# Using a Coordination Language to Specify the Invoicing System

Paolo Ciancarini and Cecilia Mascolo

Dipartimento di Scienze dell'Informazione,
Università di Bologna,
Mura Anteo Zamboni ,7, I-40127 Bologna (Italy)
tel: +39 51 354506, fax: +39 51 354510
e-mail: {ciancarini, mascolo}@cs.unibo.it

**Abstract.** In this paper we present the specification of an "invoicing system": two versions of it are shown. The first does not take into account any interaction with the environment while the second considers it. The specifications are written in PoliS, a coordination language based on multiple tuple spaces.
The operational semantics of PoliS allows the specification of behavioral aspects of the systems enhancing the dynamics. We have analyzed the specifications with a parser in order to avoid syntax errors, and with a model checker that helps in understanding the dynamics of the system specified. The model checker allowed us to verify some temporal properties on the Invoicing Systems.
Some questions have risen due to the incompleteness of the requirements document: we discuss the possible evolutions of the two specifications.

## 1 Introduction

The comparison among formal techniques for the specification of systems is a very interesting research topic. Different questions and solutions raise from different methods and different analysis approaches.

We propose a specification language based on the coordination paradigm [2] called PoliS [6].

PoliS focuses on the behavioral aspects of systems and some liveness and safety properties can be analyzed on the specifications.

In this paper we show how PoliS allows a natural specification of the systems in terms of multiple tuple spaces.

The case study proposed (an invoicing system) is quite simple and the specification we give is immediate. Due to the incompleteness of the requirement document, some questions have risen from our specification: we believe that the use of PoliS for such a case study has been convenient in term of clearness, extensibility and flexibility. In section 5 we present some possible evolutions of the specification.

As we built some automatic tools for PoliS (a theorem prover [6] and a model checker[5]) we have analyzed the case study specification isolating and verifying some interesting properties on the system behavior.

The paper is organized as follows: Section 2 introduces the PoliS language with a simple example of use of the language. In section 3 we present the case study requirements (divided in Case1 and Case2) and the corresponding PoliS specifications. In section 4 we introduce an automatic model checker for PoliS specifications. Section 5 contains a discussion on the specifications, a list of questions risen from it, and a set of possible enhancements of the specifications. The section also lists the results of the automatic analysis performed using the PoliS model checker. In the last section we conclude and present some future work.

## 2 Overview of PoliS

PoliS is a coordination language based on multiple tuple spaces [3]. A tuple space, or *space* for short, includes both tuples and other spaces. 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*, which are ordered sequences of values, and *program tuples*, which contain the coordination rules that manage activities inside the space they belong to. The execution of a program tuple is an agent, which can modify a space tree removing tuples and adding tuples and spaces. However, an agent 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 such a 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.

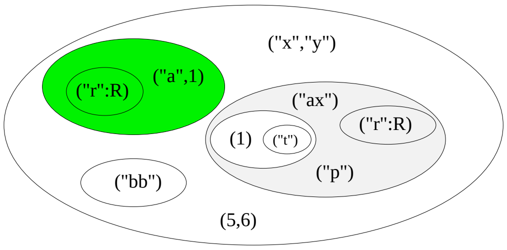


**Fig. 1.** A PoliS space tree

A space is a multi-set of tuples. A space is modified by reactions that transform multi-sets of tuples in multi-sets of tuples (this is multi-set rewriting, and is common to most coordination models based on *generative communication* [2]). The *rule* is the construct that defines which reactions can take place. A rule can act on the tuples of the space in which it resides and in the tuples of the parent space of this space: we will call this spaces the rule scope. A rule defines a reaction that reads and consumes tuples in its scope, performs a sequential computation, produces new tuples in its scope and creates new subspaces.

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

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

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

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

A program tuple is a multi-set rewriting rule: preactivation and postactivation are multi-sets and the local computation is written as annotation on the arrow between preactivation and postactivation. A tuple in the preactivation must be read if the symbol "?" is put in front of it and must be consumed otherwise; a read or consume operation involves the parent space if the symbol "↑ " is put in front of a tuple and involves the local space if the symbol is missing; a tuple in the postactivation must be produced in the parent space if the symbol " ↑" is put in front of it and must be produced locally otherwise. The input/output from/to the environment is simulated using this symbol: the main space can accept/emit tuples from/to an hypothetical external tuple space (i.e. the environment) using symbol " ↑" .

