A logic for a coordination model with multiple spaces

P. Ciancarini and M. Mazza and L. Pazzaglia
Department of Computer Science
University of Bologna - Italy

Abstract

This paper introduces and studies PoliS, a coordination model to specify the software architecture of distributed applications. PoliS is based on multiple dataspaces containing both data and programs. We define PoliS syntax and semantics, and show how it can be used as a formal notation for specifying open systems. We adopt TLA logic to reason on PoliS specifications. Finally, we discuss an application field for PoliS, namely we use it to specify and reason on software architectures of some simple distributed systems.

1 Introduction and Motivations

Designing large distributed software systems is a difficult software engineering problem. Such a problem is even more difficult if the system being designed has to be *open*, namely it includes software entities encapsulated and reactive usually called objects or agents [27], which are interoperable, i.e. that can dynamically join and leave the system itself. In practice, open systems are built up of several heterogeneous hardware and software components, often already existing before the design of a new system begins (legacy systems). Any solution to the problem of open systems design should provide a method to integrate autonomous components and must take into account heterogeneity both at architectural level (machines, networks, and operating systems [14]) and at linguistic level (languages used to build components [24]). Moreover, it is necessary to model and support the dynamicity of open systems typically due to the fact that components can be added without limitation and without interruption of offered services.

An important issue in open system design concerns *coordinating* active entities [1, 5]. A powerful approach to describe and control coordination and interaction among active entities is founded on the notion of *generative communication* [17, 2]. Generative communication is based on the notion of *shared dataspace* or *tuple space*, in which entities can be generated and later retrieved. These entities can be either passive or active: actually a dataspace can be seen as a *chemical solution* implicitly computing by multiset rewriting [3, 4].

Most researches on coordination models and languages are currently focussed on models based on single or multiple dataspaces. In this paper we

are interested in how to correctly design open systems whose architecture is modeled by multiple dataspaces. In particular we discuss a formal method for construction and verification of these systems: we develop a theoretical coordination model called PoliS. We formally introduce its syntax and semantics. We illustrate how it can be used to specify and reason on open systems. We provide a translation of PoliS specifications into Lamport's TLA and show how we use a theorem prover, namely the TLP prover, to verify PoliS formal documents [13].

The structure of this paper is the following: Sect.2 gives an informal description of the PoliS coordination model; in Sect.3 we formally specify PoliS; in Sect.4 we show how PoliS can be used to specify some coordination applications; in Sect.5 we introduce TLA, and then we develop a TLA semantics for reasoning on PoliS specifications, also shortly describing a verification tool called TLP; in Sect.6 we study a major example of distributed system specified with PoliS.

2 PoliS: an informal description

PoliS is a coordination model based on multiple tuple spaces [18, 8]. A tuple space, or *space* for short, includes both tuples and other spaces. In this way PoliS specifications are hierarchically structured: a PoliS specification 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*, that are ordered sequences of values, and *program tuples*, that contain the coordination rules which manage activities inside the space they belong to. The execution of a program tuple can modify a space tree removing tuples and adding tuples and spaces. However, a program tuple can only handle the tuples of the space it belongs to *and* the tuples of its parent space. This constraint localizes both the "input" and the "output" environment of any agent, as represented by a program tuple.

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

A space is a multiset of tuples. A space is modified by chemical reactions that transform multisets of tuples in multisets of tuples (this is multiset rewriting, and is common to most coordination models based on generative communication, see for instance [3]). The mechanism that defines which reactions can take place is the *rule*. 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's 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 multiset 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 multiset of

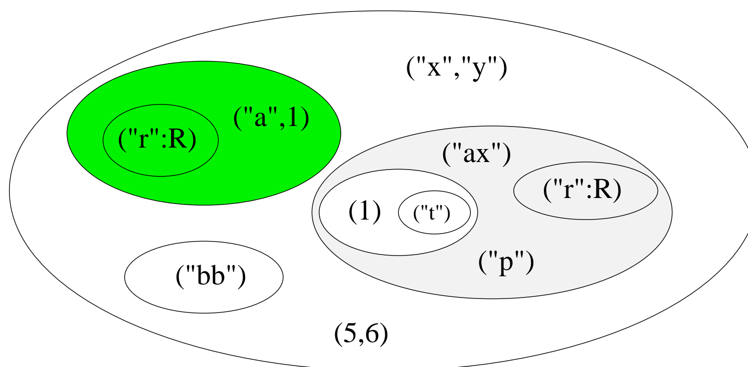

Figure 1: A PoliS space tree

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 allows 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 multiset of tuples matching the preactivation of PT. A *match* relation checks if a multiset of formal tuples M_{ft} can be instantiated by a multiset M_{nft} of ground tuples. Consequently, such a *match* relation is defined between pairs of multisets of tuples and not between pairs of tuples: any identifier appearing in the tuples of the preactivation must be univocally instantiated.

The tuples of the preactivation must be read or consumed in the rule's 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.

In other words, a program tuple is a multiset rewriting rule: preactivation and postactivation are multisets 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 $\uparrow$ 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 $\uparrow$ 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

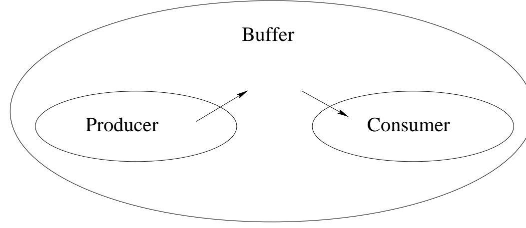

Figure 2: Producer-consumer: spaces topology

$$R_c = \left\{ \begin{array}{l} (\text{“}\mathsf{next}_c\text{”}, i_1), \\ \uparrow(\text{“}\mathsf{prod}\text{”}, i_1, p) \end{array} \right\} \xrightarrow{(i_2)\leftarrow f(i_1)} \left\{ \begin{array}{l} (\text{“}\mathsf{prod}\text{”}, i_1, p), \\ (\text{“}\mathsf{next}_c\text{”}, i_2) \end{array} \right\}$$
$$\textbf{where } f(x) = (x+1)$$

Table 1: Rule R_c

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 multisets 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.

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 another space containing also the buffer represented by tuples generated by the producer. Such a system is graphically shown in figure 2.

Table 1 shows a rule that defines how a consumer gets an item from the buffer. If a tuple of the form $(\text{“}\mathsf{next}_c\text{”}, index)$ is found locally in the consumer space, and the tuple $(\text{“}\mathsf{prod}\text{”}, index, p)$ is found in the parent space, then both tuples are deleted, and two new tuples appear in the consumer 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 **tsc**(M) operation in its postactivation causes the multiset M to be added as a child space of the space where the rule was executed.

For instance, in order to create a space tree representing the producer-consumer system, we can use the rule R_g of table 2. Such a rule creates the spaces S_p and S_c that respectively contain the tuples describing the producer and the consumer.

In order to partially constrain activities inside a tuple space we can define one or more *invariants*, namely constraints that must hold for all

$$R_g = \{\!\!\{(\text{“}\mathsf{r}_g\text{”} : R_g)\}\!\!\} \longrightarrow \{\!\!\{\mathbf{tsc}(S_p), \mathbf{tsc}(S_c)\}\!\!\}$$

$$S_p = \{\!\!\{(\text{“}\mathsf{next}_p\text{”}, 0), (\text{“}\mathsf{r}_p\text{”} : R_p)\}\!\!\}$$

$$S_c = \{\!\!\{(\text{“}\mathsf{next}_c\text{”}, 0), (\text{“}\mathsf{r}_c\text{”} : R_c), (\mathbf{invariant} : R_{inv})\}\!\!\}$$

Table 2: Rule R_g

$$R_{inv} = \{\!\!\{?(\text{“}\mathsf{prod}\text{”}, i, 0)\}\!\!\} \longrightarrow \{\!\!\{\uparrow(\text{“}\mathsf{done}\text{”})\}\!\!\}$$

Table 3: Rule R_{inv}

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 multiset 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 3 in the consumer space. The invariant fires when the consumer space contains a tuple (“prod”, $i, 0$). The result of the activation of the invariant in the consumer space is graphically shown in figure 3; tuple (“done”) represents a termination signal sent by the consumer to the parent space.

A PoliS rule can be seen both as a resource transformer and as an agent that tests and modifies the shared dataspace, performs a computation and then communicates results or requests to the other agents.

A typical way to extend tuple space models is to replace a monolithic tuple space with a multiplicity of spaces. This follows from the intuition that multiple spaces support modularity of activities and allow information hiding of both computations and data resources in order to improve security.

Interspace communication was defined in PoliS avoiding names for spaces (localities). The way PoliS spaces communicate is a simple extension of generative communication. This allows to think of a PoliS system both as an *ensemble of computation loci* inside of which there are agents (the rules) that coordinate via the tuples of the space, and as an *ensemble of agents* (the spaces) in which the siblings agents-spaces coordinate via the common coordination environment represented by the parent space. Consequently, every space is at the same time both a set of agents coordinating through

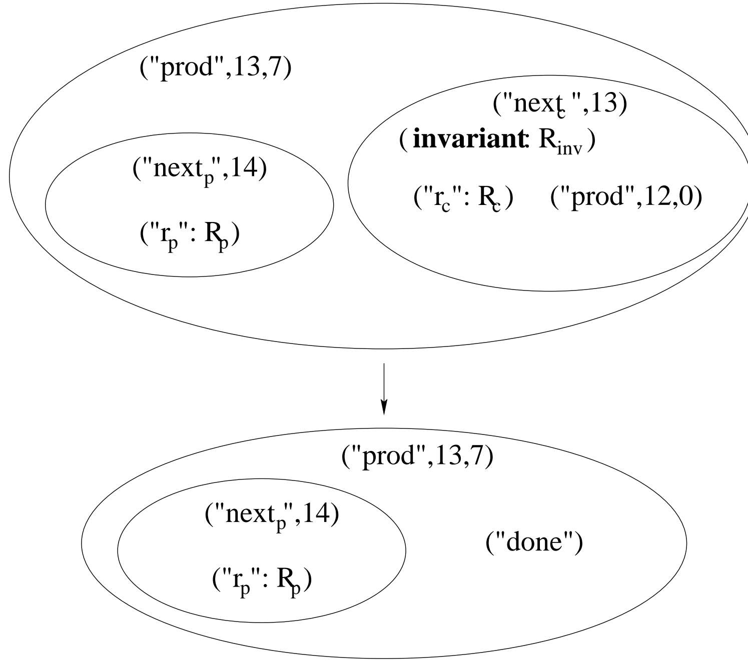

Figure 3: Rule R_{inv} execution

a shared data space, and an agent itself that uses a shared data space to coordinate with other agents.

