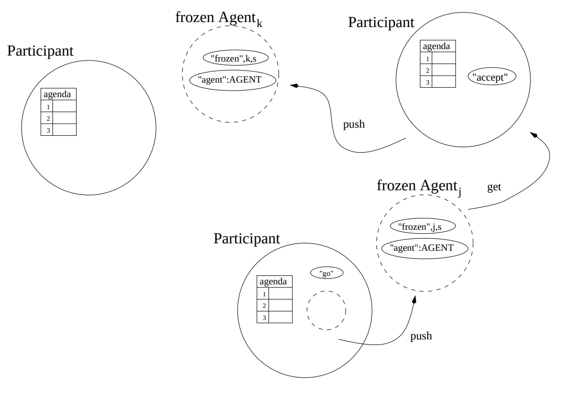

Figure 3: Agents System Architecture

Tuples (“*start*”) are consumed by agents to serialize themselves and start migrating (see rule $START$ in the *Agent* space).

The *StartContext* space includes only one program tuple (“*end*” : END): the rule END associated to the program tuple checks that all the potential attendees will participate, that is the condition for the meeting to take place (the function f_{end} checks if the number of participants has reached a prefixed number and outputs a date).

Each *participant* space can accept incoming *agents*. It contains some program tuples to activate the following rules. The rule $GETAG$ allows the agent to enter in a space. It consumes the tuples (“$frozen$”$, h, s_n$) and (“$agent$” : a) from the main space and generates them locally. It also consumes the (“*accept*”) tuple locally and generates the tuple (“*agent*”), meaning that the frozen agent has been entered in the local space.

The rule $PUSHAG$ moves the agent out of a space. It moves the tuples (“$frozen$”$, h, s_h$) and (“$agent$”) to the main space. Fig. 3 shows the actions of the two rules.

Participants can extend the set of possible dates using the rule $EXTEND$, to solve conflicts that can arise. It simply decides to free a date removing the tuple (“*day*”$, d,$ “*exclusion*”) and emitting (“*day*”$, d,$ “*free*”).

The *Agent* space contains some rules and an invariant rule to make the agent to freeze. The rule $START$ fires an agent to build a calendar (i.e. the tuples (“M”$, d_i, i$)).

The rule $AGENT$ generates (by **tsc**) a new agent space inside the participant space (Fig. 4).

The first rule enabled in a new agent space, inside a participant space, is $RESUME$: it is used to get and deserialize the frozen state of the agent. It emits a tuple (“*go*”) enabling rule $PUSHAG$ for a next move. A deserialized agent contains also rule U (Update) and rule $WITHDRAW$. The rule U updates the agenda of a participant using a policy that works as follows: a participant takes the first free date and books it, if it exists; a participant cannot book more than one date. Rule U also updates the internal agent table [1], represented by tuples like (“M”$, d, v$) where d is a day and v is the number of potential attendees.

In Fig. 4 an updating is shown: the participant agenda is updated booking day “1” with the name of the meeting (i.e. the name of the agent): “Z”, and increasing by 1 the counter of the meeting potential attendees for day “1” in the agent table (that now is 2).

The rule $WITHDRAW$ models a withdrawing from a meeting by a participant. It consumes the tuple (“*day*”$, h, me$) and emits a tuple (“*day*”$, h, free$) in the *Participant* space. It also decreases the number of supposed participants to the meeting h (i.e. it consumes the tuple (“M”$, h, , n_1$) and emits the tuple (“M”$, h, , n_2$) where $n_2 = n_1 - 1$.

The rule $EXIT$ is an *invariant* rule (see section 2 for its semantics). It terminates the agent space, by freezing the agent and moving it outside: this is performed producing a tuple that represents the serialized state (“$frozen$”$, i, s$) and a tuple (“$agent$” : $AGENT$) for re-generating an *Agent* space.

