Using a Coordination Language to Specify and Analyze Systems containing Mobile Components

P. CIANCARINI, F. FRANZÈ, and C. MASCOLO
Università di Bologna

New computing paradigms for network-aware applications need specification languages able to deal with the features of mobile code based systems. A coordination language provides a formal framework in which the interaction of active entities can be expressed. A coordination language deals with the creation and destruction of code or complex agents, their communication activities, as well as their distribution and mobility in space.

We show how the coordination language PoliS offers a flexible basis for the description and the automatic analysis of architectures of systems including mobile entities. PoliS is based on multiple tuple spaces and offers basis for defining, studying, and controlling mobility as it allows decoupling mobile entities from their environment both in space and in time. The pattern matching mechanism adopted for communication helps in abstracting from addressing issues. We have developed a model checking technique for the automatic analysis of PoliS specifications: in the paper we show how this technique can be applied to mobile code based systems.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specification; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*

General Terms: Coordination, Mobility, Specification, Verification

Additional Key Words and Phrases: Tuple Space, PoliS, Code Mobility, Temporal Logic

---

This article is an extended version of a paper published in the *Proceedings of the 9th IEEE Int. Workshop on Software Specification and Design (IWSSD-9)*, Ise-shima (Japan), April 1998. IEEE Computer Society Press.
This work was partially supported by the Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the 40% project "SALADIN".
Authors' addresses: Paolo Ciancarini, Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni, 7, I-40127 Bologna, Italy; e-mail: ciancarini@cs.unibo.it. Francesco Franzè, Dipartimento di Elettronica Informatica e Sistemistica, Università di Bologna, Viale Risorgimento 2 - I-40136 Bologna; e-mail: ffranze@deis.unibo.it. Cecilia Mascolo, Dept. Computer Science, University College London Gower Street, London WC1E 6BT, UK; e-mail: c.mascolo@cs.ucl.ac.uk.

1. INTRODUCTION

The complexity of distributed applications using global communication services represents a great challenge for software engineers. How can we build geographically distributed software systems and exploit the wide availability of hardware resources, economically? How can we deal with mobile, wireless, and remote computing [Bagrodia et al. 1995; Chess et al. 1995; Ramjee et al. 1995]?

Traditional computing models are evolving in order to match the novel requirements of a networked world offering global communication services. New computing paradigms based on *code mobility* need specification languages able to express the features of architectures supporting mobile software entities [Thorn 1997]. For instance, since the original WWW architecture supports very limited forms of distributed programming, it has been extended with network-aware programming languages, like Java, which extends the functionality of WWW browsers in order to support *applets*, that are made of code loaded on demand from a remote site. Some Java-based applications are even based on migratory agents, namely processes which can decide to move to another site in order to look for some resources, and which need to be controlled in their traveling over the network [Lange and Oshima 1998; Kiniry and Zimmerman 1997].

Formal models have been used in the specification of mobile code systems. In [Serugendo et al. 1998] a survey of different approaches to code mobility formalization is presented. The focus of our approach is related to the interaction of mobile entities with their environment and vice versa. This is a very complex issue as an architecture including mobile entities can be decomposed in at least three different layers:

—The *physical network*, made mostly of immobile hosts, reliable and fast connections. In this context a mobile entity consists of a piece of hardware using a wireless connection. The main problem at this level is that such a connection is usually unstable and offers low bandwidth.

—The *middleware network*, made mostly of immobile services or abstract machines, where a mobile entity consists of a process migrating from one host to another. Some problems at this level are resource management (e.g., load balancing among the available hosts), service reliability, and security.

—The *logical network*, made of application code scattered over the middleware network, where a mobile entity consists of an agent migrating from one abstract machine to another. An interesting problem at this level is how such a mobile entity can cooperate with other entities in the application and how the code can be moved or copied among the distributed entities.

In this paper we are interested in the last form of mobility, where the computing entities are usually called *software agents*. In any system including mobile entities an important issue concerns the modality of their interactions, both with other mobile entities and with the external environment. A *coordination language* provides a formal framework in which the interaction of software agents can be expressed [Ciancarini 1996]. A coordination language 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.

More precisely, a coordination language consists of:

(1) *Coordinables*. These are the entity types which are coordinated. These could be Unix-like processes, threads, concurrent objects, and even users.
(2) *Coordination Media*. These are the media enabling the communication among the entities. Moreover, a coordination medium can serve to aggregate entities which should be manipulated as a whole, dynamic configuration. Examples are the classic media like semaphores, monitors, or channels, or more complex media like tuple spaces, blackboards, pipelines, etc.
(3) *Coordination Laws*. A coordination language should dictate a number of laws to describe how entities coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws which enact either synchronous or asynchronous behaviors, or which exploit explicit or implicit naming schemes for coordination entities.

In this paper we show how the coordination language PoliS can be used to specify and analyze systems based on logical mobility: code mobility is represented as a first class concept. The coordination media in PoliS are multiple tuple spaces, which offer a natural basis for describing mobile entities and their dynamic rearrangement. The pattern matching mechanism adopted to access the tuple spaces helps in abstracting away from low level addressing issues. Code can be explicitly moved from one PoliS space to another, duplicated, and eliminated. Reasoning on the formal properties of a PoliS specification can be supported by either theorem proving [Ciancarini et al. 1998] or model checking. In [Ciancarini et al. 1998] we developed the basic PoliS semantics and a theorem prover for predicates based on Lamport's TLA; we applied that framework to the study of generic systems not including mobile components. In this paper we focus on PoliS features useful to describe mobile entities and use a different logic more adequate for model checking. Model checking allows a completely automatic reasoning. Furthermore, due to the modularity of the model checker we developed for PoliS, it is possible to prove properties on the global specified system or on parts of it.

The paper is organized as follows: Section 2 introduces the PoliS language. We then give the operational semantics for PoliS in Section 3. In Section 4 we show how PoliS can be used for the specification of mobility of code and agents. Section 5 describes the model checker we have built for PoliS. In Section 6 we model the architecture of a mobile "Meeting Scheduler System" [Feather et al. 1997] and analyze the system using our model checker. Section 7 discusses some related work. Finally, in Section 8 we outline some conclusions and future work.

2. OVERVIEW OF POLIS

PoliS is a coordination language whose coordination media are nested tuple spaces [Ciancarini et al. 1998]. A tuple space, or *space* for short, includes as coordinables both tuples and other spaces. PoliS specifications are modular and hierarchically structured: a PoliS specification denotes a tree of nested spaces that dynamically evolves over time. Figure 1.a shows a structure of nested spaces (i.e., the nested circles); Figure 1.b shows the corresponding tree whose nodes are the spaces in Figure 1.a. The two pictures represent the same concept. The labels inside the spaces represent tuples. A space can contain other spaces or tuples: *ordinary*

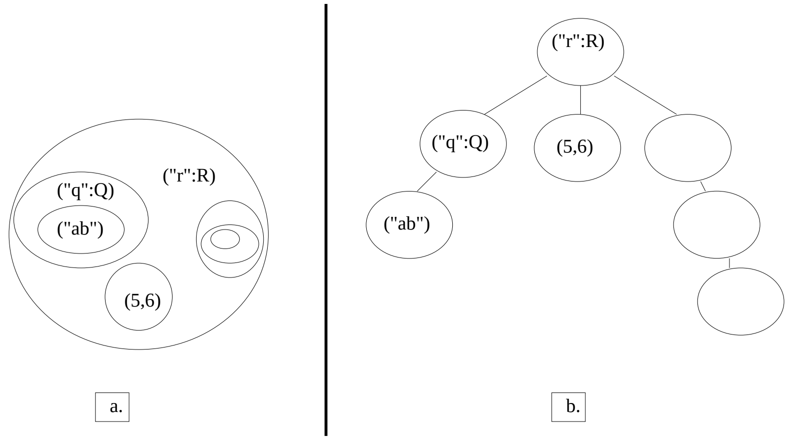

Fig. 1. PoliS nested spaces (a) and the corresponding tree interpretation (b).

tuples, which are ordered sequences of values, and *program tuples*, which contain the coordination rules that manage activities inside the space they belong to.

In Figure 1 ordered sequences of values (for example $(5, 6)$) are ordinary tuples; the tuples of the form (“r” : R) are program tuples. A program tuple (“r” : R) is composed of an identifier r and rule code represented by the placeholder R. The rule code defines which reactions can take place. The quoted notation “ ” is used to distinguish actual parameters from formal ones (i.e., the non quoted ones). The execution of a *program tuple* is an action which can modify a space tree by removing and adding tuples. However, an action can only handle the tuples of the space it belongs to *and* the tuples of its parent space. This precisely defines both the “input” and the “output” scope of any action, as represented by a program tuple. Figure 2 shows the scope of a program tuple (“r” : R). A space is modified by reactions that transform multi-sets of tuples into multi-sets of tuples (this is multi-set rewriting, common to most coordination models based on *generative communication* [Banâtre and LeMétayer 1996]). A rule defines a reaction that reads and/or consumes tuples in its scope, performs a sequential computation, and produces new tuples in its scope. More precisely, a rule consists of a *precondition*, a *local computation*, and a *postcondition*. The precondition is a multi-set of tuples to be found in the rule scope. The local computation is any sequential computation which does not modify the tuple space; it is encoded as a function that maps values of tuples of the precondition on values of tuples of the postcondition. The postcondition is made up of a multi-set of tuples to be produced in the rule scope. We remark that this is a very general definition; actually a rule can lack of some components: a rule can have an empty precondition, can involve no local computation, or can produce no tuples. The precondition can include *formal tuples*, i.e., tuples whose fields can be identifiers (i.e., the non quoted fields). In this case actual values for those identifiers are “matched” in the tuple space.

The tuples of the precondition must be read or consumed in the rule scope (Figure 2). When a program tuple is enabled, i.e., its precondition tuples exist in the program tuple scope, the reaction can take place: the tuples to be consumed lo-

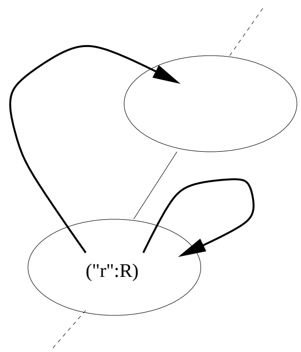

Fig. 2. The scope of a rule.

cally are removed from the space containing the program tuple, the tuples to be consumed externally are removed from the parent space of the space containing the program tuple, the local computation is performed, the tuples of the postcondition are produced. A tuple in the precondition 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 postcondition 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 PoliS rule code. The identifier simplifies reading or consuming program tuples and allows the existence of multiple copies of program tuples with the same code but different rule identifiers.