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

**$Startcontext$**

$Startcontext = \{\!|\, (\text{“}r_g\text{”} : R_g) \,|\!\}$

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

**$S_p$**

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

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

**$S_c$**

$S_c = \{\!|\, (\text{”}next_c\text{”}, 0), (\text{”}r_c\text{”} : R_c), \textbf{invariant } (R_{inv}) \,|\!\}$

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

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

**Table 1.** Specification of the Producer-Consumer System

In PoliS rules specification is written below the specification of the space containing it.

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

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

A key feature in PoliS is that a space tree can evolve dynamically: a new space is created by the primitive **tsc** (for *tuple space create*) and any space can be removed because of the execution of a special rule named *invariant* that terminates the space where it is executed. The execution of a rule containing a

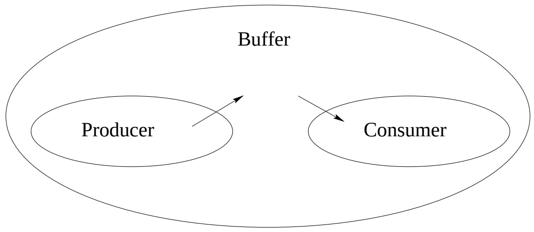


**Fig. 2.** Producer-consumer: spaces topology

**tsc**($M$) operation in its postactivation causes the multi-set $M$ to be added as a child space of the space where the rule was executed.

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

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

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

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

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

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

Communication is inspired by Linda [9]: tuples representing messages are put in a space shared by agents which have to communicate. Hence, communication is decoupled because agents do not know each other, since they access tuples by pattern matching. Since messages have no destination address, their contents determine the set of possible receivers, (communication is property driven).

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

## 3 Requirement and Specification of the Invoicing System

In this section we give the PoliS specification of the two versions of the case study *Invoicing System*.

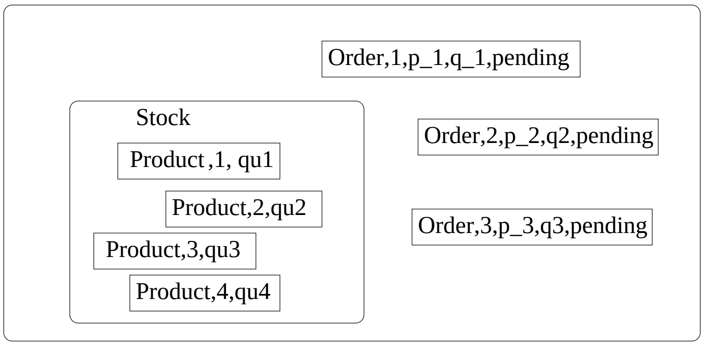


**Fig. 3.** Structure of Case1

### 3.1 The PoliS specification of the Invoicing System (Case 1)

We suppose to have a closed world: updating, input of new orders, and cancellation of orders have not to be taken into account. Then the initial space will look like the one in figure 3.

The main space contains the orders to be invoiced. Every order is defined by a number (i.e. the id), the product reference, the quantity of the product ordered, and the state of the order (*pending/invoiced*).

The *Stock* space is a sub-space of the main space (Fig. 3). It is specified in the second part of table 3.1: it contains the stocked products, their names and the stocked quantities.

The PoliS rule $IN$, in the *Stock* space, looks for an order in the parent space, and, if a sufficient quantity of the stocked product exists, it updates the stocked quantity of the product and invoices the order.

Then some questions have risen from this specification. As the rule $IN$ checks if the quantity of product stocked is larger than the quantity ordered (by the construct **ask**), the question "what happens if an order asks for a quantity larger than the one present in the stock?"

We can ask the customer some details about this situation or proposing a solution: adding a rule that cancel an order in case the asked quantity of product

$$\text{Startcontext}$$
$$Startcontext = \left\{\left| \begin{array}{l} Stock, (\text{"}Order\text{"}, 1, p_1, q_1, pending), (\text{"}Order\text{"}, 2, p_2, q_2, pending), \\ ..., (\text{"}Order\text{"}, k, p_k, q_k, pending) \end{array} \right|\right\}$$