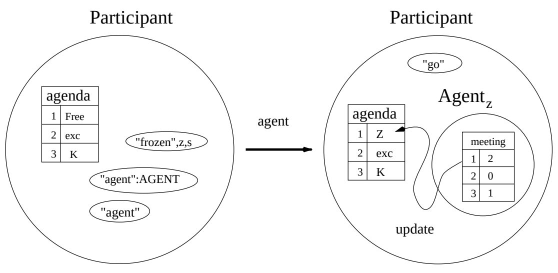

Figure 4: Deserialized agent performing update

4 Architectural Analysis in PoliS

In a previous work [7] a mapping between PoliS operational semantics and TLA (Temporal Logic of Action) has been studied. This allowed us to use a theorem prover for formal reasoning on PoliS specifications.

In this work instead we exploit a model checking technique to perform architectural analysis on PoliS specification documents.

[1]Each agent tries to establish a single meeting and the table contains for each date the number of participants that would accept that date

Model checking was introduced for hardware system verification [8]. Recently it has been used also for software systems, but we know only one previous example in which it has been used with a coordination model [6]. We explore its use for architectural analysis.

The aim of model checking is to find an assignment (*model*) for system variables that satisfies some formulae describing some system properties. Given an operational specification of a software system, a model checker builds a model and then it makes an exhaustive checking of variable values. This method could seem trivial and inefficient, but it is very powerful for systems with finite state models.

4.1 A Temporal Logic and a Model Checker for PoliS

In order to specify the architectural properties to be proved, we introduce a temporal logic. The PoliS Temporal Logic (PTL) is a CTL [8] dialect. The main differences between PTL and CTL depend on the definition of our model, that is based on multisets (spaces): all formulae are evaluated in a context (a space); we also assume that formulae without an explicit context are evaluated in the *StartContext*. An atomic proposition *atom* is a tuple; we say that proposition *atom* is true in a context C if it belongs to space denoted by constant C. We have also added the classical logic operators and some temporal operators to improve formulae representation and understanding.

In table 3 we sketch the PTL syntax.

- A *ptf* can be a *temporal*, a *classic*, a parenthesized *ptf*, an *atom*, a *ptf* can be universally or existentially quantified inside *range* over some variables;
- a *context* is a PTL formula that has a pattern like: $ptf \in C$ (space C), $ptf \in \star C$ (all C spaces), $ptf \in \& C$ (some C spaces), or $ptf \in \% C$ (exactly one C space), these because in a specification it can be more than one instance of the same space;
- a *temporal* is a CTL formula: the canonical operators **A** (for all paths) and **E** (at least a path does exist) for path quantification are described respectively by symbols ⋆ and &. **X** and **U** are PTL symbols for CTL operators Next and Until;
- ⋆◇*ptf* is defined as ⋆(true**U***ptf*): it means "*for all paths ptf will be eventually true*";
- &◇*ptf* is defined as &(true**U***ptf*): it means "*for at least a path ptf will be eventually true*";
- ⋆□ptf is defined as ¬&◇¬ *ptf*: it means that "*for all paths ptf is always true*";
- &□*ptf* is defined as ¬⋆◇¬ *ptf*: it means that "*for at least a path ptf is always true*";

ptf	::=	context |
		temporal |
		classic |
		(ptf) |
		atom |
		∀ i range (ptf) |
		∃ i range (ptf)
range	::=	ϵ |
		∈ [min,max]
context	::=	ptf ∈ C |
		ptf ∈ ⋆C |
		ptf ∈ &C |
		ptf ∈ %C
temporal	::=	⋆**X**ptf |
		&**X**ptf |
		⋆(ptf **U**ptf) |
		&(ptf **U**ptf) |
		⋆◇ptf |
		&◇ptf |
		⋆□ptf |
		&□ptf |
		ptf ⇝ptf
classic	::=	ptf ∧ptf |
		ptf ∨ptf |
		¬ptf |
		ptf ⇒ptf
atom	::=	tuple

Table 3: PTL syntax

- $ptf \leadsto ptf'$ is defined as $\star\Box(ptf \Rightarrow \star\Diamond ptf')$: it means that "*for all paths it is always true that ptf implies that for al least a path ptf' will be eventually true*";
- a *classic* is a PTL formula with classical logic operators;
- an *atom* is simply a tuple.