Rules can also create and destroy tuple spaces. They can generate new spaces using the primitive **tsc** (for *tuple space creation*) in the postcondition part. For example, the execution of a rule containing a $\mathbf{tsc}(M)$ operation in its postcondition causes the space M to be added as a child space of the space where the rule is executed. Spaces can also be destroyed by particular rules called *termination rules*. Whenever a termination rule is enabled the tuple space terminates and disappears.

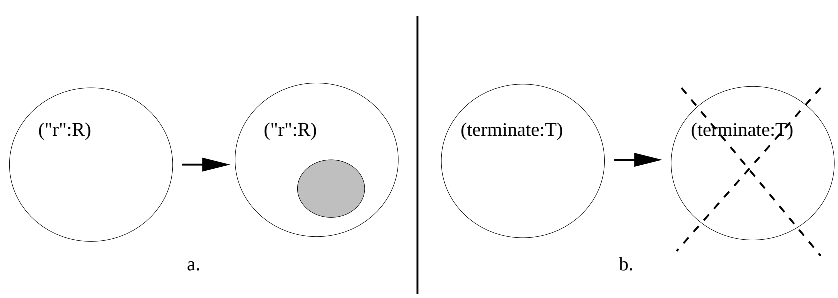

Fig. 3. Creation (a) and Termination (b) of spaces.

StartContext

$$StartContext = \{\!|\ (\text{“}create\text{”} : CREATE)\ |\!\}$$

$$CREATE = \{\!|\ (\text{“}create\text{”} : CREATE)\ |\!\} \longrightarrow \{\!|\ \mathbf{tsc}(Client), \mathbf{tsc}(Server)\ |\!\}$$

Client

$$Client = \{\!|\ (\text{“}idle\text{”}, 0), (\text{“}req\text{”} : REQ), (\text{“}get\text{”} : GET), (\mathbf{terminate} : END)\ |\!\}$$

$$REQ = \{\!|\ (\text{“}idle\text{”}, i)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}request\text{”}, i), (\text{“}wait\text{”}, i)\ |\!\}$$

$$GET = \{\!|\ (\text{“}wait\text{”}, i), \uparrow(\text{“}answer\text{”}, answ, i)\ |\!\} \xrightarrow{(j) \leftarrow f(i)} \{\!|\ (\text{“}idle\text{”}, j)\ |\!\}$$
where $f(x) = (x + 1)$

$$END = \{\!|\ ?(\text{“}idle\text{”}, 10)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}done\text{”})\ |\!\}$$

Server

$$Server = \left\{\!\!\left|\ \begin{array}{l} (\text{“}getreq\text{”} : GETREQ), (\text{“}idle\text{”}), \\ (\text{“}serve\text{”} : SERVE), (\text{“}put\text{”} : PUT) \end{array}\ \right|\!\!\right\}$$

$$GETREQ = \{\!|\ \uparrow(\text{“}request\text{”}, i), (\text{“}idle\text{”})\ |\!\} \longrightarrow \{\!|\ (\text{“}request\text{”}, i)\ |\!\}$$

$$SERVE = \{\!|\ (\text{“}request\text{”}, i)\ |\!\} \longrightarrow \{\!|\ (\text{“}answer\text{”}, answ, i)\ |\!\}$$

$$PUT = \{\!|\ (\text{“}answer\text{”}, answ, i)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}answer\text{”}, answ, i), (\text{“}idle\text{”})\ |\!\}$$

Table 1. Specification of a Client-Server System in PoliS.

Termination rules can read tuples only locally (i.e., not in the parent space, as the termination condition is meant to be local to the space configuration) and produce tuples in the parent space, as the local space disappears. When the tuples to be read are in the space, the reaction specified by the termination rule 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 terminates. Termination rules are given by means of special program tuples whose names are replaced by the keyword **terminate**. In Figure 3.a a new space is created upon the activation of rule R. In Figure 3.b a space is destroyed when the termination rule T is enabled.

A simple example helps in explaining both the syntax and the semantics of PoliS. Let us consider a client-server system. A client emits requests and a server serves them. Such a system can be described by two distinct spaces both included in the main space representing the client and the server.

Table 1 contains the specification of the system. The *StartContext* space is the main space, that contains the program tuple (“*create*” : $CREATE$). The name of the tuple is *create* (as it is quoted, it is the actual name of the tuple); instead $CREATE$ acts as a “macro”, expanded in the corresponding text below in the table. The rule denoted by $CREATE$ creates the spaces *Client* and *Server* that contain the tuples describing the client and the server, respectively. The rule also consumes the program tuple (“*create*” : $CREATE$) in order to ensure this rule is only applied once in the initialization phase. After that, the code of $CREATE$ will disappear and it will not be possible to apply the rule anymore.

$Client$ is the client space and contains the tuple $(\text{"}idle\text{"}, i)$ that indicates the state of the client, the program tuple $(\text{"}req\text{"} : REQ)$, $(\text{"}get\text{"} : GET)$, and the termination rule $(\textbf{terminate} : END)$ that contains the code of the rules REQ, GET, and END (specified below), respectively. The rule REQ emits a new request (tuple) in the main space: $\uparrow(\text{"}request\text{"}, i)$, and changes the state of the client from $(\text{"}idle\text{"}, i)$ to $(\text{"}wait\text{"}, i)$ where i is the number associated to the request. The rule GET waits for an answer in the main space $\uparrow(\text{"}answer\text{"}, answ, i)$ where i corresponds to the number of the request (the rule checks if the tuple $(\text{"}wait\text{"}, i)$ is present). It emits a new state tuple with the number i increased by one by the function f on the arrow in the rule (specified in the **where** clause). The $Client$ space terminates as soon as it receives the 10-th answer. The termination rule END checks if the $Client$ space contains the tuple $(\text{"}idle\text{""}, 10)$, that means that the client has received ten answers from the server. The tuple $(\text{"}done\text{"})$ represents a termination message sent by the consumer to the main space before dying.

$Server$ is the server space. It contains a tuple denoting its state and three rules: the rule $GETREQ$ checks if the state is idle and if a request is present in the main space, then moves the request in the local space. The rule $SERVE$ generates an answer to the request. The rule PUT resets the state of the server to idle (emitting the tuple $(\text{"}idle\text{"})$ locally), and move the answer tuple to the main space.

The example above shows that the basic communication mechanisms of PoliS are asynchronous. Rules are transactions and therefore execute in an atomic fashion. They also offer a basic mechanism for synchronization of operations: the rules can atomically read/consume multiple tuples allowing quite complex evolutions. Tuples representing messages are put in the environment by entities which have to communicate. Hence, communication is decoupled because communicating entities do not necessarily know each other; they access tuples by pattern matching. Messages have no destination address, so their contents determine the set of possible receivers. Thus, a space represents at the same time both a component performing computations and a persistent, multicast channel supporting communication among components it contains. Any space communicates with the parent space using a pattern matching mechanism, thus minimizing the assumptions over the the rest of the system.

In the next section we describe the formal operational semantics of PoliS.

3. ABSTRACT SYNTAX AND OPERATIONAL SEMANTICS FOR POLIS

We describe the semantics of a PoliS specification as the application of simple rewriting operations on multisets. In Table 2 we show the abstract syntax for PoliS. A multiset (MS) is composed of elements that are tuples or multisets, or can be built as the union or difference of multisets. A tuple can be a data tuple or a program tuple. A data tuple is a sequence of values and strings, whereas a program tuple is composed of an identifier and of rule code.

The semantics of PoliS is introduced in the Tables 3, 4, and 5. Table 3 shows the SOS (Structured Operational Semantics) axioms and rules:

- **L** is a rule describing the local computations. It formalizes the local and isolated

MS	::=	$\{\!\{$ elem $\}\!\}$ | MS $\oplus$ MS | MS $\setminus$ MS | (MS)
elem	::=	tuple | MS
tuple	::=	data | program
program	::=	("r" : $Code$)
data	::=	(datalist)
datalist	::=	"$data$" | $value$ | "$data$", datalist | $value$, datalist

"r" $\in$ $Ruleid$, the set of rule identifiers
$Code \in Rulecode$, the set of rules code specified in Table 4.
In the concrete syntax, $Code$ is usually substituted with a macro that expands in the code itself.
$value \in Values$
$data \in String$

Table 2. PoliS Abstract Syntax.

evolution of subspaces or subsets of tuples inside a space.