3 Formal Definition

We give now a formal specification of the PoliS coordination model. A PoliS specification is a pair $Spec = (StartContext, Rules)$ where $StartContext$ is the starting multiset and *Rules* is a set of rules that determine the way the spaces can evolve.

A PoliS specification is mainly operational, however it has also some declarative features. In fact, rules offer an axiomatic method to show the way a coordination application evolves, since rules can be thought of as relations between the pre and the poststatus of a portion of a space.

Systems will be described focusing on modelling interactions among activities in order to point out that PoliS is a specification language tailored to formally characterize coordination.

3.1 Operational Semantics

In the following we present the formal description of PoliS semantics using the operational model based on states and transitions. We present the definition of the relations among states using Plotkin's Structured Operational Semantics. PoliS allowed computations are described through a transition system given as a pair $(S, \longrightarrow)$ where:

- S is the states set

$Space = \mathcal{M}(Tuple \bigcup Space)$
$Tuple = RuleTuple | PassiveTuple$
$PassiveTuple = \{(t_1, \ldots, t_n) | t_i \in V \bigcup RuleId\}$
$RuleTuple = \{(name : rule) | name \in RuleId \wedge rule \in Rule\}$
$Rule =$ PoliS rules set
$V =$ values set
$RuleId =$ rule identifiers set

Table 4: PoliS transition system states set

- $\longrightarrow \subseteq S \times S$ is the transitions set.

PoliS transition system states are multisets trees since the PoliS specification execution describes the space tree evolution caused by rules activation.

PoliS transition system states are the elements of the set named *Space* shown in table 4. Notation $M = \mathcal{M}(Set)$ means that M is the set of multisets built from *Set* elements. *Space* elements are all possible PoliS spaces whose elements are tuples and multisets; *Tuple* elements are PoliS rules and ordered values sequences. The space tree topology is implicitly described by inserting multiset B in space A whenever A is B's parent space.

The transition relation describing changes of state is given through the axioms and inference rules shown in table 5. Rules R_l, R_i and R_{inv} of table 5 are shown in table 6. Rule R_l represents rules not communicating with the parent space, rule R_i represents rules communicating with the parent space, rule R_{inv} represents invariant rules.

Predicates *LocEnabled*, *IntEnabled* and *InvEnabled* of table 5 are shown in table 7. In tables 5, 6 and 7 we use the following abbreviations:

- $\overline{id}$ is the vector of rule identifiers taken according to the order they appear in the rule;
- $\overline{v}$ is a values vector with the same cardinality of $\overline{id}$;
- $\overline{x}$ and $\overline{y}$ are function f input and output identifiers vectors;
- $\overline{v}_{\overline{x}}$ is the vector of $\overline{v}$ elements taken according to $\overline{x}$ identifiers in $\overline{id}$;
- $\overline{v}_{\overline{y}}$ is the vector of $\overline{v}$ elements taken according to $\overline{y}$ identifiers in $\overline{id}$;
- notation $\overline{t}[\overline{v}/\overline{id}]$ means that $\overline{id}$ identifiers must be substituted by the values of $\overline{v}$ in every tuple of vector $\overline{t}$;
- $\subseteq$ is the multiset inclusion operator
- $\oplus$ is the multiset union operator.

The transition system describing PoliS allowed computations is the pair

$$PoliSTransitionSystem = (Space, \longrightarrow)$$

where:

$\forall M, M', M_1, M'_1, M_2 \in \mathit{Multisets}$ and $\forall \overline{v} \in \mathit{ValuesSequences}$

Local rule

$$\{\!|(\text{“}\mathsf{r}_l\text{”} : R_l)|\!\} \oplus M \longrightarrow$$
$$((\{\!|(\text{“}\mathsf{r}_l\text{”} : R_l)|\!\} \oplus M) \setminus \{\!|\overline{t}_c[\overline{v}/\overline{id}]|\!\}) \oplus \{\!|\overline{t}_p[\overline{v}/\overline{id}], \overline{S}[\overline{v}/\overline{id}]|\!\}$$

if $\mathit{LocEnabled}(R_l, M, \overline{v})$

Interaction rule

$$\{\!|\{\!|(\text{“}\mathsf{r}_i\text{”} : R_i)|\!\} \oplus M_1|\!\} \oplus M_2 \longrightarrow$$
$$\{\!|((\{\!|(\text{“}\mathsf{r}_i\text{”} : R_i)|\!\} \oplus M_1) \setminus \{\!|\overline{t}_c[\overline{v}/\overline{id}]|\!\}) \oplus \{\!|\overline{t}_p[\overline{v}/\overline{id}], \overline{S}[\overline{v}/\overline{id}]|\!\}|\!\}$$
$$\oplus (M_2 \setminus \{\!|\overline{t}_{ec}[\overline{v}/\overline{id}]|\!\}) \oplus \{\!|\overline{t}_{ep}[\overline{v}/\overline{id}]|\!\}$$

if $\mathit{IntEnabled}(R_i, M_1, M_2, \overline{v})$

Invariant rule

$$\{\!|\{\!|(\mathbf{invariant} : R_{inv})|\!\} \oplus M_1|\!\} \oplus M_2 \longrightarrow M_2 \oplus \{\!|\overline{t}_{ep}[\overline{v}/\overline{id}]|\!\}$$

if $\mathit{InvEnabled}(R_{inv}, M_1, \overline{v})$

Local transition

$$\frac{M_1 \longrightarrow M'_1}{M_1 \oplus M_2 \longrightarrow M'_1 \oplus M_2}$$

if $\forall R, \overline{v} : ((\mathbf{invariant} : R) \in M \Rightarrow \neg \mathit{InvEnabled}(R, M_1 \oplus M_2, \overline{v}))$

Subspaces transition

$$\frac{M \longrightarrow M'}{\{\!|M|\!\} \longrightarrow \{\!|M'|\!\}}$$

Table 5: PoliS operational semantics

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

where $f(\overline{v}) = (f_1(\overline{v}), \ldots, f_m(\overline{v}))$

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

where $f(\overline{v}) = (f_1(\overline{v}), \ldots, f_m(\overline{v}))$

$$R_{inv} = \{\!|?t_{t,1}, \ldots, ?t_{t,n_t}, \mathbf{ask}(\mathit{expr})|\!\} \xrightarrow{(\overline{y}) \leftarrow f(\overline{x})} \{\!| \uparrow t_{ep,1}, \ldots, \uparrow t_{ep,n_{ep}}|\!\}$$

where $f(\overline{v}) = (f_1(\overline{v}), \ldots, f_m(\overline{v}))$

Table 6: PoliS rules categories

$$
\begin{aligned}
LocEnabled(R_l, M, \overline{v}) \triangleq\ & \wedge\ \{\!|\overline{t}_c[\overline{v}/\overline{id}], \overline{t}_t[\overline{v}/\overline{id}]|\!\} \subseteq \{\!|(\text{“r}_l\text{”} : R_l)|\!\} \oplus M \\
& \wedge\ \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \wedge expr[\overline{v}/\overline{id}] \\
& \wedge\ \forall\, R, \overline{v} : ((\textbf{invariant} : R) \in M \Rightarrow \\
& \qquad \neg InvEnabled(R, M, \overline{v}))
\end{aligned}
$$

$$
\begin{aligned}
IntEnabled(R_i, M_1, M_2, \overline{v}) \triangleq\ & \wedge\ \{\!|\overline{t}_c[\overline{v}/\overline{id}], \overline{t}_t[\overline{v}/\overline{id}]|\!\} \subseteq \{\!|(\text{“r}_i\text{”} : R_i)|\!\} \oplus M_1 \\
& \wedge\ \{\!|\overline{t}_{ec}[\overline{v}/\overline{id}], \overline{t}_{et}[\overline{v}/\overline{id}]|\!\} \subseteq M_2 \\
& \wedge\ \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \wedge expr[\overline{v}/\overline{id}] \\
& \wedge\ \forall\, R, \overline{v} : ((\textbf{invariant} : R) \in M_1 \Rightarrow \\
& \qquad \neg InvEnabled(R, M_1, \overline{v}))
\end{aligned}
$$

$$
\begin{aligned}
InvEnabled(R_{inv}, M, \overline{v}) \triangleq\ & \wedge\ \{\!|\overline{t}_t[\overline{v}/\overline{id}]|\!\} \subseteq \{\!|(\textbf{invariant} : R_{inv})|\!\} \oplus M \\
& \wedge\ \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \wedge expr[\overline{v}/\overline{id}]
\end{aligned}
$$

Table 7: SOS predicates

- *Space* is the set given in table 4
- $\longrightarrow \subseteq Space \times Space$ is the least relation satisfying the axioms and inference rules of table 5.

The operational semantics of a PoliS specification *Spec* is a transition system $(S, \longrightarrow, i)$ where:

- S is the states set
- $\longrightarrow \subseteq S \times S$ is the transitions set
- i is the initial state.

The semantics of $Spec = (StartContext, Rules)$ is defined as follows:

$$[\![Spec]\!]_{Op} = (\uparrow Spec, \longrightarrow_{Spec}, StartContext)$$

where

- $\uparrow Spec \subseteq Space$ is the least set such that
  - $StartContext \in \uparrow Spec$
  - $\dfrac{S_1 \in \uparrow Spec \quad S_1 \longrightarrow S_2}{S_2 \in \uparrow Spec}$
- $\longrightarrow_{Spec} \subseteq \longrightarrow$ is the restriction of relation $\longrightarrow$ to set $\uparrow Spec$.

4 Modelling Coordination with PoliS

A coordination model introduced as a tool to describe open systems must support the dynamicity characterizing open systems.

Open systems dynamicity needs a different approach in describing communication among agents. In distributed systems we usually assume that

an agent supplying a service will keep supplying it also in the future. To invoke the agent services we need to explicitly know that agent. Establishing a link between the agent supplying the service and an agent asking for that service can be an efficient way to exchange queries and answers; moreover the demanding agent can memorize the supplier agent identifier in order to use it again later.

This approach is unsatisfactory in open systems: establishing a connection between agents conditions their behaviour, preventing them from leaving the system as long as the connection lasts; storing an address does not guarantee that the address will be correctly reusable because it is not possible to ensure that the agent owning that address will still be present in the future; the need to know the agent able to supply a service demands an always updated knowledge of the system state.

PoliS allows to model communication between agents in a different way because it makes the system structure transparent thus allowing every agent not to take care of the fact that agents leave or join the system: tuple spaces and uncoupled communication support a communication that frees an agent from explicitly knowing the entity the agent is communicating with and that does not need addresses nor communication channels.

The traditional message sending scheme relying on system addresses to identify the specific recipient of a message does not apply in open systems since the dynamic reconfigurability of such systems implies that agents may change their roles with respect to each other; hence the desirability of making agents communicate on the basis of their properties rather than of their name. Such a communication is named *property-driven*: an agent accepts a message if the message properties match the agents properties. Agents are allowed to send their requests without specifying an address, but simply requiring whoever is in charge to process them. PoliS nameless spaces and uncoupled communication support and promote this kind of communication.