$$\text{Stock}$$
$$Stock = \left\{\left| \begin{array}{l} (\text{"}Product\text{"}, p_1, q_1), (\text{"}Product\text{"}, p_2, q_2), \\ ..., (\text{"}Product\text{"}, p_n, q_n), (\text{"}invoice\text{"} : IN) \end{array} \right|\right\}$$
$$IN = \left\{\left| \begin{array}{l} (\text{"}Product\text{"}, p_i, q_i), \\ \mathbf{ask}(q_j \leq q_i) \\ \uparrow(\text{"}Order\text{"}, j, p_i, q_j, pending) \end{array} \right|\right\} \xrightarrow{(q_m) \leftarrow f(q_j, q_i)} \left\{\left| \begin{array}{l} (\text{"}Product\text{"}, p_i, q_m), \\ \uparrow(\text{"}Order, , j, p_i, q_j, invoiced) \end{array} \right|\right\}$$
**where** $f(q1, q2) = (diff(q1, q2))$

**Table 2.** PoliS specification of Case1

is larger than the stocked one. This implies also the adding of the state *canceled* to the order states *invoice* and *pending*.

An other interesting question is "if an order asks for a non-stocked product?". Again, we can choose to handle this case with a rule that cancel the order when it asks for a non-stocked product.

### 3.2 The PoliS specification of the Invoicing System (Case 2)

Now we specify a refinement of Case1, taking into account also the input of new orders, the cancellation of orders and the entries of new quantities in the stock.

In this version of the invoicing system the main space can accept input from the environment exploiting the operation ($\uparrow$) (see section 2 for details). Figure 4 shows the whole system.

Orders can now be in the state *canceled*, besides *pending* and *invoiced*.

The refined specification allows the system to interact with the environment: new orders and new quantities for the stock can be accepted, orders can be canceled (they remain in the space as canceled).

The main space now contains new rules for the handling of these situations; the counter tuple introduced records the number of orders accepted (it is updated by the rule $GET$). The rule $GET$ accepts a new order recording it as pending. The counter helps in assign sequential identifier to the input orders.

The rule $CANC$ cancels an order marking it as canceled.

The rule $NEWQUANT$ simply accept a new quantity of a product as input from the environment. The rule $SEND$ communicate to the environment an invoiced order.

The *Stock* space is now enriched with an $UPDATE$ rule that updates the product quantities in the stock with the new quantity received from the main space.

Some questions have risen also from this specification. The question risen on the specification of Case1 "what happens if an order asks for a quantity larger

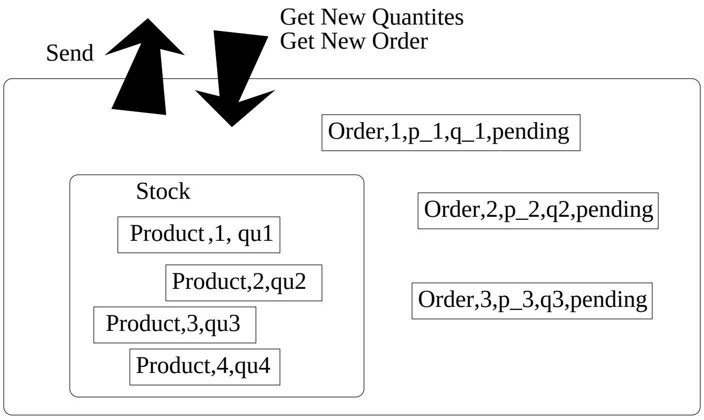


**Fig. 4.** Structure of Case2

*Startcontext*

$$Startcontext = \left\{\!\!\left|\begin{array}{l} Stock, (\text{“}Order\text{”}, 1, p_1, q_1, pending), (\text{“}Order\text{”}, 2, p_2, q_2, pending),\\ ..., (Order, p_k, k, q_k, pending)(\text{“}ordercounter\text{”}, n), (\text{“}getord\text{”} : GET),\\ (\text{“}canc\text{”} : CANC), (\text{“}newq\text{”} : NEWQUANT), (\text{“}send\text{”} : SEND) \end{array}\right|\!\!\right\}$$

$$GET = \left\{\!\!\left|\begin{array}{l} (\uparrow(\text{“}neword\text{”}, p, q),\\ (\text{“}ordercounter\text{”}, n) \end{array}\right|\!\!\right\} \xrightarrow{(nn)\leftarrow f(n)} \left\{\!\!\left|\begin{array}{l} (\text{“}ordercounter\text{”}, nn),\\ (\text{“}Order\text{”}, n, p, q, pending) \end{array}\right|\!\!\right\}$$