- The axioms **RL**, **RI**, and **T** define the semantics of PoliS rules. These axioms show the reaction taking place when a *program tuple* ("r" : R) is enabled in a space (M). The formalization of the code macro R is shown in Table 4 according to the type of R (an example of use of these macro can be found in Table 1). R can be a local rule (i.e., R_l), or an interactive rule (i.e., R_i), or a termination rule (i.e., R_{term}). The notation t_c, t_p, t_{ec}, t_{ep} denote the lists of tuples to be consumed locally, produced locally, produced and consumed in the parent space, respectively. $\overline{S}$ is the list of spaces to be created by the rule. $v_{\overline{x}}$ and $v_{\overline{y}}$ are the formal parameter lists to be substituted by $\overline{x}$ and $\overline{y}$.
  Table 3 shows these three types of semantics rules:

L: $\dfrac{MS \longrightarrow MS'}{\{\!\{MS\}\!\} \longrightarrow \{\!\{MS'\}\!\}}$

Local Rule

RL: $\{\!\{(\text{“}r_l\text{”} : R_l)\}\!\} \oplus M \longrightarrow$
$((\{\!\{(\text{“}r_l\text{”} : R_l)\}\!\} \oplus M) \setminus \{\!\{\overline{t}_c[\overline{v_{\overline{x}}}/\overline{x}]\}\!\}) \oplus \{\!\{\overline{t}_p[\overline{v_{\overline{x}}}/\overline{x}, \overline{v_{\overline{y}}}/\overline{y}], \overline{S}[\overline{v_{\overline{x}}}/\overline{x}, \overline{v_{\overline{y}}}/\overline{y}]\}\!\}$
if $LocEnabled(R_l, \text{“}r_l\text{”}, M, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}})$

Interaction Rule

RI: $\{\!\{\{\!\{(\text{“}r_i\text{”} : R_i)\}\!\} \oplus M_1\}\!\} \oplus M_2 \longrightarrow$
$\{\!\{((\{\!\{(\text{“}r_i\text{”} : R_i)\}\!\} \oplus M_1) \setminus \{\!\{\overline{t}_c[\overline{v_{\overline{x}}}/\overline{x}]\}\!\}) \oplus \{\!\{\overline{t}_p[\overline{v_{\overline{x}}}/\overline{x}, \overline{v_{\overline{y}}}/\overline{y}], \overline{S}[\overline{v_{\overline{x}}}/\overline{x}, \overline{v_{\overline{y}}}/\overline{y}]\}\!\}\}\!\}$
$\oplus(M_2 \setminus \{\!\{\overline{t}_{ec}[\overline{v_{\overline{x}}}/\overline{x}]\}\!\}) \oplus \{\!\{\overline{t}_{ep}[\overline{v_{\overline{x}}}/\overline{x}, \overline{v_{\overline{y}}}/\overline{y}]\}\!\}$
if $IntEnabled(R_i, \text{“}r_i\text{”}, M_1, M_2, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}})$

Termination Rule

T: $\{\!\{\{\!\{(\textbf{terminate} : R_{term})\}\!\} \oplus M_1\}\!\} \oplus M_2 \longrightarrow M_2 \oplus \{\!\{\overline{t}_{ep}[\overline{v_{\overline{x}}}/\overline{x}, \overline{v_{\overline{y}}}/\overline{y}]\}\!\}$
if $TermEnabled(R_{term}, M_1, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}}) \wedge \neg\exists R_t, \overline{v_{\overline{z}}}, \overline{v_{\overline{k}}} | TermEnabled(R_t, M_2, \overline{v_{\overline{z}}}, \overline{v_{\overline{k}}})$

Table 3. Structured Operational Semantics Rules and Axioms.

$$R_l = \left\{\!\!\left|\begin{array}{l} t_{c,1},\ldots,t_{c,n_c}, \\ ?t_{t,1},\ldots,?t_{t,n_t}, \\ \mathbf{ask}(boolexpr) \end{array}\right|\!\!\right\} \xrightarrow{(\overline{y})\leftarrow f(\overline{x})} \left\{\!\!\left|\begin{array}{l} t_{p,1},\ldots,t_{p,n_p}, \\ \mathbf{tsc}(S_1),\ldots,\mathbf{tsc}(S_n) \end{array}\right|\!\!\right\}$$
$$\textbf{where } f((\overline{z})) = ((f_1(\overline{z}),\ldots,f_m(\overline{z})))$$

$$R_i = \left\{\!\!\left|\begin{array}{l} t_{c,1},\ldots,t_{c,n_c}, \\ \uparrow t_{ec,1},\ldots,\uparrow t_{ec,n_{ec}}, \\ ?t_{t,1},\ldots,?t_{t,n_t}, \\ ?\uparrow t_{et,1},\ldots,?\uparrow t_{et,n_{et}}, \\ \mathbf{ask}(boolexpr) \end{array}\right|\!\!\right\} \xrightarrow{(\overline{y})\leftarrow f(\overline{x})} \left\{\!\!\left|\begin{array}{l} t_{p,1},\ldots,t_{p,n_p}, \\ \uparrow t_{ep,1},\ldots,\uparrow t_{ep,n_{ep}}, \\ \mathbf{tsc}(S_1),\ldots,\mathbf{tsc}(S_n) \end{array}\right|\!\!\right\}$$
$$\textbf{where } f((\overline{z})) = ((f_1(\overline{z}),\ldots,f_m(\overline{z})))$$

$$R_{term} = \left\{\!\!\left|\begin{array}{l} ?t_{t,1},\ldots,?t_{t,n_t}, \\ \mathbf{ask}(boolexpr) \end{array}\right|\!\!\right\} \xrightarrow{(\overline{y})\leftarrow f(\overline{x})} \{\!\!|\ \uparrow t_{ep,1},\ldots,\uparrow t_{ep,n_ep},\ |\!\!\}$$
$$\textbf{where } f((\overline{z})) = ((f_1(\overline{z}),\ldots,f_m(\overline{z})))$$

Table 4. Classification of PoliS Rules Macro.

—Local rules consume, test, and produce only local tuples, without involving the parent space. The axiom **RL** shows the transition applied on the space M: if the program tuple ("r_l" : R_l) is in M and if the rule R_l is enabled (condition expressed by the predicate *LocEnabled* specified in Table 5), the space M is updated deleting the tuples that the rule consumes and adding the tuples (and the new spaces) that the rule produces.
Table 4 contains the specification of R_l.

—Interaction rules interact also with the parent space. The specification of the axiom **RI** shown in Table 3 is similar to the one of **RL** just described. Besides updating the space M_1 it updates the space M_2, parent of M_1 as the rule acts on it as well.

—Termination rules, when enabled, cause the termination of the space they are in. These rules have priority over the other rules. Moreover, a termination rule can only test internal tuples and produce external ones; other operations do not make sense since the local space terminates. The axiom T shows how a space terminates and how some tuples are added to the parent space.

The rule macros R_l, R_i, and R_{term} in Table 3 expand as shown in Table 4. The notation $(t_{c,1},\ldots,t_{c,n_c})$ denotes the tuples to be consumed locally, the notation $(?t_{t,1},\ldots,?t_{t,n_t})$ denotes the tuples that are tested locally, and $(t_{p,1},\ldots,t_{p,n_p})$ are the tuples that are produced. $(\mathbf{tsc}(S_1),\ldots,\mathbf{tsc}(S_n))$ denotes the generated subspaces. In the specification of R_i, and R_{term} the notation $(\uparrow t_{ec,1},\ldots,\uparrow t_{ec,n_{ec}})$ denotes the tuples consumed in the parent space, while $(?\uparrow t_{et,1},\ldots,?\uparrow t_{et,n_{et}})$ are the tuples that are only read from the parent space. Finally, the notation $(\uparrow t_{ep,1},\ldots,\uparrow t_{ep,n_{ep}})$ denotes the tuples produced in the parent space. The PoliS construct **ask** checks the values of tuples parameters. The general form of the **ask** predicate is **ask**(*predicate*) and it is another condition to be added to the precondition set.

The predicates *TermEnabled*, *LocEnabled*, and *IntEnabled* used in Table 3 are formally described in Table 5 to check if the rules are enabled. The *TermEnabled* condition is true if the rule R_{term} is enabled, that is, if the program tuple (*term* :

$$\begin{array}{l}
LocEnabled(R_l, \text{“}r_l\text{”}, M, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}}) \triangleq \\
\quad \{\!\!\{\overline{t}_c[\overline{v_{\overline{x}}}/\overline{x}], \overline{t}_t[\overline{v_{\overline{x}}}/\overline{x}]\}\!\!\} \subseteq \{\!\!\{(\text{“}r_l\text{”} : R_l)\}\!\!\} \oplus M \;\wedge\; \overline{v_{\overline{y}}} = f(\overline{v_{\overline{x}}}) \wedge boolexpr[\overline{v_{\overline{x}}}/\overline{x}] \\
\quad \wedge\; \forall R, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}} : ((\mathbf{terminate} : R) \in M \Rightarrow \neg TermEnabled(R, M, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}}) \\[4pt]
IntEnabled(R_i, \text{“}r_i\text{”}, M_1, M_2, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}}) \triangleq \\
\quad \{\!\!\{\overline{t}_c[\overline{v_{\overline{x}}}/\overline{x}], \overline{t}_t[\overline{v_{\overline{x}}}/\overline{x}]\}\!\!\} \subseteq \{\!\!\{(\text{“}r_i\text{”} : R_i)\}\!\!\} \oplus M_1 \\
\quad \wedge\; \{\!\!\{\overline{t}_{ec}[\overline{v_{\overline{x}}}/\overline{x}], \overline{t}_{et}[\overline{v_{\overline{x}}}/\overline{x}]\}\!\!\} \subseteq M_2 \;\wedge\; \overline{v_{\overline{y}}} = f(\overline{v_{\overline{x}}}) \wedge boolexpr[\overline{v_{\overline{x}}}/\overline{x}] \\
\quad \wedge\; \forall R, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}} : ((\mathbf{terminate} : R) \in M_1 \Rightarrow \neg TermEnabled(R, M_1, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}}) \\
\quad \wedge\; \forall R, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}} : ((\mathbf{terminate} : R) \in M_2 \Rightarrow \neg TermEnabled(R, M_2, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}}) \\[4pt]
TermEnabled(R_{term}, M, \overline{v_{\overline{x}}}, \overline{v_{\overline{y}}}) \triangleq \\
\quad \{\!\!\{\overline{t}_t[\overline{v_{\overline{x}}}/\overline{x}]\}\!\!\} \subseteq \{\!\!\{(\mathbf{terminate} : R_{term})\}\!\!\} \oplus M \;\wedge\; \overline{v_{\overline{y}}} = f(\overline{v_{\overline{x}}}) \wedge boolexpr[\overline{v_{\overline{x}}}/\overline{x}]
\end{array}$$

Table 5. Precondition predicates.

R_{term}) is in the same space M as the tuples to be tested. The *LocEnabled* condition is true if the rule R_l is enabled, that is, if the program tuple (“r” : R_l) is in the same space M as the tuples to be consumed and tested. Furthermore, no termination rules (which have priority) should be enabled. The *IntEnabled* condition is true if the interaction rule R_i is enabled, that is, if the program tuple (“r” : R_i) is in the space M_1 and if the tuple it has to test and consume on the local and parent spaces are respectively in M_1 and in the parent of M_1 (i.e., M_2). Furthermore, no termination rules should be enabled in M_1 or M_2.