PoliS ability of describing open systems comes from its implicit dynamicity: the possibility of creating and removing spaces allows to formally describe the behaviour of systems in which new localities and components are added or removed during a system's lifetime.

Program tuples can be added or removed from a space and can be delivered through spaces enabling an intuitive description of mobile code. The presence of a set of rules in a space defines a set of space features; the possibility to modify such a set allows to dynamically modify the service capabilities of the entities that make up an open system. The possibility to transfer rules from a space to another allows to model systems in which nodes supply services not only by communicating data or performing remote computations, but also by transferring computation abilities to other nodes.

PoliS effectiveness in modelling open systems can be further stressed and summarized by schematically analyzing relations between PoliS distinguishing features and open systems distinguishing features:

- multiple tuple spaces allow to embody the abstraction of different computation loci able to communicate in a predefined fashion

- the possibility to use any language for local computations (without the need of choosing a unique language for all rules) allows to describe the heterogeneity of languages of open systems components
- communication based on tuples manipulation allows to think of services autonomously with respect to processes able to supply such services
- invariants and primitives to create new spaces are an elegant means to describe eventual system topology changes
- migrating rules give high dynamicity to the system and have the expressive power to describe the behaviour of objects such as Java applets
- the existence of a communication protocol among tuple spaces allows to describe interactions among locally defined independent subsystems.

PoliS inherits all the benefits of generative communication, of chemical metaphor and of multiple tuple spaces, defining an integrated model able to intuitively and effectively describe open systems.

4.1 An Example: Modelling Web Server Replication with PoliS

Server replication is usually employed to increase performance and ensure service even when a machine where a server resides is down. We now show a PoliS specification that models server replication.

To model an architecture with dynamic servers and clients (browsers in Web terms) we associate a space to every server and every browser. We suppose that there are two server machines, that we call Macbeth and Leporello (these are the actual names used in our department). Documents and requests from clients are represented as tuples: a browser submits a request through a tuple containing an url address; a server gets these tuples and interprets them as requests of documents under its control.

Figure 4 depicts graphically an instance of such an architecture: the browsers and the space containing the two servers are subspaces of a common parent space that is the coordination environment. A browser puts its request tuples in the parent space and the space containing the two servers gets request tuples from the parent space. When a server is able to satisfy a request tuple, it consumes it and gives back tuples representing the requested document possibly together with a Java applet. PoliS models a Java applet by a program tuple containing a rule that is executed in the browser after being downloaded from the root space.

Our PoliS specification is shown in tables 8,...,11. A specification is a set of modules; each module contains a PoliS space and a set of rule definitions. In particular, a module defines all rules that appear in the program tuples belonging to the space defined in the module.

Table 8 shows the root space containing the rule that creates the space representing the server group and the rule that creates browsers.

Table 9 shows the space representing the server group: it contains the rules to restart the servers if they are down, the rule to get request tuples

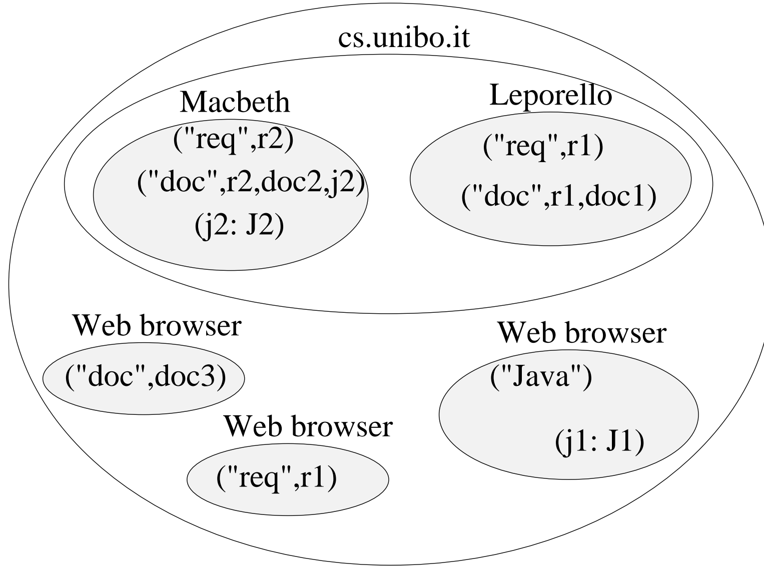

Figure 4: Describing a Web topology (Macbeth and Leporello are two server machines)

StartContext

$$StartContext = \{\!|(\text{“}\mathsf{r}_{cs}\text{”} : R_{cs}), (\text{“}\mathsf{r}_{cc}\text{”} : R_{cc})|\!\}$$

$$R_{cs} = \{\!|(\text{“}\mathsf{r}_{cs}\text{”} : R_{cs})|\!\} \longrightarrow \{\!|\mathbf{tsc}(S_{unibo})|\!\}$$

$$R_{cc} = \{\!||\!\} \longrightarrow \{\!|\mathbf{tsc}(S_c)|\!\}$$

Table 8: WWW: the *StartContext*

$$S_{unibo}$$

$$S_{unibo} = \left\{ \begin{array}{l} (\text{"down leporello"}), (\text{"down macbeth"}), \\ (\text{"r}_{get}\text{"} : R_{get}), \\ (\text{"r}_{push1}\text{"} : R_{push1}), (\text{"r}_{push2}\text{"} : R_{push2}) \\ (\text{"r}_{sl}\text{"} : R_{sl}), (\text{"r}_{sm}\text{"} : R_{sm}) \end{array} \right\}$$

$$R_{get} = \{\!| \uparrow(\text{"req"}, \text{"unibo"}, r) |\!\} \longrightarrow \{\!| (\text{"req"}, r) |\!\}$$

$$R_{push1} = \{\!| (\text{"doc"}, r, doc) |\!\} \longrightarrow \{\!| \uparrow(\text{"doc"}, r, doc) |\!\}$$

$$R_{push2} = \left\{\!\!\left| \begin{array}{l} (\text{"doc"}, r, doc, j), \\ (j : J) \end{array} \right|\!\!\right\} \longrightarrow \left\{\!\!\left| \begin{array}{l} \uparrow(\text{"doc"}, r, doc, j), \\ \uparrow(j : J) \end{array} \right|\!\!\right\}$$

$$R_{sl} = \{\!| (\text{"down leporello"}) |\!\} \longrightarrow \{\!| \mathbf{tsc}(S_l) |\!\}$$

$$R_{sm} = \{\!| (\text{"down macbeth"}) |\!\} \longrightarrow \{\!| \mathbf{tsc}(S_m) |\!\}$$

Table 9: A group including two WWW servers: leporello and macbeth

that ask for documents in the server and the rules that put documents and Java applets in the root space.

Table 10 shows the space representing a browser: it contains the rule that generates and submits new request tuples, the rules to get documents and Java applets from the root space and the rules to terminate the browser.

Table 11 shows the space representing one of the servers: it contains a documents database and a set of Java applets. Moreover it contains the rules that get request tuples and satisfy them giving back the proper document and possibly a Java applet and the rules that simulate server failure. The specification of the other server is similar and it is not shown for brevity.

A specification is a description of a system we want to model. To get confidence in the specification we want to be able to study it and to reason about its behaviours. In particular we would like to have a way to demonstrate that a specification has some properties.

The specification of server replication models a system where browsers ask for documents and servers send the asked documents. Therefore we would like to prove that if a browser submits a request to a server group, it will eventually receive an answer and that any request is satisfied once.

In the following we adopt a logic based on TLA for software architectures modeled with PoliS, and we show how it can help in proving that a specification exhibits safety and liveness properties.

S_c

$$S_c = \left\{\!\!\left| \begin{array}{l} (\text{“r}_{cr}\text{”} : R_{cr}), (\text{“r}_{gr1}\text{”} : R_{gr1}), \\ (\text{“r}_{gr2}\text{”} : R_{gr2}), (\text{“local”}, 0), \\ (\text{“r}_{done}\text{”} : R_{done}), (\textbf{invariant} : R_{inv}), (\text{“Java”}), \end{array} \right|\!\!\right\}$$

$$R_{cr} = \{\!|(\text{“local”}, i_1)|\!\} \xrightarrow{(r, url, i_2) \leftarrow f(i_1)} \left\{\!\!\left| \begin{array}{l} (\text{“req”}, r), \\ (\text{“local”}, i_2), \\ \uparrow(\text{“req”}, url, r) \end{array} \right|\!\!\right\}$$

where $f(x) = (gen_req(x), gen_url(x), x + 1)$

$$R_{gr1} = \left\{\!\!\left| \begin{array}{l} \uparrow(\text{“doc”}, r, doc), \\ (\text{“req”}, r) \end{array} \right|\!\!\right\} \longrightarrow \{\!|(\text{“doc”}, r, doc)|\!\}$$

$$R_{gr2} = \left\{\!\!\left| \begin{array}{l} \uparrow(\text{“doc”}, r, doc, j), \\ \uparrow(j : J), (\text{“req”}, r) \end{array} \right|\!\!\right\} \longrightarrow \left\{\!\!\left| \begin{array}{l} (\text{“doc”}, r, doc), \\ (j : J) \end{array} \right|\!\!\right\}$$

$$R_{done} = \{\!|\,|\!\} \longrightarrow \{\!|(\text{“done”})|\!\}$$
$$R_{inv} = \{\!|?(\text{“done”})|\!\} \longrightarrow \{\!|\,|\!\}$$