PoliMC is a model checker for PoliS. The model checker gets two inputs: a system specification written in PoliS, and a set of properties to be verified written in PTL. PoliMC first parses the PoliS specification and builds up a model for it. Starting from the SOS formal operational semantics of PoliS we also defined a transition system. The graph obtained from unfolding a transition system of a real system is something quite similar to our model of the system.

The main difference between SOS unfolding and our model is that in SOS a unique monolithic graph is built to represent the system, while we have a graph for each space definition. Nodes show how spaces evolve and edges are labelled with tuples produced/consumed and tested in the parent spaces. PoliMC works recursively starting from the more nested spaces, going up to the root space (*StartContext*), using the information collected during the visit.

After having built the graph, the checker parses PTL formulae and builds syntax trees including only CTL operators, finally PoliMC can start performing model checking. Its algorithm follows the guidelines given in [8]: the checking is performed recursively starting from simpler sub-formulae (which are deeper in a syntactic tree), a difference to remark is that each formula is checked inside its context, that is the model checker make the checks using the graph of the space (context).

4.2 Analysis of the Meeting Scheduler System

We have used PoliMC to analyse some liveness properties. For instance, we would like to prove that an agent will be able to establish a meeting date, or we would like to prove properties on the migration of an agent inside/outside the components.

Formally we can write:

$$End = (\forall agent(\exists day((\text{"}end\text{"}, day, agent) \in \ StartContext)))$$

$$Move = (((\text{"}done\text{"}) \in \ \&Agent) \in \ \&Participant)$$

End states that each agent finds a date for its meeting (i.e. all meetings are arranged).

Move states that an agent is inside a participant space and it has performed some actions.

If we study a configuration where the number of meetings to be arranged (i.e. the number of agents), is smaller than the available days. We would like to verify the following:

$$\star\Box\star\Diamond End \qquad (1)$$

That is: infinitely often *End* will be valid. However PoliMC shows that (1) is false. To understand this we can think about a scenario where agents are not able to agree, choosing the same date and then withdrawing it. By the way, PoliMC also verifies the falsity of:

$$\star\Box\star\Diamond Move \qquad (2)$$

Property (2) states that infinitely often *Move* is valid (i.e. agents can move indefinitely). Falsity of (2) guarantees that this cannot happen, so we have a scenario where all meeting are arranged. PoliMC can verify that this property is not true if we take a number of meeting (agents) greater than available days.

Instead, we can verify the following formula:

$$\star\Box\star\Diamond(End \vee Move) \qquad (3)$$

That is, infinitely often some agents move or all meetings are arranged. This shows that the system cannot deadlock.

Properties (1) and (2) above cannot help us to guarantee progress, but we can verify:

$$\star\Box\&\Diamond End \qquad (4)$$

It states that from all states of all paths (always) we can find at least one path where *End* is eventually valid. Formula (4) is quite different from (1). While (1) states that in all cases meeting *will* be arranged, (4) states that in all case meeting *could* be arranged. In order to ensure that all meeting will be arranged we need a fairness condition like:

$$\star\Box\&\Diamond End \Rightarrow \star\Box\star\Diamond End \qquad (5)$$

If from all states of all paths we can find at least one path in which *End* is eventually valid then *End* is valid infinitely often. In other words if we are always in condition to arrange all meetings, then this will eventually happen.

Using (4) in conjunction with (5) leads to the verification of (1), (i.e. the system infinitely often comes to *End*, but given that *End* is true in states without exit transitions we can state that system always comes to *End*).

We remark that if we remove some rules used to resolve conflicts (like rule $WITHDRAW$ or rule $EXTEND$), (4) is not verified, that is, there are some states where no path brings to End. In other words, sometimes a system can reach a state from which it is impossible to arrange some meetings, and some agents move indefinitely.

5 Related Work and Conclusions

We have shown how we use PoliS to specify and analyse a system including mobile agents. The idea consists of having a coordination model able to express a dynamic topology of spaces (multi-sets) which are active themselves, and can move. We analyse PoliS documents with a model checker able to prove or disprove some formal properties of the system being specified.