For sake of brevity we do not describe the semantics for operator $\oplus$ and $\backslash$; they have their intuitive meanings of multiset union and difference, respectively. We are now ready to define a transition system for PoliS.

$$PoliS_{TransitionSystem} = (MS, \longrightarrow_{MS})$$

where the MS syntax is defined in Table 2 and $\longrightarrow_{MS} \subseteq MS \times MS$ is the minimal relation satisfying the rules described above.

The transition system to be associated to a PoliS specification *Spec* is formally defined as a triple
($\uparrow Spec, \longrightarrow_{Spec}, StartContext$) where:

—$StartContext$ is the initial MS (called *initial state*)

—$\uparrow Spec \subseteq MS$ is a minimal subset of MS such that:
—$\dfrac{MS_1 \;\in \uparrow Spec \quad MS_1 \longrightarrow MS_2}{MS_2 \;\in \uparrow Spec}$

—$\longrightarrow_{Spec} \subseteq \uparrow Spec \times \uparrow Spec$ is the restriction of $\longrightarrow$ to $\uparrow Spec$;

—$StartContext \;\in \uparrow Spec$.

The transition system model and the operational semantics have been used for the construction of a model checker for PoliS, that we present in Section 5.

4. POLIS AND MOBILITY

Modern network technologies including mobile computers and devices, and the programming languages for the Internet, like Java, require novel software design techniques. An important feature in network applications is mobility; however, it is still unclear which entities can be mobile and especially why and when they should move over the network. Mobility can range from mobility of data, as in

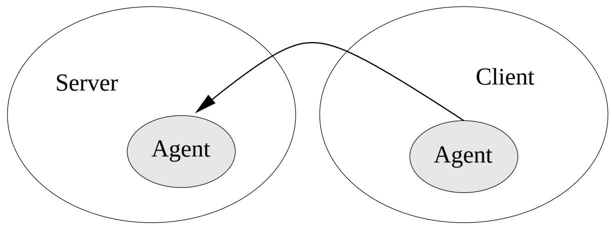

Fig. 4. A simple Client-Server system with a Mobile Agent.

client-server architectures, to mobility of code, as in Java based applications, to mobility of agents, as in some applications for electronic commerce, to mobility of whole operating environments, as in platforms including mobile hardware.

In this section we show how PoliS can be used for the specification of systems containing mobile components, and in the next sections we illustrate how we use our model checker to analyze mobile systems.

The PoliS language allows the specification of both data and code mobility as first class operations. Mobility of data is denoted by rules able to consume tuples locally and to produce tuples outside the local space (or vice versa). Code mobility is denoted by rules able to consume and produce tuples containing code, i.e., other rules. The ability of moving code and data and the creation/destruction operations acting on spaces allow the specification of mobility of complex agents carrying code and data as well.

Agents in this context are represented by spaces containing tuples and rules. Agent mobility is coded by a combination of code and data mobility. In order to show how agent mobility can be expressed in PoliS we modify the example in Table 1 by adding an agent that is sent from the client to the server to perform some computations (Figure 4). The example shows how we specify mobility of data, code, and agents. Table 6 contains the PoliS specification of the system.

$Client$ is the client space. It contains the subspace $Agent$, the tuple (“*get_ready*”), and the program tuple (“*send*” : $SEND$). The data tuple (“*get_ready*”) tells the agent to get ready to be sent. The code of the rule $SEND$ actually sends the agent (once ready), i.e., the tuple $frozen$ and a program tuple (“*agent*” : a), where a is a formal parameter that is matched with a piece of code (in this case the code $AGENT$ when present).

The $Agent$ space is described in the same table. The rule $START$ consumes the data tuple (“*get_ready*”) from the client space (i.e., the parent space) and produces the tuple (“*state*”, “*readytogo*”, 0) into the agent local space enabling the rule $READY$ for execution. The termination rule $READY$ terminates the space saving the status of the agent, i.e., the $frozen$ tuple, and the activation rule (“*agent*” : $AGENT$) in the client space. The predicate **ask** checks if the value of s (contained in the *state* tuple) is either *readytogo* or $afterjob$, i.e., the agent is ready to be sent, or it has finished a job. This enables the client's rule $SEND$, already described. The $Agent$ space also contains the program tuples (“*agent*” : $AGENT$), (“*resume*” : $RESUME$), and (“*job*” : JOB). The first acts as an “unfreeze” for the agent space whenever the agent is “frozen” (i.e. it generates the new space). The rule $RESUME$ reacts when the agent space has been created, getting the

StartContext

$$StartContext = \{\!\!\{ (\text{"}create\text{"} : CREATE) \}\!\!\}$$

$$CREATE = \{\!\!\{ (\text{"}create\text{"} : CREATE) \}\!\!\} \longrightarrow \{\!\!\{ \mathbf{tsc}(Client), \mathbf{tsc}(Server) \}\!\!\}$$

Client

$$Client = \{\!\!\{ Agent, (\text{"}get\_ready\text{"}), (\text{"}send\text{"} : SEND) \}\!\!\}$$

$$SEND = \left\{\!\!\left\{ \begin{array}{l} (\text{"}frozen\text{"}, t), \\ (\text{"}agent\text{"} : a) \end{array} \right\}\!\!\right\} \longrightarrow \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{"}agent\text{"} : a), \\ \uparrow(\text{"}frozen\text{"}, t), (\text{"}wait\text{"}) \end{array} \right\}\!\!\right\}$$

Agent

$$Agent = \left\{\!\!\left\{ \begin{array}{l} (\text{"}start\text{"} : START), (\text{"}resume\text{"} : RESUME), (\text{"}job\text{"} : JOB), \\ (\mathbf{terminate} : READY), (\text{"}agent\text{"} : AGENT) \end{array} \right\}\!\!\right\}$$

$$START = \{\!\!\{ \uparrow(\text{"}get\_ready\text{"}) \}\!\!\} \longrightarrow \{\!\!\{ (\text{"}state\text{"}, \text{"}readytogo\text{"}, 0) \}\!\!\}$$

$$READY = \left\{\!\!\left\{ \begin{array}{l} ?(\text{"}state\text{"}, s, r), \\ ?(\text{"}agent\text{"} : AGENT), \\ \mathbf{ask}(s = \text{"}readytogo\text{"} \ or \\ s = \text{"}afterjob\text{"}) \end{array} \right\}\!\!\right\} \xrightarrow{(t) \leftarrow f(s,r)} \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{"}frozen\text{"}, t), \\ \uparrow(\text{"}agent\text{"} : AGENT) \end{array} \right\}\!\!\right\}$$

where $f(state, result) = (freeze(state, result))$

$$RESUME = \{\!\!\{ \uparrow(\text{"}frozen\text{"}, t) \}\!\!\} \xrightarrow{(s,r) \leftarrow f(t)} \{\!\!\{ (\text{"}state\text{"}, s, r) \}\!\!\}$$

where $f(tuple) = (\text{"}beforejob\text{"}, result(tuple))$

$$JOB = \left\{\!\!\left\{ \begin{array}{l} (\text{"}state\text{"}, s, r), \uparrow(\text{"}data\text{"}, d), \\ \mathbf{ask}(s = \text{"}beforejob\text{"}) \end{array} \right\}\!\!\right\} \xrightarrow{(r', s') \leftarrow f(d,r,s)} \{\!\!\{ (\text{"}state\text{"}, s', r') \}\!\!\}$$

where $f(data, oldresult, \text{"}beforejob\text{"}) = (calculate(data, oldresult), \text{"}afterjob\text{"}))$

$$AGENT = \{\!\!\{ (\text{"}unfreeze\text{"}), (\text{"}agent\text{"} : AGENT) \}\!\!\} \longrightarrow \{\!\!\{ \mathbf{tsc}(Agent) \}\!\!\}$$

Server

$$Server = \{\!\!\{ (\text{"}getag\text{"} : GET), (\text{"}data\text{"}, \text{"}d\text{"}) \}\!\!\}$$

$$GET = \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{"}frozen\text{"}, t), \\ \uparrow(agent : a) \end{array} \right\}\!\!\right\} \longrightarrow \left\{\!\!\left\{ \begin{array}{l} (\text{"}frozen\text{"}, t), \\ (agent : a), (\text{"}unfreeze\text{"}) \end{array} \right\}\!\!\right\}$$

Table 6. Specification of a Client-Server System with a Mobile Agent.

frozen status of the agent and emitting the status tuple in the agent space. The rule JOB denotes the real code for executing a job: it is used when the agent is at the server site and the "data" are available. The **where** clause is abstractly specified as a function f because in the spirit of most coordination languages (which separate computation from coordination) we omit computation details, however it could be refined defining the exact mechanism for the computation of the result. The $Server$ space contains the data that will be used by the agent code JOB to compute a result, and the program tuple ("$getag$" : GET), to gather an agent from

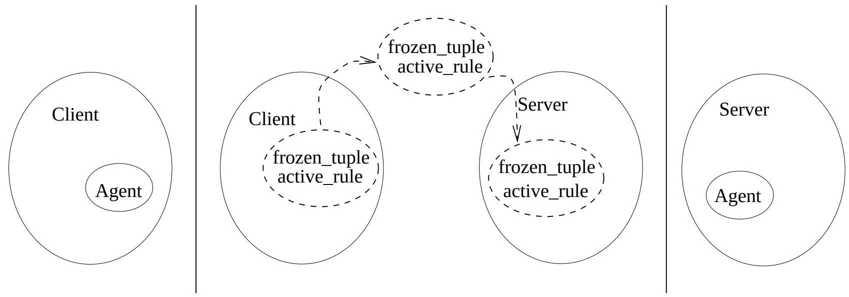

Fig. 5. Agent Mobility in PoliS.

the environment. The rule GET gathers the frozen agent and the activation code (i.e.,(“$agent$” : a), with the formal parameter a matching the real code), and emits the tuple (“$unfreeze$”) so that the agent can unfreeze itself.

As the example shows, code mobility can be modeled in PoliS consuming and producing tuples representing code (i.e., containing rules). For instance, the rule $SEND$ consumes locally and produces in the environment the program tuple containing the activation code for the agent, i.e., (“$agent$” : a), where a is a formal parameter matched with the code $AGENT$ when available. Agent mobility is depicted in Figure 5. An agent is “frozen” and the code for the re-activation of the agent is moved together with its frozen status to another location, where the agent will be reactivated.