Table 10: WWW: a browser

S_l

$$S_l = \left\{\!\!\left| \begin{array}{l} (\text{“r}_{s1}\text{”} : R_{s1}), (\text{“r}_{s2}\text{”} : R_{s2}), \\ (\text{“r}_{done}\text{”} : R_{done}), (\textbf{invariant} : R_{invl}), \\ (\text{“doc”}, \mathsf{r}_1, \mathsf{d}_1), \ldots, (\text{“doc”}, \mathsf{r}_n, \mathsf{d}_n), \\ (\text{“doc”}, \mathsf{r}_{n+1}, \mathsf{d}_{n+1}, \mathsf{j}_1), \ldots, (\text{“doc”}, \mathsf{r}_{n+m}, \mathsf{d}_{n+m}, \mathsf{j}_m), \\ (\mathsf{j}_1 : \mathsf{J}_1), \ldots, (\mathsf{j}_m : \mathsf{J}_m) \end{array} \right|\!\!\right\}$$

$$R_{s1} = \left\{\!\!\left| \begin{array}{l} \uparrow(\text{“req”}, r), \\ ?(\text{“doc”}, r, doc) \end{array} \right|\!\!\right\} \longrightarrow \{\!|\uparrow(\text{“doc”}, r, doc)|\!\}$$

$$R_{s2} = \left\{\!\!\left| \begin{array}{l} \uparrow(\text{“req”}, r), ?(j : J), \\ ?(\text{“doc”}, r, doc, j) \end{array} \right|\!\!\right\} \longrightarrow \left\{\!\!\left| \begin{array}{l} \uparrow(\text{“doc”}, r, doc, j), \\ \uparrow(j : J) \end{array} \right|\!\!\right\}$$

$$\mathsf{J}_1 = \{\!|?(\text{“Java”}), (\mathsf{j}_1 : \mathsf{J}_1)|\!\} \longrightarrow \{\!|(\text{“DoneJ”}, \mathsf{DJ}_1)|\!\}$$
$$\vdots$$
$$\mathsf{J}_m = \{\!|?(\text{“Java”}), (\mathsf{j}_m : \mathsf{J}_m)|\!\} \longrightarrow \{\!|(\text{“DoneJ”}, \mathsf{DJ}_m)|\!\}$$

$$R_{done} = \{\!|\,|\!\} \longrightarrow \{\!|(\text{“down”})|\!\}$$
$$R_{invl} = \{\!|?(\text{“down”})|\!\} \longrightarrow \{\!|\uparrow(\text{“down leporello”})|\!\}$$

Table 11: WWW: server Leporello

5 TLA Semantics

In this section we formally describe PoliS using a static analysis approach. Such an approach is chosen since it does not need any simulation of possible executions to infer program-wide properties by analyzing the specification document. Here we study PoliS semantics in terms of the Temporal Logic of Actions [22]. We will show how we use such a semantics to study safety and liveness properties in PoliS.

5.1 The Temporal Logic of Actions

TLA is a temporal logic used to specify and verify systems [20, 22]. A TLA specification is a logical formula describing all possible correct behaviours of a system.

TLA specifications can always be written in the form

$$\Phi \triangleq Init \wedge \Box[\mathcal{N}]_f \wedge L$$

where

$Init$ is a predicate specifying the set of allowed initial states

$\mathcal{N}$ is the specification *next-state* relation

f is the n-tuple of all flexible variables

L is a conjunction of fairness conditions

TLA formulae are interpreted on *behaviours*; a behaviour is an infinite sequence of *states* and a state is a mapping that assigns *values* to *variables*. Given a TLA specification $\Phi \triangleq Init \wedge \Box[\mathcal{N}]_f \wedge L$ and a behaviour $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$, σ satisfies Φ iff

- s_0 satisfies $Init$
- every pair of states (s_i, s_{i+1}) in σ satisfies $\mathcal{N}$ or leaves f unchanged
- L holds

State change is defined by actions that are boolean expressions built of primed and unprimed variables. An action is true or false with respect to a pair of states (s_i, s_{i+1}): non primed variables refer to state s_i, primed variables refer to state s_{i+1}.

A distinctive feature of TLA is the fact that a system is described not by a set of properties that must hold, but by a unique, global formula establishing allowed actions and actions execution modalities.

In TLA both systems and properties are represented in the same logic. The assertion "specification Φ has property P" is expressed by the validity of the formula $\Phi \Rightarrow P$ which asserts that every behaviour satisfying Φ satisfies P. P can be a safety property or a liveness property.

TLA users can take advantage of the existence of a theorem prover, named TLP, that can be used to certify proofs [13]. TLP is a (semi)automatic

verifier that allows to incrementally build and verify proofs in a structured and top-down fashion. The current TLP version is described in [12]. Presently TLP is made up of an interactive interface and of a translator acting as a front-end for the automatic verifier *Larch Prover* (LP) [15, 16]. The front-end is a translator that transforms TLP formulae in a codification understandable by LP and augments LP by TLA axioms and inference rules properly coded. TLP offers an attractive interactive development environment, based on emacs, that allows to write proofs and to start the verifier. In fact, we have used it to validate our specifications.

5.2 TLA semantics of a PoliS specification

The TLA semantics of a PoliS specification $Spec = (StartContext, Rules)$ is a TLA specification Φ whose *Init* predicate describes the initial state of PoliS specification, whose actions are the PoliS specification rule semantics, and whose fairness conditions describe rule fairness.

The idea behind the formal definition of TLA semantics is schematically shown in figure 5. The figure describes graphically how we translate a PoliS specification into a TLA formula.

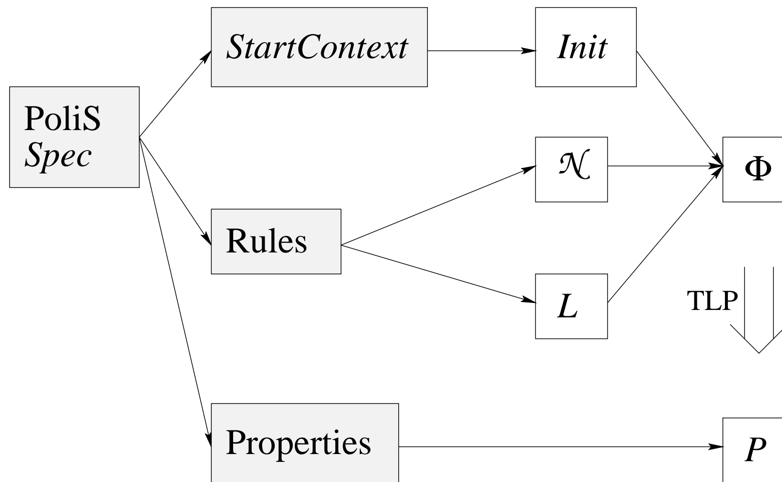

Figure 5: TLA semantics for PoliS

Any formal description of the PoliS coordination model has to give account of multiple tuple spaces and their nesting. A space tree is described using two TLA variables: an infinite multisets array named *mul*, and an infinite address array named *parent*. Every element in *mul* contains a space whereas every element in *parent* contains the address of the parent of a space: this means that $parent[i]$ is the address of the parent of space $mul[i]$.

Since TLA is a typeless logic, arrays are described through TLA functions. Hence *mul* and *parent* are TLA functions; however we will refer informally to *mul* and *parent* as arrays.

We recall that a TLA specification in canonical form includes an initial state, a set of actions, and a set of liveness properties. PoliS initial state

$$R = \left\{\begin{array}{l} t_{c,1}, \ldots, t_{c,n_c}, \\ \uparrow t_{ec,1}, \ldots, \uparrow t_{ec,n_{ec}}, \\ ?t_{t,1}, \ldots, ?t_{t,n_t}, \\ ?\uparrow t_{et,1}, \ldots, ?\uparrow t_{et,n_{et}}, \\ \mathbf{ask}(expr) \end{array}\right\} \xrightarrow{(\overline{y}) \leftarrow f(\overline{x})} \left\{\begin{array}{l} t_{p,1}, \ldots, t_{p,n_p}, \\ \uparrow t_{ep,1}, \ldots, \uparrow t_{ep,n_{ep}}, \\ \mathbf{tsc}(S_1), \ldots, \mathbf{tsc}(S_n) \end{array}\right\}$$

$$\mathbf{where}\ f(\overline{v}) = (f_1(\overline{v}), \ldots, f_m(\overline{v}))$$

where	$t_{i,j}$ is a generic tuple	$S_1, \ldots, S_n$ are multisets
	$\overline{x}$, $\overline{y}$ are n-tuples of identifiers	$f, f_1, \ldots, f_m$ are functions

Table 12: A generic rule

semantics asserts that the only existing space is the *StartContext*, that is the root space.

PoliS rule execution transforms multisets of tuples. Since a rule can contain formal tuples, it can be thought of as the set of the rules containing only non formal tuples that are admissible instances of the formal tuples of the rule. Consequently, a rule is translated in an action existentially quantified with respect to the values that can be assigned to the identifiers of the formal tuples. The action representing the rule semantics can be executed in a space if such a space contains the tuples to be read and consumed and it ensures that after its execution the space will not contain the tuples to be consumed and will contain the tuples to be produced. If a rule contains the primitive $\mathbf{tsc}(S)$, the action representing its semantics ensures that in the poststatus the space S will be added as a child of the space where the rule is executed.

In table 12 we present a non invariant rule R; notice that this is a generic definition since a rule could not contain some of the allowed components. The execution of rule R of table 12 in a multiset M_s child of a multiset M_f can be informally described in terms of preconditions and postconditions: preconditions are the properties that must be verified in order to have rule R executed, whereas the postconditions are the properties that are true after the execution of rule R.
Preconditions can be stated as follows:

- the program tuple containing rule R belongs to space M_s
- there is a multiset of tuples $\{\!\!\{s_{c,1}, \ldots, s_{c,n_c}, s_{t,1}, \ldots, s_{t,n_t}\}\!\!\}$ that is included in space M_s and that *matches* the multiset of tuples to be read and consumed locally $\{\!\!\{t_{c,1}, \ldots, t_{c,n_c}, t_{t,1}, \ldots, t_{t,n_t}\}\!\!\}$
- there is a multiset of tuples $\{\!\!\{s_{ec,1}, \ldots, s_{ec,n_{ec}}, s_{et,1}, \ldots, s_{et,n_{et}}\}\!\!\}$ that is included in space M_f and that *matches* the multiset of tuples to be read and consumed externally $\{\!\!\{t_{ec,1}, \ldots, t_{ec,n_{ec}}, t_{et,1}, \ldots, t_{et,n_{et}}\}\!\!\}$
- *expr* predicate of primitive **ask** is made true by the values assigned to the identifiers of the tuples to be read and consumed.

Postconditions can be stated as follows:

- tuples of multiset $\{\!\!\{s_{c,1}, \ldots, s_{c,n_c}\}\!\!\}$ are removed from multiset M_s
- tuples of multiset $\{\!\!\{s_{ec,1}, \ldots, s_{ec,n_{ec}}\}\!\!\}$ are removed from multiset M_f
- tuples $s_{p,1}, \ldots, s_{p,n_p}$ (that amount to tuples $t_{p,1}, \ldots, t_{p,n_p}$ with identifiers instantiated by the reading from the space and by local computation) are added to multiset M_s
- tuples $s_{ep,1}, \ldots, s_{ep,n_{ep}}$ (that amount to tuples $t_{ep,1}, \ldots, t_{ep,n_{ep}}$ with identifiers instantiated by the reading from the space and by local computation) are added to multiset M_f
- spaces $S_1, \ldots, S_n$ are added as children of space M_s.

A TLA action is a boolean expression built of variables in the pre and poststatus and hence it can represent an operation whose description is given in terms of pre and postconditions: TLA semantics of a PoliS rule is an action that is enabled if the rule execution preconditions are verified and whose poststatus verifies rule postconditions. In TLA rule description, multiset M_s and multiset M_f are *mul* elements and the creation of new spaces due to **tsc** is realized by adding elements to the array *mul*.