PoliS is based on multiset rewriting like the CHAM. A CHAM specification of code mobility would be quite complex because would require some coding of programs and workspaces using the *airlock* and *membrane* constructs, whose properties are not simple. We have shown that the PoliS notation allows the specification of code mobility and agent migration in a natural way.

There is a growing interest in formal methods for specifying logical (code) or physical (user) mobility. For instance, in [19, 17] Mobile Unity has been used to specify mobility of code: the idea is to extend Unity with features for describing localities and agent migration. The Mobile Unity notation provides a logic framework to perform analysis, however we have not seen any analysis (either automatic or manual) applied on the specification of systems including mobile agents.

We are currently interested in developing PoliS in several directions.

A current line of research investiogates how PoliS compares with other formalisms for mobility, like the Join Calculus [11] and the Ambient Calculus [4]. Moreover, we are also trying to specify a number of different mobility paradigms in PoliS.

Security issues are an important problem in systems supporting mobile agents. In [9] a typed process calculi for multiple tuple spaces that also addresses security aspects is presented. We could use a similar approach for PoliS. Moreover, we remark that PoliS rules cannot be changed dynamically because rules are encoded in the specification: a program tuple (“r” : R) refers to a given specification of R. A program tuple (“r” : R) can be consumed but no new tuple containing non-specified “programs” can be introduced. These features simplify the task of extending model checking to deal with security issues.

We are investigating if the PoliS model could fit for analysis of security issues; we are especially interested in studying if we can use the model checker to verify security properties on a system.

Another feature which is missing is the ability to deal with typed data and agents [18]. Another activity consists of using PoliS as coordination model of a platform for programming mobile agents.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, June 1997.

[2] J. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Trends in Object-Based Concurrent Computing*, pages 257–280. MIT Press, Cambridge, MA, 1993.

[3] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[4] L. Cardelli. Mobile Ambient Synchronization. Technical Report SRC Tech Note 1997-013, Digital, July 1997.

[5] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[6] X. Chen, P. Inverardi, and C. Montangero. ESP-MC: An Experiment in the Use of Verification Tools. In K. Kanchanasut and J. Levy, editors, *Proc. Asian Computing Science Conference*, volume 1023 of *Lecture Notes in Computer Science*, pages 396–406, Thailand, Dec 1995. Springer-Verlag, Berlin.

[7] P. Ciancarini, M. Mazza, and L. Pazzaglia. A Logic for a Coordination Model with Multiple Spaces. *Science of Computer Programming*, 31(2/3):(to appear), July 1998.

[8] E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[9] R. DeNicola, G. Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In D. Garlan and D. LeMetayer, editors, *Proc. 2nd Int. Conf. on Coordination Models and*

Languages, volume 1282 of *Lecture Notes in Computer Science*, pages 220–237, Berlin, Germany, September 1997. Springer-Verlag, Berlin.

[10] G. DiMarzoSerugendo, M. Muhugusa, and C. Tschudin. An Survey of Theories for Mobile Agents. *Web Journal*, (to appear), 1998.

[11] C. Fournet, G. Gonthier, J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag, Berlin.

[12] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[13] D. Gelernter. Multiple Tuple Spaces in Linda. In E. Odijk, M. Rem, and J. Syre, editors, *Proc. Conf. on Parallel Architectures and Languages Europe (PARLE 89)*, volume 365 of *Lecture Notes in Computer Science*, pages 20–27. Springer-Verlag, Berlin, 1989.

[14] P. Inverardi and A. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[15] D. Lea. *Concurrent Programming in Java. Design Principles and Patterns.* Addison-Wesley, 1997.

[16] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153, Sitges, Spain, September 1995. Springer-Verlag, Berlin.

[17] G. Picco, G. Roman, and P. McCann. Expressing Code Mobility in Mobile Unity. In M. Jazayeri and H. Schauer, editors, *Proc. 6th European Software Eng. Conf. (ESEC 97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 500–518. Springer-Verlag, Berlin, 1997.

[18] J. Riely and M. Hennessy. A Typed Language for Distributed Mobile Processes. In *Proc. 25th ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[19] G. Roman, P. McCann, and J. Plun. Mobile UNITY: Reasoning and Specification in Mobile Computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, June 1997.

[20] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997.