This approach to agent mobility has several advantages. PoliS shows clearly that code and state mobility are orthogonal concepts. For instance, we can specify the movement of several agents sharing the same code simply using as many status tuples as agents and a single code tuple. Another example is that we can redefine the behavior of an agent changing its code but keeping its state. Another advantage is related to the performance of the model checker we have implemented for the language (see Section 5): the consideration of space (i.e., agent) mobility as first class in the language on one hand would allow rules to consume and produce spaces as normal tuples. On the other this would lead to an explosion in the number of states to be considered by the tool. Nevertheless, we are exploring the possibility of enhancing the language with space mobility and studying how we can still reason automatically on such a model.

The basic mobility mechanism we have in PoliS is constrained to be “step by step”, that is no general visibility on all the possible locations is considered. Agents can be either “pushed” to known locations, or “pulled” inside a space by the space itself. A tuple of data or code can be moved from one space to the parent, or pulled from the parent to the local space and a complex path composed of these steps can be generated. This abstraction models a general network architecture including layered routers, hierarchical LANs structure, and fire-walls [Cardelli and Gordon 2000]. The ability to dynamic re-arrange the hierarchy of spaces allows a strong control on agent interactions. Moreover, from a model checking perspective the “step by step” mobility mechanism permits a more constrained space explosion than with a general “move to location” mechanism.

In the next section we introduce the model checker that we have built on PoliS and its features.

5. MODEL CHECKING A COORDINATION MODEL

Theorem proving has been the most traditional method of system analysis [Broy 1996]. In theorem proving a deductive system with axioms and derivation rules is usually defined. Starting from the axioms and using the rules it is possible to prove new theorems. Such a method can be applied to software systems as well: if the axiom set is enriched with a formal definition of a software system, then the properties derived from the deductive system are the properties that the system satisfies. In [Ciancarini et al. 1998] a mapping between the PoliS operational semantics and TLA (Temporal Logic of Action) [Lamport 1994] has been studied. This allowed us to use a theorem prover for formal reasoning on PoliS specifications. However, theorem provers require human interaction in order to complete proofs while model checking techniques provide completely automatic verification frameworks. In this paper we exploit a model checking technique to perform analysis on PoliS specification documents. Model checking was initially used for the verification of hardware systems. A landmark paper [Clarke et al. 1986] suggested and studied a model checking approach for software systems. Model checking aims at finding an assignment (*model*) for system variables that satisfies the formulae describing some system properties. Given a model of a software system (derived from its operational specification) a model checker makes an exhaustive analysis of variable values possible in the model. This method may seem trivial and inefficient, but it is very powerful for systems with finite state models.

Model checkers are completely automatic. An important feature of model checking is the ability to find counter-examples (i.e., a path that leads to a scenario where the property is false). Abstract model checking [Clarke et al. 1994] and deductive model checking [Simpa et al. 1996] are the most often exploited techniques to deal with infinite systems. The exponential explosion of the number of system states can also be managed with symbolic model checking and the use of BDD (Binary Decision Diagrams) [Burch et al. 1992].

The model checker we have built exploits PoliS modularity features (i.e., spaces defining context boundaries) in order to reduce the space of the graphs built for a specification. The algorithm applied for the verification of properties follows the one presented in [Clarke et al. 1986]. The logic is based on CTL (Computation Tree Logic) [Clarke et al. 1986]: the differences between our logic and CTL are related to the spaces-based coordination model. We will give the details of the graph construction, the logic and the model checking in the following sections.

5.1 The PoliS Graph Construction

In Section 3 we have described an operational semantics for PoliS. We now consider the transition system defined by the Structural Operational Semantics (SOS); the graph obtained from the unfolding of a transition system of a real system is something quite similar to our model. The main difference between SOS unfolding and our model is that in SOS a unique monolithic graph is built to represent a system, while here we associate a graph to each sub-space definition. The nodes of the graph show how a space evolves; instead, edges are labeled with tuples produced/consumed and tested in the parent spaces. As an example consider the space

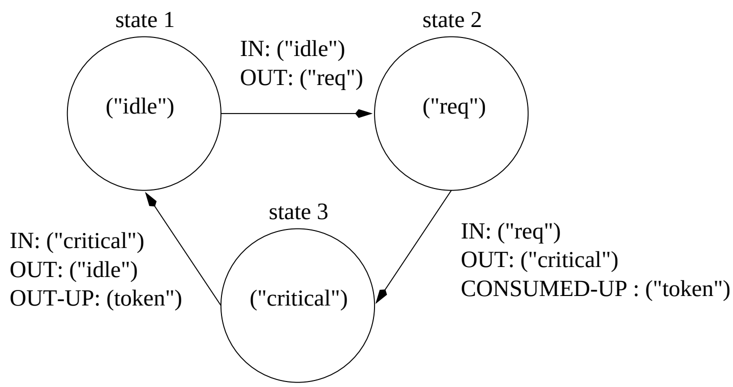

Fig. 6. Graph for the simple space Component.

Component in Table 7 and the graph built for this space in Figure 6. The component can be idle or performing some critical actions (when it obtains the token). It can also return the token (by rule PUT). The three nodes in Figure 6 indicate the possible states for the space *Component*, namely *idle*, *critical*, and *req*. The arrows show the transitions due to the application of the rules REQ, GET, or PUT. The labels on the arrows describe the tuples tested, consumed, and produced in each transition. Our model checker works recursively starting from the more deeply nested spaces, up to the main space.

We distinguish two kinds of spaces: spaces which do not contain other spaces and spaces which contain subspaces. From hereafter we call *simple spaces* the former, and *compound spaces* the latter. The graph for simple spaces is built according to the SOS transition system. In graphs for compound spaces we exploit *configurations*. A configuration is a triple $(graph, instance, state)$ that uniquely identifies a state in a graph of a space: a configuration is a descriptor for a subspace instance. A graph for a compound space contains a configuration for each subspace. In Table 8 we describe the specification for a root space (named $StartContext$) including two instances of space *Component* given in Table 7. The graph is built according to the SOS : a labeled transition for each rule activation is built from the initial state (defined by an initial multiset). The final state represents the multiset with

Component

$$Component = \left\{\!\!\left| \begin{array}{l} (\text{“}idle\text{”}), (\text{“}req\text{” } : REQ), \\ (\text{“}get\text{” } : GET), (\text{“}put\text{” } : PUT) \end{array} \right|\!\!\right\}$$

$$REQ = \{\!|\ (\text{“}idle\text{”})\ |\!\} \longrightarrow \{\!|\ (\text{“}req\text{”})\ |\!\}$$

$$GET = \{\!|\ (\text{“}req\text{”}), \uparrow(\text{“}token\text{”})\ |\!\} \longrightarrow \{\!|\ (\text{“}critical\text{”})\ |\!\}$$

$$PUT = \{\!|\ (\text{“}critical\text{”})\ |\!\} \longrightarrow \{\!|\ (\text{“}idle\text{”}), \uparrow(\text{“}token\text{”})\ |\!\}$$

Table 7. Specification of a simple component.

StartContext

$$StartContext = \{\!\!|\ Component, Component, (\text{“}token\text{”})\ |\!\!\}$$

Table 8. Specification for *StartContext*.

rewritten tuples. The transition label includes tuples to be tested, consumed, or produced in the parent space. When a computation is performed inside a subspace, everything in the state representing the parent space is unchanged, but the configuration of the subspace. In Figure 7 we show how we exploit configurations[1]: the initial state of the graph corresponding to the main space (*StartContext*) contains the tuple *token* and two configurations corresponding to the two instances of the two components $(C, 1, 1)$, $(C, 2, 1)$, (C stands for *Component*) where the second parameter denotes the instance id (i.e., *Component*1 and *Component*2), and the last parameter is a pointer to the state of the graph of the *Component* space (i.e., the state 1 in both cases).

Our model is more useful and powerful than the SOS model mainly for two reasons: first, we save space when there are several instances of some graph definition, as in the previous example; second, we can abstract a single space and analyze its model independently from other spaces. However, building a graph independently from its context introduces some problems. For example, the case in which a formal tuple has to be consumed in the parent space has to be handled. Uninstantiated identifiers can hold any value, so for correctness, while building the graph, all the cases have to be considered (i.e., all values for each domain). We handled this problem making a guess on a fixed range of natural numbers given with the specification of the system to be analyzed.

In the following we introduce the logic we use to reason on these graphs and then, the details of the model checking tool.

5.2 The PoliS Temporal Logic

The PoliS Temporal Logic (PTL) is a CTL [Clarke et al. 1986] dialect. The main differences between PTL and CTL depend on the definition of our model, that is based on spaces (multi-sets). All the formulae are evaluated in a context (a space); moreover, we assume that formulae without an explicit context are evaluated in the *StartContext*. An atomic proposition *atom* is a tuple. *atom* is true in a context C if the tuple it represents belongs to a space C. We have also added classical logic operators and some temporal operators. In Table 9 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 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), $ptf \in \&C$ (at least one C space), or $ptf \in \%C$ (exactly one C space), these because in a specification there can be more than one instance of the same space;

[1]To avoid confusion the transition labels of the figure do not contain the list of the tested, consumed, or produced tuples; instead, we label the edges with the names of the rules applied.

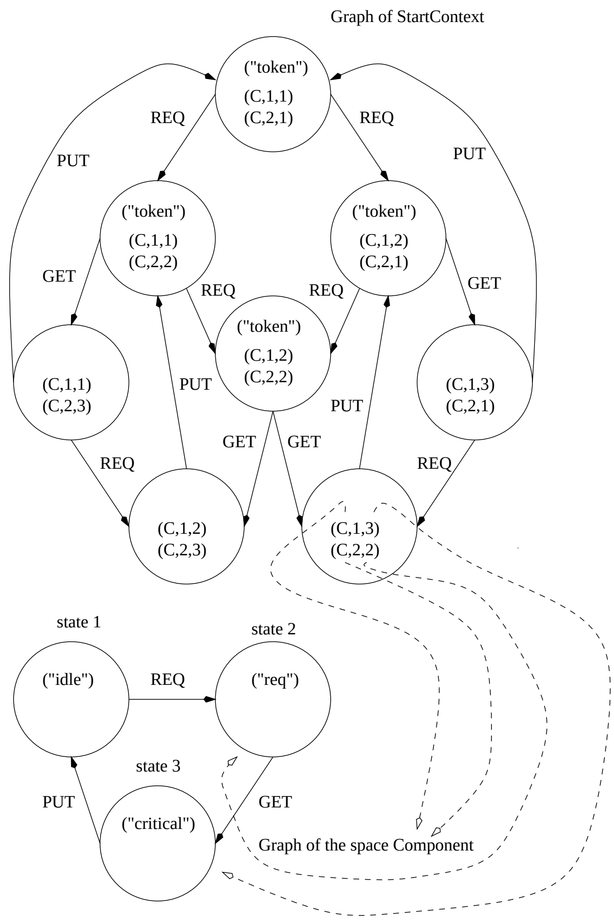

Fig. 7. Representation of configurations.