The action describing a rule is the disjunction of actions $\mathcal{N}(\overline{v})$ where $\overline{v}$ parametrizes the action with respect to the multisets where the rule can be executed and with respect to the values that must be assigned to the identifiers in the rule. Parametrization with respect to multisets describes the fact that a rule is potentially executable in any space because of its mobility; parametrization with respect to values describes the fact that a rule whose preactivation or postactivation contain formal tuples can be seen as the set of all rules that are admissible instantiations of the rule.

A new space is added to the multisets array by inserting it in one of *mul* free elements: the element where to put the new space is not deterministically fixed by the TLA action since the space position in *mul* has no semantical meaning.

An invariant rule is described in TLA as if it was an ordinary rule; its semantics however asserts that the space where the rule is executed will be removed by its execution. The semantics of invariant rules ensures that the space will disappear as soon as its contents violates the invariant (i.e. as soon as the space includes a multiset of tuples *matching* the invariant preactivaction). The preconditions of any non invariant rule are augmented in order to prevent the activation of a rule in a space while the space contains an enabled invariant.

A TLA specification can contain the description of liveness properties. PoliS intuitive semantics suggests that a rule is infinitely often executed if it is infinitely often enabled; this fairness property is ensured by asserting the strong fairness of every action describing the semantics of the specification rules.

TLA semantics of a PoliS specification *Spec* is the formula Φ whose *Init* predicate says that initially the only existing space is *StartContext*,

$$
\begin{array}{rcl}
Init & \triangleq & \wedge\ mul = [m \in Addr \mapsto \textbf{ if } m = 1 \textbf{ then } StartContext \textbf{ else } \bot] \\
 & & \wedge\ parent = [m \in Addr \mapsto \textbf{ if } m = 1 \textbf{ then } \mathsf{nil} \textbf{ else } \bot]
\end{array}
$$

Table 13: *Init* predicate

whose actions are the semantics of the specification rules and whose liveness properties assert the strong fairness for all actions.

5.3 Formal Semantics

Let us consider a PoliS specification $Spec = (StartContext, Rules)$ whose initial space is the multiset $StartContext = \{\!\{a_1, \ldots, a_n\}\!\}$ and whose *Rules* set contains the definitions of rules $R_1, \ldots, R_n$.

The semantics of the system initial state is predicate *Init* shown in table 13. *Init* predicate asserts that *mul* contains only the multiset *StartContext* as root of the space tree. Symbol $-$ is different from any multiset and address and it is used to distinguish free elements in *mul* and *parent*.

In the following we will define PoliS semantics referring also to the abbreviations defined on page 7. Moreover we will write formulae using the conventions suggested by Lamport in [21].

The semantic function that associates TLA semantics to any PoliS rule is defined by separately analyzing the different kind of rules that can be found in a PoliS specification: any rules category exhibits different features and hence a separate description helps both in explaining and in understanding semantics. PoliS rules can be partitioned into five categories: rules not interacting with parent space and not creating spaces, rules not interacting with parent space and creating spaces, rules interacting with parent space and not creating spaces, rules interacting with parent space and creating spaces, and invariant rules. In the following we will show the semantics of a generic rule not interacting with parent space and not creating spaces, of a generic rule interacting with parent space and creating spaces, and of a generic invariant rule. The semantics of rules belonging to the other categories is omitted for brevity but it can easily be inferred by analogy.

Since invariant rules must be executed as soon as they are enabled, any non invariant rule must be activated when no invariant is enabled in the space where the rule has to be executed. Such a constraint is given as a further precondition added to the preconditions of the semantics of any non invariant rule. To formally describe the absence of an enabled invariant rule in a space where a rule must be executed, we need to define function $Act_{|m}$:

$$Act_{|m}(\exists\, \overline{v} : \mathcal{N}(\overline{v}), m) = \exists\, \overline{v} : (v_1 = m) \wedge \mathcal{N}(\overline{v})$$

Given an action $\exists\, \overline{v} : \mathcal{N}(\overline{v})$ and a multiset address m, function $Act_{|m}$ is action $\mathcal{N}$ executable only in the multiset having address m.

Table 14 shows R_l semantics. R_l is a generic rule not interacting with the parent space and not creating new spaces. Action $\mathcal{N}_l(m, \overline{v})$ is enabled if m is the address of a space containing rule R_l, if values in $\overline{v}$ are correct with

$[\![R_l]\!]$

Syntactic description:

$$R_l = \left\{\!\!\left| \begin{array}{l} t_{c,1}, \ldots, t_{c,n_c}, \\ ?t_{t,1}, \ldots, ?t_{t,n_t}, \mathbf{ask}(expr) \end{array} \right|\!\!\right\} \xrightarrow{(\overline{y}) \leftarrow f(\overline{x})}\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\perp\!\!\perp\quad \{\!\!\mid \; t_{p,1}, \ldots, t_{p,n_p} \; \mid\!\!\}$$
$$\mathbf{where}\ f(\overline{v}) = (f_1(\overline{v}), \ldots, f_m(\overline{v}))$$

Semantic mapping:

$$[\![R_l]\!] \;\triangleq\; \exists\, \overline{v}_l \, : \mathcal{N}_l(\overline{v}_l)$$

where $\overline{v}_l = m, \overline{v}$ and

$$\begin{array}{l} \mathcal{N}_l(m, \overline{v}) \;\triangleq \\ \quad \wedge\; m \in Addr \\ \quad \wedge\; \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \\ \quad \wedge\; expr[\overline{v}/\overline{id}] \\ \quad \wedge\; \{\!\!\mid \overline{t}_c[\overline{v}/\overline{id}], \overline{t}_t[\overline{v}/\overline{id}] \mid\!\!\} \subseteq mul[m] \\ \quad \wedge\; (\text{“r}_l\text{”} \,:\, R_l) \in mul[m] \\ \quad \wedge\; \forall\, R \,:\, ((\mathbf{invariant} \,:\, R) \in mul[m] \Rightarrow \neg \textsc{Enabled}\ \langle Act_{|m}([\![R]\!], m) \rangle_w) \\ \quad \wedge\; mul' = [mul \ \ \text{EXCEPT} \\ \qquad\quad ![m] \;=\; (mul[m] \setminus \{\!\!\mid \overline{t}_c[\overline{v}/\overline{id}] \mid\!\!\}) \oplus \{\!\!\mid \overline{t}_p[\overline{v}/\overline{id}] \mid\!\!\}] \\ \quad \wedge\; parent' = parent \end{array}$$

Table 14: Semantics of a local rule

respect to function f and predicate *expr* and if the space having address m contains the tuples to be read and consumed. If preconditions are satisfied, the space having address m will be deprived of the tuples to be consumed and augmented by the tuples to be produced; the other elements in *mul* and *parent* will not be modified.

Action $[\![R_l]\!]$ is rule R_l semantics and it describes the fact that the rule is represented by the set of all actions $\mathcal{N}_l(\overline{v}_l)$ whose parameters in $\overline{v}$ respect local computation function f and predicate *expr*. Intuitively this means that a rule R amounts to the set of all rules with the same preactivation and postactivation and with identifiers properly instantiated with respect to f and *expr*.

Table 15 shows R_{ci} semantics. R_{ci} is a generic rule interacting with the parent space and creating new spaces $S_1, \ldots, S_n$.

Action $\mathcal{N}_{ci}(m_s, m_f, m_1, \ldots, m_n, \overline{v})$ is enabled if m_s is the address of a space containing rule R_{ci}, if m_f is the address of the parent space of space m_s, if addresses $m_1, \ldots, m_n$ point to unused *mul* and *parent* elements, if values in $\overline{v}$ respect function f and predicate *expr*, if the space having address m_s contains the tuples to be locally read and consumed, if the space having address m_f contains the tuples to be read and consumed externally.

If such preconditions are satisfied, the space m_s will be deprived of the tuples to be locally consumed and augmented by the tuples to be locally

produced. The space m_f will be deprived of the tuples to be externally consumed and augmented by the tuples to be externally produced; the new spaces will be added to array *mul* in the elements having addresses $m_1, \ldots, m_n$ and space m_s will become their parent; the elements in *mul* and *parent* having addresses different from $m_s, m_f, m_1, \ldots, m_n$ will not be modified.

Action $[\![R_{ci}]\!]$ is rule R_{ci} semantics; it says that the rule is represented by the set of all actions $\mathcal{N}_{ci}(\overline{v}_{ci})$ whose parameters in $\overline{v}$ respect local computation function f and predicate *expr* and whose m_f parameter corresponds to the address of the parent space of the space having address m_s.

Table 16 shows the semantics of a generic invariant rule R_{inv}. Action $\mathcal{N}_{inv}(m_s, m_f, \overline{v})$ is enabled if m_s is the address of a space containing rule R_{inv}, if m_f is the address of the parent space of the space having address m_s, if values in $\overline{v}$ respect local computation function f and predicate *expr* and if the space having address m_s contains the tuples to be read. If the preconditions are satisfied, the space having address m_s and all its descendants will be removed from arrays *mul* and *parent*; the space having address m_f will be augmented by the tuples to be produced externally; the elements in *mul* and *parent* having addresses different from m_s, m_f and from m_s descendants will not be modified.

Spaces nested in the space having address m_s are found using predicate $\mathit{IsAncestorOf}(m_1, m_2)$ that is true if $\mathit{mul}[m_1]$ is an ancestor of $\mathit{mul}[m_2]$. Formally:

$$\begin{array}{l} \mathit{IsAncestorOf}(m_1, m_2) \;\triangleq \\ \quad \vee\ \mathit{parent}[m_2] = m_1 \\ \quad \vee\ \exists\, m_3 \in \mathit{Addr} : \wedge\ \mathit{parent}[m_2] = m_3 \\ \qquad\qquad\qquad\qquad\quad \wedge\ \mathit{IsAncestorOf}(m_1, m_3) \end{array}$$

We remove all descendants of the space where an invariant fires and causes the termination of space S: in fact, they cannot survive their ancestor.

Action $[\![R_{inv}]\!]$ is rule R_{inv} semantics and it describes the fact that the rule is represented by the set of all actions $\mathcal{N}_{inv}(\overline{v}_{inv})$ whose parameters in $\overline{v}$ respect local computation function f and predicate *expr* and whose m_f parameter corresponds to the address of the parent space of the space having address m_s.

In order to define the TLA formula representing the semantics of a specification we have to define the action $\mathcal{N}$ that is the "next state" relation; since *Spec* possible actions are the rules in *Rules*, $\mathcal{N}$ is defined as the disjunction of the semantics of all the rules in a specification:

$$\mathcal{N} \;\triangleq\; [\![R_1]\!] \vee \ldots \vee [\![R_n]\!]$$

TLA specifications can express liveness conditions. Let w be the state function $\langle \mathit{mul}, \mathit{parent} \rangle$, formula L ensures the strong fairness of every specification action:

$$L \;\triangleq\; \forall\, \overline{v}_1 : \mathit{SF}_w(\mathcal{N}_1(\overline{v}_1)) \wedge \ldots \wedge \forall\, \overline{v}_n : \mathit{SF}_w(\mathcal{N}_n(\overline{v}_n))$$

The TLA formula representing the semantics of a PoliS specification is Φ, whose initial state predicate is *Init*, whose allowed actions are $[\mathcal{N}]_w$, and whose liveness conditions are given by formula L:

$[\![R_{ci}]\!]$

Syntactic description:

$$R_{ci} = \left\{\begin{array}{l} t_{c,1},\ldots,t_{c,n_c}, \\ \uparrow t_{ec,1},\ldots,\uparrow t_{ec,n_{ec}}, \\ ?t_{t,1},\ldots,?t_{t,n_t}, \\ ?\uparrow t_{et,1},\ldots,?\uparrow t_{et,n_{et}}, \\ \mathbf{ask}(expr) \end{array}\right\} \perp\!\!\perp \xrightarrow{(\overline{y})\leftarrow f(\overline{x})} \left\{\begin{array}{l} t_{p,1},\ldots,t_{p,n_p}, \\ \uparrow t_{ep,1},\ldots,\uparrow t_{ep,n_{ep}}, \\ \mathbf{tsc}(S_1),\ldots,\mathbf{tsc}(S_n) \end{array}\right\}$$

$$\textbf{where } f(\overline{v}) = (f_1(\overline{v}),\ldots,f_m(\overline{v}))$$

Semantic mapping:

$$[\![R_{ci}]\!] = \exists\,\overline{v}_{ci} : \mathcal{N}_{ci}(\overline{v}_{ci})$$

where $\overline{v}_{ci} = m_s, m_f, m_1, \ldots, m_n, \overline{v}$ and

$$\begin{array}{l}
\mathcal{N}_{ci}(m_s, m_f, m_1, \ldots, m_n, \overline{v}) \;\triangleq \\
\quad \wedge\ m_s, m_f, m_1, \ldots, m_n \in Addr \\
\quad \wedge\ parent[m_s] = m_f \\
\quad \wedge\ \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \wedge expr[\overline{v}/\overline{id}] \\
\quad \wedge\ \{\!\!\{\overline{t}_c[\overline{v}/\overline{id}], \overline{t}_t[\overline{v}/\overline{id}]\}\!\!\} \subseteq mul[m_s] \\
\quad \wedge\ (\text{“}\mathsf{r}_{ci}\text{”} : R_{ci}) \in mul[m_s] \\
\quad \wedge\ \{\!\!\{\overline{t}_{ec}[\overline{v}/\overline{id}], \overline{t}_{et}[\overline{v}/\overline{id}]\}\!\!\} \subseteq mul[m_f] \\
\quad \wedge\ mul[m_1] = \ldots = mul[m_n] = parent[m_1] = \ldots = parent[m_n] = \perp \\
\quad \wedge\ \forall R : ((\mathbf{invariant} : R) \in mul[m_s] \Rightarrow \neg\textsc{Enabled}\ \langle Act_{|m}([\![R]\!], m_s)\rangle_w) \\
\quad \wedge\ mul' = [mul\ \ \textsc{except} \\
\qquad\quad ![m_s] \ = \ (mul[m_s] \setminus \{\!\!\{\overline{t}_c[\overline{v}/\overline{id}]\}\!\!\}) \oplus \{\!\!\{\overline{t}_p[\overline{v}/\overline{id}]\}\!\!\}, \\
\qquad\quad ![m_f] \ = \ (mul[m_f] \setminus \{\!\!\{\overline{t}_{ec}[\overline{v}/\overline{id}]\}\!\!\}) \oplus \{\!\!\{\overline{t}_{ep}[\overline{v}/\overline{id}]\}\!\!\}, \\
\qquad\quad ![m_1] \ = \ S_1[\overline{v}/\overline{id}], \\
\qquad\qquad \vdots \\
\qquad\quad ![m_n] \ = \ S_n[\overline{v}/\overline{id}]] \\
\quad \wedge\ parent' = [parent\ \ \textsc{except} \\
\qquad\quad ![m_1] \ = \ m_s, \\
\qquad\qquad \vdots \\
\qquad\quad ![m_n] \ = \ m_s]
\end{array}$$

Table 15: Semantics of a rule interacting with parent space and creating new spaces

$[\![R_{inv}]\!]$

Syntactic description:

$$R_{inv} = \{\!\{?t_{t,1}, \ldots, ?t_{t,n_t}, \mathbf{ask}(expr)\}\!\} \perp\!\!\!\perp \xrightarrow{(\overline{y}) \leftarrow f(\overline{x})} \{\!\{\uparrow t_{ep,1}, \ldots, \uparrow t_{ep,n_{ep}}\}\!\}$$
$$\mathbf{where}\ f(\overline{v}) = (f_1(\overline{v}), \ldots, f_m(\overline{v}))$$

Semantic mapping:

$$[\![R_{inv}]\!] = \exists\, \overline{v}_{inv} : \mathcal{N}_{inv}(\overline{v}_{inv})$$

where $\overline{v}_{inv} = m_s, m_f, \overline{v}$ and

$$
\begin{array}{l}
\mathcal{N}_{inv}(m_s, m_f, \overline{v}) \quad \triangleq \\
\quad \wedge\ m_s, m_f \in Addr \\
\quad \wedge\ parent[m_s] = m_f \\
\quad \wedge\ \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \\
\quad \wedge\ expr[\overline{v}/\overline{id}] \\
\quad \wedge\ \{\!\{\overline{t}_t[\overline{v}/\overline{id}]\}\!\} \subseteq mul[m_s] \\
\quad \wedge\ (\mathbf{invariant} : R_{inv}) \in mul[m_s] \\
\quad \wedge\ mul' = [addr \in Addr \mapsto \\
\qquad \begin{array}{lll}
\mathbf{case} & addr = m_s & \rightarrow \bot \\
& addr = m_f & \rightarrow mul[m_f] \oplus \{\!\{\overline{t}_{ep}[\overline{v}/\overline{id}]\}\!\} \\
& IsAncestorOf(m_s, addr) & \rightarrow \bot \\
& \mathbf{else} & \rightarrow mul[addr]]
\end{array} \\
\quad \wedge\ parent' = [addr \in Addr \mapsto \\
\qquad \begin{array}{lll}
\mathbf{case} & addr = m_s & \rightarrow \bot \\
& IsAncestorOf(m_s, addr) & \rightarrow \bot \\
& \mathbf{else} & \rightarrow parent[addr]]
\end{array}
\end{array}
$$

Table 16: Semantics of an invariant rule

$$[\![Spec]\!]_{TLA} = \Phi \quad \triangleq \quad Init \wedge \Box[\mathcal{N}]_w \wedge L$$

Formula $\Phi = [\![Spec]\!]_{TLA}$ is satisfied only by all admissible behaviours. The TLA formula describing a PoliS system can be used to infer any safety or liveness property of the system itself through logical reasoning.

6 Case Study: Multiclient-Multiserver

In a multiserver-multiclient architecture a set of processes act as servers for a set of client processes. A client submits a request that some servers can satisfy. After serving a request, a server communicates the answer to the client. Any request is characterized by a type that determines the service needed.

Any server is able to satisfy a subset of the allowed types of requests and any client is able to generate a request whose type belongs to a subset of the allowed types of requests. This restriction means that a client could be prevented from accessing to some resources and that different servers can have different service abilities.

This multiserver-multiclient architecture can be specified in PoliS describing any server and any client with a space nested in a root space. Such a root space acts as a coordination medium uncoupling clients and servers: in fact, it allows clients to submit requests without explicitly indicating a server, whereas servers can satisfy requests without knowing from where they come from. To invoke a service a client puts a request tuple into its parent space; a server able to satisfy the request gets the request tuple, satisfies it and puts the result tuple in the parent space so that it can be read by the client which submitted the request.

Client-server systems having the architecture just described allow clients to be free not to know which processes are able to satisfy a given set of queries since a client can simply submit a request to the servers pool waiting for someone to give him back the result. In this way emphasis is put on the service rather than on the particular server process able to supply it.

We remark that this example can be seen both as an abstraction and as a simplification of the WWW example. We present the proof of a property of the client-server system, which can be considered as a simpler version of an analogous property of the WWW system.

6.1 Design Specification of a Software Architecture

We present the specification of a software architecture including two servers and one client. The set of possible requests by clients contains only two types of requests (namely "type_1" and "type_2"). The client is allowed to submit requests of either type. The first server is able to satisfy requests of either type, whereas the second server is able to satisfy only "type_1" requests.

Graphically the structure of the space tree describing the system architecture is shown in figure 6: servers and clients are nested in a common parent space coordinating their activities.

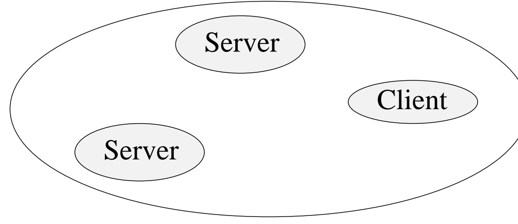

Figure 6: A client-server architecture

StartContext
$StartContext = \{\!\!\{ (\text{"}\mathsf{r}_c\text{"} : R_c) \}\!\!\}$
$R_c = \{\!\!\{ (\text{"}\mathsf{r}_c\text{"} : R_c) \}\!\!\} \longrightarrow \{\!\!\{ \mathbf{tsc}(S_1), \mathbf{tsc}(S_2), \mathbf{tsc}(C) \}\!\!\}$

Table 17: Client-multiserver: the *StartContext*

Table 17 shows the space defining the specification initial state. Rule R_c creates spaces C, S_1 and S_2 representing, respectively, the client, the server able to satisfy only "type_1" queries and the server able to satisfy queries of both types. Rule R_c is consumed by its activation: it is executed once at the beginning to create servers and client.

Table 18 shows a space C representing the client. Such a space contains tuples ("next_1", 1) and ("next_2", 1) that are used to generate new requests. Rule R_{cri} creates "type_i" requests: function req_i takes as an input the value of the second field of tuple ("next_i", n) and gives as an output a request r that is produced in the client space through tuple ("req", "type_i", r). Since the client can submit requests of both kinds, space C contains both rule R_{cr1} and rule R_{cr2}. Rule R_{ur} looks for request tuples in the client space and communicates them to the parent space, keeping local trace of the sent requests. Such a rule is able to handle tuples containing requests of both kinds: the second field of tuple ("req", *type*, r) is an identifier and hence the tuple matches any tuple requesting a service. Tuples ("requested", *type*, r) are produced locally in order to have an always updated knowledge of the queries submitted and not yet satisfied. Rule R_{gs} wants to find tuples containing answers to submitted queries; such tuples are searched for in the parent space where the queries were put. Rule R_{cs} consumes the answers; possible actions that could be taken after receiving an answer are not shown since they are not relevant for this specification.