- 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 $\star$ and &. **X** and **U** are PTL symbols for CTL operators Next and Until;
- $\star\Diamond$*ptf* is defined as $\star$(true**U***ptf*): it means "*for all paths ptf will be eventually true*";
- &$\Diamond$*ptf* is defined as &(true**U***ptf*): it means "*for at least one path ptf will be eventually true*";
- $\star\Box$ptf is defined as $\neg$&$\Diamond\neg$ *ptf* : it means that "*for all paths ptf is always true*";
- &$\Box$*ptf* is defined as $\neg\star\Diamond\neg$ *ptf* : it means that "*for at least a path ptf is always true*";
- *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 all paths ptf' will be eventually true*";
- a *classic* is a PTL formula with classical logic operators;
- an *atom* is simply a tuple.

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

Table 9. PTL Syntax.

5.3 The PoliS Model Checker

We now describe the details of our model checking tool. PoliMC is our 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 as described in Section 5.1; then, it parses the PTL formulae and builds syntactic trees. Finally, it starts the model checking phase.

The model checking algorithm we apply follows the guidelines given by Clarke in [Clarke et al. 1986]. As we have shown in Section 5.2 all formulae can be rewritten using these operators: $\mathbf{X}$,$\mathbf{U}$(preceded by & or ⋆),∧,¬. Thus, the only temporal formulae to verify are of the form:

$$p\wedge\, q,\ \neg\, p,\ \&\mathbf{X}p,\ \star\mathbf{X}p,\ \&(p\mathbf{U}q),\ \star(p\mathbf{U}q)$$

The quantified formulae are handled like macros. A universally quantified formula is expanded in a logic conjunction of its sub-formulae while an existentially quantified formula is substituted by the logic disjunction of its sub-formulae. For instance:

$$\forall\ i \in [0,3]\ (ptf_i)\ \equiv\ (ptf_0\wedge ptf_1\wedge ptf_2\wedge ptf_3)$$

$$\exists\ i \in [0,3]\ (ptf_i)\ \equiv\ (ptf_0\vee ptf_1\vee ptf_2\vee ptf_3)$$

The main difference with respect to the Clarke algorithm is in the handling of *context* formulae. Each sub-formula is checked inside its context. When, during the checking, PoliMC finds a *context* formula like $p \in C$, it leaves the current graph and it starts checking the graph bound to C. This task is performed recursively. When the checking is finished the currently checked state of the parent graph which contains a configuration $(state, graph, instance)$ where *graph* is bound to C and *state* is a state of the graph satisfying p, is labeled with the formula $p \in C$.

The verification of a formula of the form $p \in \{\%, \&, \star\}C$ is similar: a state of the parent of the graph bound to C can be labeled if among all the configurations it contains, which have the *graph* component bound to C, there is respectively only one configuration, some configurations, or all the configurations satisfying the formula p.

The verification of a formula is performed bottom-up: if the length of the formula is n, PoliMC first checks all the sub-formulae of length less than n, then it labels each state according to the labeling of the sub-formulae.

The verification of *atom* formulae is trivial: an *atom* is a tuple and a state will be labeled with this formula if and only if it represents a space which contains the tuple.

The verification of formulae $p \wedge q$, and $\neg q$ depends on the verification of p and q. $\&\mathbf{X}\ p$ and $\star\mathbf{X}\ p$ can be easily checked too: a state s is labeled with this formulae if some or all the states s' in the transitions of type (s, s') are labeled with p.

The verification of formulae that contain the until ($\mathbf{U}$) operator is more complex. The check for $\star(p\ \mathbf{U}\ q)$ is done forward, while the check for $\&(p\ \mathbf{U}\ q)$ is done backward, operating recursively. According to the $\mathbf{U}$ definition a state s can be labeled with $\star(p\ \mathbf{U}\ q)$ if s is labeled with q, or if it is labeled with p and all its successor states are labeled with $\star(p\ \mathbf{U}\ q)$. On the contrary, a state s can be labeled with $\&(p\ \mathbf{U}\ q)$ if it is labeled with q or if it is labeled with p and one of its successor states is labeled with $\&(p\ \mathbf{U}\ q)$. We remark that if a specification contains the creation of new spaces we could obtain infinite graphs, thus we use model checking on a constrained version of the specification where we limit the number of possible generated spaces. This implies that we cannot "prove" properties on the model, as we do not explore all the possible paths. Therefore, we use the tool to test the specifications, to see if formulas are satisfiable, and to find counter-examples.

In the following section we formalize a system with mobile components using PoliS and show how it can be analyzed using the model checker.

6. SPECIFICATION OF AN ARCHITECTURE WITH MOBILE AGENTS

We use PoliS to specify a "Meeting Scheduler System" including mobile agents. This problem was proposed as a case study in mobility for the International Workshop on Software Specification and Design [Feather et al. 1997]. We first give an informal description (Sect. 6.1), then a PoliS specification (Sect. 6.2).

6.1 The Meeting Scheduler System: an Informal Description

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

- a set of dates on which they cannot attend a meeting (exclusion set);
- a set of dates on which they would prefer a meeting to take place (preference set). For simplicity, and without 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.

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.

$$StartContext = \left\{ \begin{array}{|l} Participant \oplus \{\!\!\{ (\text{“}day\text{”}, 1, \text{“}free\text{”}), \ldots, (\text{“}day\text{”}, m, \text{“}exclusion\text{”}) \}\!\!\}, \\ Participant \oplus \{\!\!\{ \ldots \}\!\!\}, \ldots \\ Participant \oplus \{\!\!\{ (\text{“}day\text{”}, 1, \text{“}exclusion\text{”}), \ldots, (\text{“}day\text{”}, m, \text{“}free\text{”}) \}\!\!\}, \\ Agent, \ldots, Agent, \\ (\text{“}start\text{”}, 1), \ldots, (\text{“}start\text{”}, n), \\ (\text{“}end\text{”} : END) \end{array} \right\}$$

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

Table 10. The Meeting Scheduler: the Main Space.

- 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 canceled.
- 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.

Admittedly, this problem can be solved with more conventional technologies: there is no need of mobile agents if we centralize all data in some “meeting server”. The main advantage of using mobile agents is that an agent can exploit reliable links to travel and perform local computations on a site avoiding movement when, for instance, the net is congestioned. We use this case study only to show how PoliS can be used to deal with a solution based on mobile agents.

6.2 A Specification including Mobile Agents

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 preferences and trying to decide a date (see Fig. 8). 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 the space:

Tables 10, 11, and 12 show the specification of three kinds of spaces. The $StartContext$ (Table 10) is the initial space: it includes p participants and n agents, one for each meeting. Each $Participant$ space (Table 11) has an initial state consisting of tuples representing its agenda: some days are marked “free” and others 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 the multi-sets after

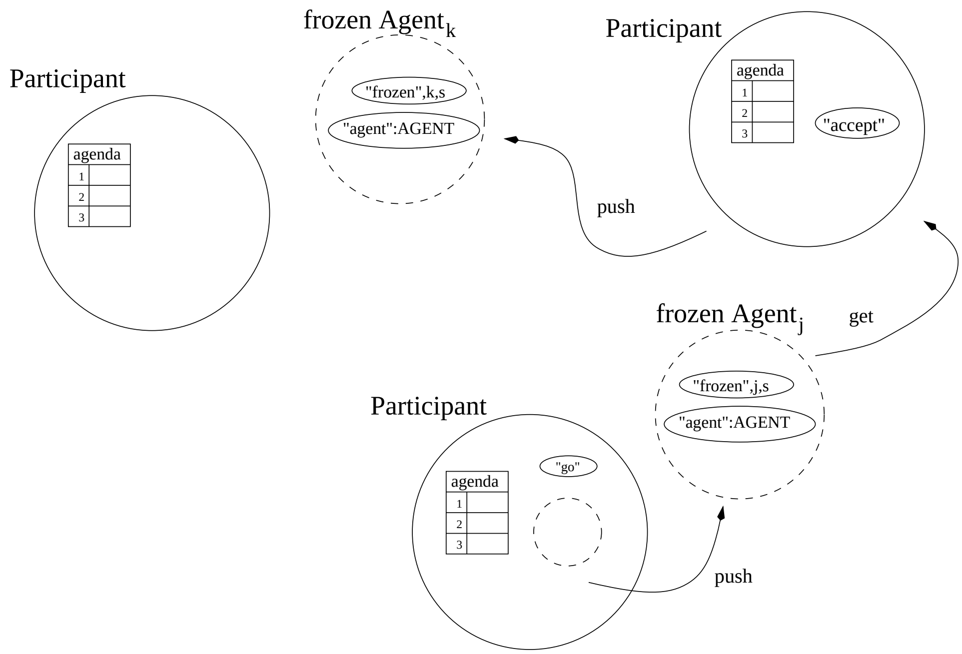

Fig. 8. Agents System Architecture.

the $\oplus$ operator in the *StartContext* definition of Table 10. Tuples (“*start*”, n) are consumed by agents to prepare themselves for the shipping, get a self identifier n and start migrating (see rule $START$ in the *Agent* space).

The *StartContext* space includes just the program tuple (“*end*” : END): the code of 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 given number and outputs a date).

Participant

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

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

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

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

Table 11. The Meeting Scheduler: the Participant Space.