Table 19 shows space S_1 representing the server able to process requests of "type_1"; table 20 shows space S_2 representing the server able to process requests of both kinds. Since S_1 and S_2 differ only because server S_2 is able to handle also requests of "type_2", server S_1 rules are a subset of server S_2 rules.

Rules R_{sri} handle "type_i" requests by means of function $serve_i$; answers

C

$$C = \left\{\!\!\left| \begin{array}{l} (\text{“next}_1\text{”}, 1), (\text{“next}_2\text{”}, 1), (\text{“r}_u\text{r”} : R_{ur}), (\text{“r}_g\text{s”} : R_{gs}), \\ (\text{“r}_{cs}\text{”} : R_{cs}), (\text{“r}_{cr1}\text{”} : R_{cr1}), (\text{“r}_{cr2}\text{”} : R_{cr2}) \end{array} \right|\!\!\right\}$$

$$R_{cr1} = \{\!|(\text{“next}_1\text{”}, i_1)|\!\} \xrightarrow{(i_2, r) \leftarrow f(i_1)} \{\!|(\text{“req”}, \text{“type}_1\text{”}, r), (\text{“next}_1\text{”}, i_2)|\!\}$$
where $f(x) = (x + 1, req_1(x))$

$$R_{cr2} = \{\!|(\text{“next}_2\text{”}, i_1)|\!\} \xrightarrow{(i_2, r) \leftarrow f(i_1)} \{\!|(\text{“req”}, \text{“type}_2\text{”}, r), (\text{“next}_2\text{”}, i_2)|\!\}$$
where $f(x) = (x + 1, req_2(x))$

$$R_{ur} = \{\!|(\text{“req”}, type, r)|\!\} \longrightarrow \{\!| \uparrow(\text{“req”}, type, r), (\text{“requested”}, type, r)|\!\}$$

$$R_{gs} = \left\{\!\!\left| \begin{array}{l} (\text{“requested”}, type, r), \\ \uparrow(\text{“served”}, type, r, s) \end{array} \right|\!\!\right\} \longrightarrow \{\!|(\text{“served”}, type, r, s)|\!\}$$

$$R_{cs} = \{\!|(\text{“served”}, type, r, s)|\!\} \longrightarrow \{\!||\!\}$$

Table 18: Client-multiserver: client C

S_1

$$S_1 = \{\!|(\text{“r}_{us}\text{”} : R_{us}), (\text{“r}_{sr1}\text{”} : R_{sr1}), (\text{“r}_{gr1}\text{”} : R_{gr1})|\!\}$$

$$R_{sr1} = \{\!|(\text{“req”}, \text{“type}_1\text{”}, r)|\!\} \xrightarrow{(s) \leftarrow f(r)} \{\!|(\text{“served”}, \text{“type}_1\text{”}, r, s)|\!\}$$
where $f(x) = (serve_1(x))$

$$R_{gr1} = \{\!| \uparrow(\text{“req”}, \text{“type}_1\text{”}, r)|\!\} \longrightarrow \{\!|(\text{“req”}, \text{“type}_1\text{”}, r)|\!\}$$

$$R_{us} = \{\!|(\text{“served”}, type, r, s)|\!\} \longrightarrow \{\!| \uparrow(\text{“served”}, type, r, s)|\!\}$$

Table 19: Client-multiserver: server S_1

S_2

$$S_2 = \left\{\!\!\left| \begin{array}{l} (\text{“r}_{us}\text{”} : R_{us}), (\text{“r}_{sr1}\text{”} : R_{sr1}), (\text{“r}_{sr2}\text{”} : R_{sr2}), \\ (\text{“r}_{gr1}\text{”} : R_{gr1}), (\text{“r}_{gr2}\text{”} : R_{gr2}) \end{array} \right|\!\!\right\}$$

$$R_{sr1} = \{\!|(\text{“req”}, \text{“type}_1\text{”}, r)|\!\} \xrightarrow{(s)\leftarrow f(r)} \{\!|(\text{“served”}, \text{“type}_1\text{”}, r, s)|\!\}$$
where $f(x) = (serve_1(x))$

$$R_{sr2} = \{\!|(\text{“req”}, \text{“type}_2\text{”}, r)|\!\} \xrightarrow{(s)\leftarrow f(r)} \{\!|(\text{“served”}, \text{“type}_2\text{”}, r, s)|\!\}$$
where $f(x) = (serve_2(x))$

$$R_{gr1} = \{\!| \uparrow(\text{“req”}, \text{“type}_1\text{”}, r)|\!\} \longrightarrow \{\!|(\text{“req”}, \text{“type}_1\text{”}, r)|\!\}$$

$$R_{gr2} = \{\!| \uparrow(\text{“req”}, \text{“type}_2\text{”}, r)|\!\} \longrightarrow \{\!|(\text{“req”}, \text{“type}_2\text{”}, r)|\!\}$$

$$R_{us} = \{\!|(\text{“served”}, type, r, s)|\!\} \longrightarrow \{\!| \uparrow(\text{“served”}, type, r, s)|\!\}$$

Table 20: Client-multiserver: server S_2

are produced locally as tuples (“served”, “type$_i$”, r, s). Server S_1 contains only rule R_{sr1} and server S_2 contains both rule R_{sr1} and rule R_{sr2}. Rules R_{gri} look in the parent space in order to find requests of “type$_i$” and put them in the server where they can be processed. Server S_1 contains only rule R_{gr1} and server S_2 contains both rule R_{gr1} and rule R_{gr2}. Rule R_{us}, owned by both servers, looks for answer tuples and puts them in the parent space from which the client will be able to get them.

This specification is satisfied by behaviours that allow a *request_tuple* to be submitted, processed and given back as a *served_tuple* going through the following steps:

- *request_tuple* is produced in the client space
- *request_tuple* is put in the parent space
- *request_tuple* is taken by one of the servers if it is of “type$_1$”; it is taken by server S_2 if it is of “type$_2$”
- *request_tuple* is processed by the proper rule in the space that took it
- *served_tuple* is put in the parent space
- *served_tuple* is removed from the parent space and put in the client space
- *served_tuple* is consumed by the client.

The actions that are performed between the request creation and the answer reception are not the only executable actions: they can be interleaved by other actions and they can be executed simultaneously with other actions.

$$Request(\overline{t}) \triangleq \wedge\ Stable(\overline{t}) \\ \wedge\ (\text{“}\mathsf{req}\text{”}, \text{“}\mathsf{type}_1\text{”}, \mathsf{req}) \in mul[t_3]$$

$$Received(\overline{t}) \triangleq (\text{“}\mathsf{served}\text{”}, \text{“}\mathsf{type}_1\text{”}, \mathsf{req}, serve_1(\mathsf{req})) \in mul[t_3]$$

Table 21: Predicates *Request* and *Received*

Such a description was presented only to show that any behaviour of the specification will lead to the answer reception performing the given sequence of steps.

6.2 Analysis

In order to show how PoliS can take advantage of the TLA logic, we present the formal proof of the fact that the specification given above describes behaviours such that any request submitted by the client will eventually be satisfied.

In the following we will demonstrate that if a client produces a request tuple $(\text{“}\mathsf{req}\text{”}, \text{“}\mathsf{type}_1\text{”}, \mathsf{req})$ with $\mathsf{req} = req_1(\mathsf{k})$, such a request will be added to the root space from which one of the two servers is able to get and process it, answering back in the root space to the client.

The formal specification of such an informally described property is given by the following TLA formula:

$$\Phi \Rightarrow \forall\, \overline{t} \in T\ :\ (Request(\overline{t}) \leadsto Received(\overline{t})) \tag{1}$$

where Φ represents TLA semantics of system PoliS specification. Formula Φ is not shown for brevity but it can easily be derived by applying the semantic mapping function described in section 5. Formula 1 asserts that the specification implies that, for every allowed behaviour, if there is a state s_{req} satisfying predicate $Request(\overline{t})$, then there will be a state following s_{req} satisfying predicate $Received(\overline{t})$. Predicates $Request(\overline{t})$ and $Received(\overline{t})$ are shown in table 21.

Predicate $Request(\overline{t})$ asserts that the space representing the client contains the request $(\text{“}\mathsf{req}\text{”}, \text{“}\mathsf{type}_1\text{”}, \mathsf{req})$. Request req of “type_1” is chosen to simplify the proof without loss of generality: choosing a request $\mathsf{req} = req_1(\mathsf{k})$ for a generic k amounts to choosing any request of “type_1”.

Predicate $Received(\overline{t})$ asserts that the space representing the client contains the tuple $(\text{“}\mathsf{served}\text{”}, \text{“}\mathsf{type}_1\text{”}, \mathsf{req}, serve_1(\mathsf{req}))$ that is the answer to the request represented by tuple $(\text{“}\mathsf{req}\text{”}, \text{“}\mathsf{type}_1\text{”}, \mathsf{req})$.

Predicate $Stable(\overline{t})$ is a factor of predicate $Request(\overline{t})$ and it describes the way the rules are distributed in the spaces and the way the spaces are distributed in array mul. These informations are not strictly needed to express the property but they are necessary for the formal proofs.

Formula 1 introduces array $\overline{t}$, that represents a way to put spaces in array mul; it is used to keep trace of the location of client and server spaces in mul.

Formula 1 asserts that specification Φ implies that $\mathit{Request}(\overline{t}) \leadsto \mathit{Received}(\overline{t})$ for any $\overline{t}$. This means that its meaning is independent from the place where a space is put in array *mul*: such knowledge is needed to record the relations among spaces but it plays no role in the abstract expression of the property we have to prove.

The proof of formula 1 is structured in agreement with the behaviours we expect from the specification. Such a structure is graphically shown in figure 7; it shows a diagram where a node represents any state satisfying the predicate labelling that node. An arrow from node n_1 to node n_2 means that it is possible to prove that $\Phi \Rightarrow \mathit{label}(n_1) \leadsto \mathit{label}(n_2)$.

We want to stress that this diagram is not a state-transition diagram: the existence of an arrow from a node to another node means that from a state satisfying the predicate labelling the first node it will be possible to eventually reach a state satisfying the predicate labelling the second node for any behaviour satisfying specification Φ.

In figure 7 the starting node represents states satisfying predicate *Request*, namely where the client has produced a request (“**req**”, “**type**$_1$”, **req**).