Agent

$$Agent = \left\{\!\!\left|\begin{array}{l} (\text{``}resume\text{''}), (\text{``}start\text{''} : START), (\text{``}resume\text{''} : RESUME), \\ (\text{''}update\text{''} : U), (\mathbf{terminate} : EXIT), (\text{``}withdraw\text{''} : WITHDRAW), \\ (\text{``}agent\text{''} : AGENT) \end{array}\right|\!\!\right\}$$

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

$$AGENT = \left\{\!\!\left|\begin{array}{l} (\text{``}agent\text{''} : AGENT), \\ ?(\text{``}agent\text{''}) \end{array}\right|\!\!\right\} \longrightarrow \{\!|\ \mathbf{tsc}(Agent)\ |\!\}$$

$$RESUME = \left\{\!\!\left|\begin{array}{l} \uparrow(\text{``}frozen\text{''}, i, s), \\ (\text{``}resume\text{''}), \uparrow(\text{``}agent\text{''}) \end{array}\right|\!\!\right\} \xrightarrow{(d_1,\ldots,d_m)\leftarrow f(s)} \left\{\!\!\left|\begin{array}{l} (\text{``}M\text{''}, 1, d_1), \ldots, \\ (\text{``}M\text{''}, m, d_m), \\ (\text{``}self\text{''}, i), \uparrow(\text{``}go\text{''}) \end{array}\right|\!\!\right\}$$

where $f(x) = (unzip_1(x), \ldots, unzip_m(x))$

$$U = \left\{\!\!\left|\begin{array}{l} \uparrow(\text{``}day\text{''}, 1, d_1), \ldots, \\ \uparrow(\text{``}day\text{''}, m, d_m), \\ (\text{``}M\text{''}, 1, v_1), \ldots, \\ (\text{``}M\text{''}, m, v_m), \\ ?(\text{``}self\text{''}, me) \end{array}\right|\!\!\right\} \xrightarrow{(\overline{e},\overline{w})\leftarrow f(\overline{d},\overline{v},me)} \left\{\!\!\left|\begin{array}{l} \uparrow(\text{``}day\text{''}, 1, e_1), \ldots, \\ \uparrow(\text{``}day\text{''}, m, e_m), \\ (\text{``}M\text{''}, 1, w_1), \ldots, \\ (\text{``}M\text{''}, m, w_m), \\ (\text{``}done\text{''}) \end{array}\right|\!\!\right\}$$

where $f(\overline{x}, \overline{y}, z) = (if\ (\forall j x_j \neq z \wedge k = min\{j | x_j = \text{``}free\text{''}\}) then\ (\overline{x}_{|x_k=z}, \overline{y}_{|y_k=y_k+1})$ $else\ (\overline{x}, \overline{y}))$

$$WITHDRAW = \left\{\!\!\left|\begin{array}{l} \uparrow(\text{``}day\text{''}, h, me), \\ (\text{``}M\text{''}, h, n_1), \\ ?(\text{``}self\text{''}, me), \\ \mathbf{ask}(n_1 > 0) \end{array}\right|\!\!\right\} \xrightarrow{(n_2)\leftarrow f(n_1)} \left\{\!\!\left|\begin{array}{l} \uparrow(\text{``}day\text{''}, h, \text{``}free\text{''}), \\ (\text{``}M\text{''}, h, n_2), \\ (\text{``}done\text{''}) \end{array}\right|\!\!\right\}$$

where $f(x) = (x - 1)$

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

where $f(x_1, \ldots, x_m) = (zip(x_1, \ldots, x_m))$

Table 12. The Meeting Scheduler: the Agent Space.

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 $(\text{``}frozen\text{''}, h, s)$ and $(\text{``}agent\text{''} : a)$ from the main space and generates them locally. It also consumes the $(\text{``}accept\text{''})$ tuple locally and generates the tuple $(\text{``}agent\text{''})$, 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 $(\text{``}agent\text{''})$ and $(\text{``}frozen\text{''}, h, s)$ to the main space. Fig. 8 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. This rule simply decides to free a date removing the tuple $(\text{``}day\text{''}, d, \text{``}exclusion\text{''})$ and emitting $(\text{``}day\text{''}, d, \text{``}free\text{''})$.

The *Agent* space (Table 12) contains some rules and a termination rule to make

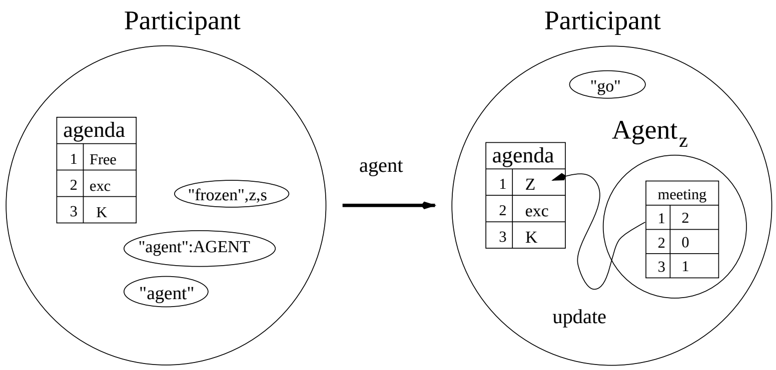

Fig. 9. Agent performing update.

the agent to freeze. The rule $START$ fires an agent to build a calendar (i.e. the tuples (“M”$, d, v$) where d is a day and v is the number of potential attendees for day d, initially all days are free then for all these tuples v is 0. The rule $AGENT$ generates (by **tsc**) a new agent space inside the participant space (Fig. 9). The first rule enabled in a new agent space, inside a participant space, is $RESUME$: this rule is used to get and restore the frozen state of the agent. It emits a tuple (“*go*”) enabling rule $PUSHAG$ for a next move. An agent contains also rule U (Update) and rule $WITHDRAW$. The rule U updates the agenda of a participant using the following policy: a participant takes the first free date, if it exists, and books it; a participant cannot book more than one date. Rule U also updates the internal agent table[2], represented by tuples like (“M”$, d, v$) as explained above. In Fig. 9 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” (that now is 2) in the Agent Table. 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 a *termination* 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 frozen state (“*frozen*”$, i, s$) and a tuple (“*agent*” : $AGENT$) for regenerating an *Agent* space.

The model of mobility of the meeting agents is exactly the one specified in Section 4 and shown in Figure 5. The meeting agent space is frozen when the agent has finished collecting informations in the participant site. Then, some tuples are emitted in the main space and other participants can get the agent tuples and the agent is regenerated inside another participant site. When the meeting agent has finished, a tuple with the decided meeting date is emitted and the agent is destroyed (by the rule END).

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

6.3 Analysis of the Meeting Scheduler System

We have used PoliMC to analyze some liveness properties of this system. We use the model checker on finite versions of the specification. The properties we prove for these versions are satisfiable on the abstract specification but we cannot say that such properties are verified in general. Since some components can move we are interested in studying the dynamic behavior of the system. For instance, we would like to prove that an agent will be able to establish a meeting date, or that some properties on the migration of an agent inside/outside the components are true. 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 the meetings are arranged). $Move$ states that an agent is in a participant site (i.e., an agent space is inside a participant space) and it has performed some actions (i.e., the tuple (“*done*”) is produced). We study a configuration where the number of meetings to be arranged, namely the number of agents, is smaller than the available days, otherwise, trivially, some agents will never find a date. We would like to verify the following:

$$\star\Box\star\Diamond End \tag{1}$$

That is: the *End* property (i.e., each agent finds a date for its meeting) will be valid for all the execution (i.e., all the execution paths lead to a state where the *End* condition is verified). However, PoliMC shows that (1) is false. To understand this we can think of a scenario where agents are not able to agree, choosing the same date and then withdrawing it. Nevertheless, PoliMC also verifies the falsity of:

$$\star\Box\star\Diamond Move \tag{2}$$

Property (2) states that the $Move$ property will be valid (i.e. agents move indefinitely) for all executions. The falsity of (2), verified with the model checker, guarantees that this cannot happen, so we are sure to have a scenario where all meetings are arranged. PoliMC verifies that this property is not true if the number of meetings (agents) is greater than the number of the available days.

As (1) and (2) are proved false we can verify the following formula:

$$\star\Box\star\Diamond(End \vee Move) \tag{3}$$

That is, in all the executions it is true that some agents move or all the meetings are arranged. This shows that the system cannot deadlock. In this example however, properties (1) and (2) above cannot help us to guarantee progress. Therefore, to ensure that all the meetings will be arranged we need a fairness condition in the form:

$$\star\Box\&\Diamond End \Rightarrow (1) \tag{4}$$

that is:

$$\star\Box\&\Diamond End \Rightarrow \star\Box\star\Diamond End \tag{5}$$

Property 5 states that if from all the states of all paths we can find at least one path in which *End* is eventually valid, then *End* will be valid in all the paths. In other words if we are always in a state that allows to arrange all the meetings, then this will eventually happen. PoliMC verifies successfully the hypothesis of Property 5:

$$\star\Box\&\Diamond End \tag{6}$$

hence, Property (6) in conjunction with (5) leads to the verification of (1).

Finally we remark that if we remove some rules used to resolve conflicts (like rule $WITHDRAW$ or rule $EXTEND$), (6) is not verified, that is, there are some states where no path leads to *End*. In other words, sometimes a system can reach a state in which it is impossible to arrange some meetings, and some agents move indefinitely. To avoid these situations the withdrawing or the extension of the free dates by some participants should be considered.

In order to verify properties using the model checker we have instantiated the specification of the Meeting Scheduler System. Here we show a part of the specification instantiated with two possible meeting days (i.e. two agents), and two participants. The model checker accepts as input two files containing respectively the PoliS specification of the system and the formulae to be verified. What follows is a part of the specification file for the Meeting Scheduler:

```
startcontext={
 Participant,Participant,
 Agent,Agent,("start",1),("start",2),
 ("end":END)
}

rule END={
    ("frozen",k,(day1,num1,day2,num2)),
    ask(num1=PART \/ num2=PART)
    }
    [(d)<--f(day1,num1,day2)]-->
    {
     ("end",k,d)
    }
    where f(x,y,z)=(if (y=PART) then (x) else (z));
```

`PART` is a constant defining the number of participants. Notice that the state s of the frozen agent ("*frozen*", k, s) consumed by the rule END (Table 10) has been expanded in order to express the conditions on the **where** clause, `(day1, num1, day2, num2)`: `day1` and `day2` indicate the two possible meeting dates while `num1` and `num2` indicate respectively the number of the participants to the two meetings.

The check (**ask**) on the expanded state of the meeting agent has also been inserted: it checks if one of the two dates has been chosen by both the participants. Here is the specification of property (3):

```
*[]*<> (
(forall agent in [1,2] (
        exists day in [1,2] (
```

```
            ("end",day,agent) ))) \/
    ((("done") in & Agent) in & Participant))
```

The range of the agents and the days is explicitly set (`[1,2]`). At the moment the textual specifications input for the model checker have to be written by the specifier, but an interface tool that translates PoliS-LaTeX specifications into textual ones can easily be designed. As the model checker works on finite instances of the specification, the user has to define the range of the parameters. Furthermore, she has to declare the abstract PoliS functions after the **where** clause (i.e., `where f(x,y,z)=(if (y=PART) then (x) else (z))`.

In order to further constrain the state explosion, we are researching techniques of context constrains for compositional reachability analysis (CRA) [Graf and Steffen 1990; Cheung and Kramer 1996]. As in PoliS the components (namely the spaces) make assumptions on their external environment (namely their parent space) using the rule scope (see Figure 2), this kind of analysis can be applied in order to drastically reduce the number of states of the graph. An other approach that could be followed to further reduce the state explosion is symbolic model checking. In [Enders et al. 1992] a technique for building BDD of parallel processes from basic BDD is exploited. The bottom-up fashion of this approach is similar to our technique of building compound spaces from simple spaces.

7. RELATED WORK

PoliS is not the first formal language used to study systems including mobile entities. However, to our knowledge, our proposal is the first attempt to build an automatic framework to analyze properties on specifications of mobile code based systems.

In terms of formal languages, process algebras have been used for the specification of mobility aspects. The π-calculus [Milner 1999] has been the first language offering features to specify movement across *channels*. However the π-calculus does not support a notion of *location*. Many extensions to the π-calculus have been developed in order to overcome this and other issues. For instance, both the Join calculus [Fournet et al. 1996], and the extension defined in [Amadio 1997] explicitly add a notion of location. Klaim [DeNicola et al. 1998] is a language based on the tuple space coordination model, like PoliS, in which a process algebra is integrated with a type system in order to study security issues.

In general, process algebra based languages focus on the notion of *process*, and do not provide the notion of "environment" of the computation. More sophisticated languages offer a concept of environment that we provide with tuple-spaces.

The Chemical Abstract Machine (CHAM) [Berry and Boudol 1992] has *membranes* that are very similar to our spaces, however the CHAM does not support code mobility as the rules are globally defined outside the "chemical solution" (i.e., the global tuple space).

The ambient calculus [Cardelli and Gordon 2000] provides the notion of *ambient*, which is the mobility unit of the language. *Ambients* are like our spaces, but their mobility is first class in the language, unlike in PoliS. The ambient calculus introduces a concept of "step by step" mobility as well. The complexity due to the first class *ambients* mobility make it quite difficult to reason, in particular with tools, on the specifications.

Mobile UNITY [McCann and Roman 1998] is a state-based language used for the specification of physical and logical mobility. It provides a temporal logic that allows reasoning. However no automatic tools exist supporting Mobile UNITY. In [Picco et al. 1997] Mobile UNITY has been used to formalize some common mobile code paradigms (i.e., Code on Demand, Remote Evaluation, and Mobile Agents). All these paradigms can be also encoded in PoliS, and we are looking in the possibility of reason with our automatic tool on them. Furthermore, in Mobile UNITY the dynamic replication of components is not allowed. Therefore, in the Code on Demand paradigm the used code needs to be sent back to the server to be sent again. In PoliS the dynamic cloning of code is allowed and the Code on Demand paradigm can be formalized more directly.

8. FUTURE WORK AND CONCLUSIONS

Traditional languages, models, and methods used to specify and design applications on a single computer usually lack of abstractions to appreciate and understand the problems raised when the computing platform is a "network computer". Mobility is an obvious example: even if admittedly we could define "mobile code" an application going around on a floppy disk, it is a concept that is especially interesting and complex when a networked programmable infrastructure is available, like an intranet or even the whole Internet.

In this paper we have studied how a coordination language can be used to specify and analyze systems including mobile components. The idea consists of having a coordination language that can express a dynamic topology of components and the mobility of code and data. We analyze PoliS documents with a model checker that can test formal properties of the system being specified. An enhancement of PoliS with first class space mobility is possible [Mascolo 1999]. We are interested in studying the ability of automatic reasoning on such a model using an approach similar to the one we use in the model checker we presented in this paper.

Security issues are important in a mobile code setting. Languages such as Klaim [DeNicola et al. 1998] and the Ambient calculus [Cardelli and Gordon 2000] use "capabilities" on operations, or type systems to face security aspects. We see a possible development of PoliS in this direction, and it would be interesting to study which kind of automatic analysis we can perform on security properties.

In term of language interface improvements we are studying a visual notation for PoliS in order to simplify the impact on the users. We are developing an XML-based abstract syntax [Bray et al. 1998] in order to make PoliS specifications more portable and possibly to be able to be integrated with other XML-based frameworks, like the UML notation [Booch et al. 1999].

ACKNOWLEDGMENTS

We would like to thank W. Emmerich, A. Finkelstein and the anonymous reviewers for their careful reading and valuable comments.

REFERENCES

Amadio, R. 1997. An Asynchronous Model of Locality, Failure, and Process Mobility. In *Proc. of the 2nd Int. Conf. on Coordination Models and Languages (COORDINATION '97)*, Volume 1282 of *LNCS* (1997). Springer.

Bagrodia, R., Chu, W., Kleinrock, L., and Popek, G. 1995. Vision, Issues, and Architecture for Nomadic Computing. *IEEE Personal Communications 2*, 6 (December), 14–27.

Banâtre, J. P. and LeMétayer, D. 1996. Gamma and the Chemical Reaction Model: Ten Years After. In J. Andreoli, C. Hankin, and D. LeMétayer Eds., *Coordination Programming: Mechanisms, Models and Semantics*, pp. 3–41. Imperial College Press.

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

Booch, G., Jacobson, I., and Rumbaugh, J. 1999. *The Unified Modeling Language User Guide*. Addison Wesley.

Bray, T., Paoli, J., and Sperberg-McQueen, C. M. 1998. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210 (March), World Wide Web Consortium.

Broy, M. 1996. Experiences with Software Specification and Verification using LP, The Larch Prover Assistant. *Formal Methods in System Design 8*, 3, 221–272.

Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation 98*, 2 (June), 142–170.

Cardelli, L. and Gordon, A. 2000. Mobile Ambients. *Theoretical Computer Science 240*, 1, To appear.

Chess, D. et al. 1995. Itinerant Agents for Mobile Computing. *IEEE Personal Communications 2*, 5 (October), 34–49.

Cheung, S. and Kramer, J. 1996. Context Constraints for Compositional Reachability Analysis. *ACM Trans. on Software Engineering and Methodology 5*, 4 (October), 334–377.

Ciancarini, P. 1996. Coordination Models and Languages as Software Integrators. *ACM Computer Surveys 28*, 2, 300–302.

Ciancarini, P., Mazza, M., and Pazzaglia, L. 1998. A Logic for a Coordination Model with Multiple Spaces. *Science of Computer Programming 31*, 2/3 (July), 231–262.

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

Clarke, E., Grumberg, D., and Long, D. 1994. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems 16*, 5, 1512–1542.

DeNicola, R., Ferrari, G., and Pugliese, R. 1998. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering 24*, 5, 315–330.

Enders, R., Filkorn, T., and Taubner, D. 1992. Generating BDDs for symbolic model checking in CCS. In *Proceedings of Computer Aided Verification (CAV '91)*, Volume 575 of *LNCS* (Berlin, Germany, July 1992), pp. 203–213. Springer.

Feather, M., Fickas, S., Finkelstein, A., and vanLamsweerde, A. 1997. Requirements and Specification Exemplars. *Automated Software Engineering 4*, 4, 419–438.

Fournet, C., Gonthier, G., Levy, J., Maranget, L., and Remy, D. 1996. A Calculus of Mobile Agents. In *Proc. 7^{th} Int. Conf. on Concurrency Theory (CONCUR '96)*, Volume 1119 of *LNCS* (1996). Springer.

Graf, S. and Steffen, B. 1990. Compositional minimization of finite state systems. In *Proc. 2^{nd} Int. Conf. on Computer-Aided Verification (CAV'90)* (New York, 1990). ACM Press.

Kiniry, J. and Zimmerman, D. 1997. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing 1*, 4, 21–33.

Lamport, L. 1994. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May), 872–923.

Lange, D. B. and Oshima, M. 1998. *Programming and Deploying Java^{TM} Mobile Agents with Aglets^{TM}*. Addison-Wesley.

Mascolo, C. 1999. MobiS: a Specification Language for Mobile Systems. In *Proc. 3^{rd} Int. Conf. on Coordination Languages and Models (COORDINATION '99)*, Volume 1594 of *LNCS* (April 1999), pp. 37–52. Springer.

McCann, P. and Roman, G.-C. 1998. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering 24*, 2, 97–110.

Milner, R. 1999. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press.

Picco, G., Roman, G.-C., and McCann, P. 1997. Expressing Code Mobility in Mobile Unity. In *Proc. 6th European Software Eng. Conf. (ESEC/FSE'97)*, Volume 1301 of *LNCS* (1997), pp. 500–518. Springer.

Ramjee, R., LaPorta, T., and Veeraragahvan, M. 1995. The Use of Network-Based Migrating Agents for Personal Communication Services. *IEEE Personal Communications 2*, 6 (December), 62–68.

Serugendo, G. D., Muhugusa, M., and Tschudin, C. 1998. An Survey of Theories for Mobile Agents. *World Wide Web 1*, 3, 139–153.

Simpa, H., Uribe, T., and Manna, Z. 1996. Deductive Model Checking. In *8th Int. Conf. on Computer-Aided Verification (CAV'96)*, Volume 1102 of *LNCS* (1996). Springer.

Thorn, T. 1997. Programming Languages for Mobile Code. *ACM Computer Surveys 29*, 3, 213–239.