From such a node it is possible to reach node Up_r representing states in which the request has been inserted in the root space. From node Up_r both node Down_1 and node Down_2 are reachable; these nodes represent respectively states in which the request has been taken by server S_1 or by server S_2. From node Up_r we can however reach node Up_s since both from node Down_1 and from node Down_2 it is possible to reach node Up_s that represents states in which tuple (“**served**”, “**type**$_1$”, **req**, serve_1(**req**)) is in the root space.

From node Down_1 it is possible to reach node Served_1 representing states where server S_1 has processed the request and similarly from node Down_2 it is possible to reach node Served_2. From nodes Served_i it is possible to reach node Up_s since the answer tuple will eventually be transferred in the root space regardless of the server that produced it. From node Up_s it is possible to reach node *Received* since the client can take the answer when it is in the parent space.

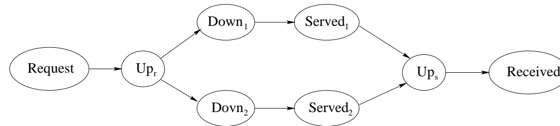

Figure 7: Lemma *RequestToReceived* structure

The formal proof of formula 1 validity is built of a set of TLA theorems and lemmas. For the sake of brevity in this paper we will only present the proof general structure that is shown in two lemmas verified with TLP.

Lemma *ForallTRequestToReceived* of table 22 proves the validity of formula 1. Such a lemma is proved in two steps: step $\langle 1 \rangle 1$ proves that $\Phi \Rightarrow \mathit{Request}(\overline{t}) \leadsto \mathit{Received}(\overline{t})$ for any fixed $\overline{t}$ and step $\langle 1 \rangle 2$ proves that the validity of a formula instantiated by a generic parameter implies the validity of the formula universally quantified.

```
Lemma ForallTRequestToReceived
  Phi => Forall t: Request(t) ~>  Received(t) [* RToRF *]
Proof
  <1>1 Phi => Request(t) ~> Received(t)
  Qed
  <1>2 (Request(t) ~> Received(t)) => \
         (Forall j: Request(j) ~>  Received(j) [* RToRF *])
  By-Implication
  Activate FunctionDefinitions*
  Instantiate ProveTempForall with ab_f <- f_RToRF, a_x <- (t)
  Qed
Activate ImpliesTransitivity
Qed
```

Table 22: Lemma *ForallTRequestToReceived*

Step $\langle 1 \rangle 1$ of lemma *ForallTRequestToReceived* of table 22 is proved in table 23 by lemma *RequestToReceived* that presents the proof of the validity of formula $\Phi \Rightarrow Request(\overline{t}) \leadsto Received(\overline{t})$ for a fixed $\overline{t}$.

To satisfy the request ("req", "type_1", req) we have to perform some steps: the request must be delivered to the root space; it must be input by one of the servers; it must be satisfied; the answer tuple must be put in the root space; the client must input the answer tuple from the root space. Such an expected behaviour explains the structure on which lemma *RequestToReceived* proof was built as shown by diagram in figure 7.

The substeps of lemma of table 22 are proved through TLP lemmas that are mainly based on the application of the inference rule WF1 of TLA proof system; such a rule suggests the method to prove the validity of formulae $\Phi \Rightarrow P \leadsto Q$ in cases in which it is possible to find an action whose execution establishes Q if executed in a state satisfying P.

7 Conclusions

We have presented a formal method for specifying and analyzing coordination applications. PoliS offers a conceptual framework to formally design and develop distributed software architectures for the new class of systems based on interoperability and mobility of software components.

In contrast to other formal approaches to coordination, like [10, 6, 7, 9, 19, 11, 23], PoliS is oriented to specification design of software architectures, namely PoliS documents are used to reason on the coordination architecture of the systems being designed, rather than on their actual behavior as programs. The reader interested in the behavioral approach should refer to [9], in which it is shown how the framework of process algebras can be used to develop an abstract coordination model useful to study the semantics of object-oriented coordination languages.

We believe that the main result of this paper is the development of a formal method (based on TLA logic) for reasoning on coordination of

```
Lemma RequestToReceived
  Phi => Request(t) ~> Received(t)
Proof
  <1>1 Phi => Request(t) ~> Upr(t)
  Qed
  <1>2 Phi => Upr(t) ~> Ups(t)
    <2>1 Phi => Upr(t) ~> (Down1(t) \/ Down2(t))
    Qed
    <2>2 Phi => Down1(t) ~> Ups(t)
      <3>1 Phi => Down1(t) ~> Served1(t)
      Qed
      <3>2 Phi => Served1(t) ~> Ups(t)
      Qed
    Activate LatticeTransitivityHyp
    Qed
    <2>3 Phi => Down2(t) ~> Ups(t)
      <3>1 Phi => Down2(t) ~> Served2(t)
      Qed
      <3>2 Phi => Served2(t) ~> Ups(t)
      Qed
    Activate LatticeTransitivityHyp
    Qed
  LatticeDisjunctionIntrHyp (Phi) (Down1(t)) \
                            (Down2(t)) (Ups(t))
  Activate LatticeTransitivityHyp
  Qed
  <1>3 Phi => Ups(t) ~> Received(t)
  Qed
Activate LatticeTransitivityHyp
Qed
```

Table 23: Lemma *RequestToReceived*

multiple tuple spaces. In comparison, Swarm [10] was the first attempt to define a (Unity-like) logic to reason on a Linda-like coordination model. This approach in some way inspired the work discussed in [6], which shows how it is possible to design a coordination program by refinement of a formal specification. Another attempt to use a logic to formally reason on the coordination language Gamma is given in [25]. All these approaches use a monolithic data space.

The work [7] explored the formal semantics of a coordination model similar to PoliS, including a hierarchy of spaces called blackboards, in which however no program manipulation or mobility was allowed.

In [19] the coordination model Bauhaus is fully developed: it is based on nested multisets like PoliS, and it is somewhat more elegant than PoliS, because only one simple concept, namely the multiset, is used, in different forms, for representing dataspaces, tuples, and programs. However, even if Bauhaus has been introduced as an evolution of Linda to study software architectures as well, as its name suggests, currently there is no logic to help designers to reason on their documents. We believe that PoliS can be adapted to support Bauhaus design: this is a topic for further research.

Possibly, the closest work in spirit to what we have discussed here is the μlog model with multiple blackboards introduced in [11]. However, their formal context is logic programming and it is unclear how smoothly μlog with multiple blackboards integrates with classic LP semantic framework.

We are currently using PoliS and its formal apparatus as a tool to specify, analyze, and design software architectures, namely software structures made of components which can interoperate and interact [26].

Acknowledgments. We thank TLP designers, who gave us permission to install and use TLP on our machines (the original version was for Solaris, we have ported it to SunOS; the new version is available on request).

This paper has been partially sponsored by Italian MURST and CNR.

References

[1] 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.

[2] J. Andreoli, C. Hankin, and D. LeMetayer, editors. *Coordination Programming: Mechanisms, Models and Semantics*. Imperial College Press, 1996.

[3] J. Banatre and D. LeMetayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[4] J. Banatre and D. LeMetayer. Gamma and the Chemical Reaction Model: Ten Years After. In J. Andreoli, C. Hankin, and D. LeMetayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, pages 3–41. Imperial College Press, 1996.

[5] M. Banville. Sonia: an Adaptation of Linda for Coordination of Activities in Organizations. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*, pages 57–74, Cesena, Italy, April 1996. Springer-Verlag, Berlin.

[6] N. Brown. Correctness-Preserving Transformations for the Design of Parallel Programs. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 29–48. Springer-Verlag, Berlin, 1995.

[7] X. Chen and C. Montangero. Compositional Refinements in Multiple Blackboard Systems. *Acta Informatica*, 32(5):415–458, 1995.

[8] P. Ciancarini. PoliS: a Programming Model for Multiple Tuple Spaces. In C. Ghezzi and G. Roman, editors, *Proc. 6th IEEE Int. Workshop on Software Specification and Design*, pages 44–51, Como, Italy, October 1991. IEEE Computer Society Press.

[9] P. Ciancarini, R. Gorrieri, and G. Zavattaro. Towards a Calculus for Generative Communication. In E. Najm and J. Stefani, editors, *Proc. IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems*, pages 289–306, Paris, France, 1996. Chapman and Hall, London.

[10] H. Cunningham and G. Roman. A Unity-Style Programming Logic for Shared Dataspace Programs. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, July 1990.

[11] K. DeBosschere and J. Jacquet. m2log: Towards Remote Coordination. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*, pages 142–159, Cesena, Italy, April 1996. Springer-Verlag, Berlin.

[12] U. Engberg. *Reasoning in the Temporal Logic of Actions.* PhD thesis, CS Dept., Univ. of Aarhus, Denmark, September 1995.

[13] U. Engberg, P. Gronning, and L. Lamport. Mechanical verification of Concurrent Systems with TLA. In U. Martin and J. Wing, editors, *First Int. Workshop on Larch*, Workshops in Computing, pages 86–97. Springer-Verlag, Berlin, 1992.

[14] D. Garlan and M. Shaw. *Software Architecture. Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[15] S. Garland and J. Guttag. An overview of LP, the Larch Prover. In B. Springer-Verlag, editor, *Proc. 3rd Int. Conf. on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 137–151, 1989.

[16] S. J. Garland and J. V. Guttag. A guide to LP, the Larch Prover. Technical Report RR 82, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, Dec. 1991.

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

[18] 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.

[19] S. Hupfer. *Turingware: An Integrated Approach to Collaborative Computing.* PhD thesis, Dept. of Computer Science, Yale University, New Haven, CT, December 1996.

[20] L. Lamport. Verification and Specifications of Concurrent Programs. In J. deBakker, W. deRoever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 347–374. Springer-Verlag, Berlin, 1993.

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

[22] L. Lamport. TLZ. In J. Bowen and J. Hall, editors, *Proc. 8th Z Users Workshop (ZUM94)*, Workshops in Computing, pages 267–268, Cambridge, 1994. Springer-Verlag, Berlin.

[23] D. LeMetayer. Software Architecture Styles as Graph Grammar. In D. Garlan, editor, *Proc. 4th ACM SIGSOFT Conf. on Foundations of Software Engineering*, volume 21:6 of *ACM SIGSOFT Software Engineering Notes*, pages 15–23, 1996.

[24] O. Nierstrasz. Composing Active Objects. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Trends in Object-Based Concurrent Computing*, pages 151–173. MIT Press, Cambridge, MA, 1993.

[25] M. Reynolds. Temporal Semantics for Gamma. In J. Andreoli, C. Hankin, and D. LeMetayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, pages 141–170. Imperial College Press, 1996.

[26] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In J. vanLeeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, Berlin, 1995.

[27] P. Wegner. Tradeoffs between Reasoning and Modeling. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Trends in Object-Based Concurrent Computing*, pages 22–41. MIT Press, Cambridge, MA, 1993.