**where** $f(n) = (n + 1))$

$$CANC = \left\{\!\!\left|\begin{array}{l} \uparrow(\text{“}cancel\text{”}, id_{order}),\\ (\text{“}Order\text{”}, id_{order}, p, q, pending) \end{array}\right|\!\!\right\} \longrightarrow \{\!\!|\, (\text{“}Order\text{”}, id_{order}, p, q, canceled) \,|\!\!\}$$

$$NEWQUANT = \{\!\!|\, \uparrow(\text{“}newq\text{”}, id_{prod}, q) \,|\!\!\} \longrightarrow \{\!\!|\, (\text{“}newq\text{”}, id_{prod}, q) \,|\!\!\}$$

$$SEND = \{\!\!|\, (\text{“}Order\text{”}, j, p_j, q_j, invoiced) \,|\!\!\} \longrightarrow \{\!\!|\, \uparrow((\text{“}Order\text{”}, j, p_j, q_j, invoiced) \,|\!\!\}$$

*Stock*

$$Stock = \left\{\!\!\left|\begin{array}{l} (\text{“}Product\text{”}, p_1, q_1), (\text{“}Product\text{”}, p_2, q_2),\\ ..., (\text{“}Product\text{”}, p_n, q_n), (\text{“}invoice\text{”} : IN)(\text{“}update\text{”}, UPDATE) \end{array}\right|\!\!\right\}$$

$$IN = \left\{\!\!\left|\begin{array}{l} (\text{“}Product\text{”}, p_i, q_i),\\ \mathbf{ask}(q_j \le q_i)\\ \uparrow(\text{“}Order\text{”}, j, p_i, q_j, pending) \end{array}\right|\!\!\right\} \xrightarrow{(q_m)\leftarrow f(q_j, q_i)} \left\{\!\!\left|\begin{array}{l} (\text{“}Product\text{”}, p_i, q_m),\\ \uparrow(\text{“}Order\text{”}, j, p_i, q_j, invoiced) \end{array}\right|\!\!\right\}$$

**where** $f(q1, q2) = (diff(q1, q2))$

$$UPDATE = \left\{\!\!\left|\begin{array}{l} \uparrow(\text{“}newq\text{”}, id_{prod}, q),\\ (\text{“}Product\text{”}, id_{prod}, qi) \end{array}\right|\!\!\right\} \xrightarrow{(qnew)\leftarrow f(q, qi)} \{\!\!|\, (\text{“}Product\text{”}, id_{prod}, qnew) \,|\!\!\}$$

**where** $f(qplus) = (plus(q1, q2))$

**Table 3.** PoliS specification of Case2

than the one present in the stock?" is no more a problem: the input of new quantity of product from the environment allows an order of a larger quantity of product than the stocked one to be invoiced in a future (i.e. when the stocked quantity is updated).

The other question on Case1 ("if an order asks for a non-stocked product?") still is a problem, in fact an order of a non-stocked product will never be invoiced. The specification describes the input of new quantities but not the input of new products. "It is possible to have input of new products?": some rules could be added th handle these situations.

Then a related question is "if a quantity of a non-stocked product arrives?": should this case be considered as "input of new product" or simply an error of non-stocked product quantity input?".

"If the environment asks to cancel a non-present order?" is an other question risen from the specification: the $CANC$ rule checks if the order is present and then cancel it. If the order is not present the $CANC$ rule is not applied: we could add, if needed, a rule to handle this kind of error signaling it to the environment (as an output).

## 4 The Model Checker for PoliS

Model checking has been initially used for hardware system verification. Recently it has been used also for software systems, but we know few examples in which it has been applied to architectural analysis. The aim of model checking is to find an assignment (*model*) for system variables that satisfies formulae describing some system properties. Given an operational specification of a software system, a model checker builds a model and then makes an exhaustive checking of variable values. This method could seem trivial and inefficient, but it is very powerful for finite systems.

In order to specify the properties to be verified, we introduce a temporal logic. The PoliS Temporal Logic (PTL) is a CTL [7] dialect. The main differences between PTL and CTL depend on the definition of our model, that is based on spaces (multi-sets).

All formulae are evaluated in a context (a space); we also assume that formulae without an explicit context are evaluated in the main space. An atomic proposition *atom* is a tuple; we say that proposition *atom* is true in a context C if it belongs to space denoted by constant C. We have also added the classical logic operators and some temporal operators to improve formulae representation and understanding.

In table 4 we sketch the PTL syntax.

- A *ptf* can be a *temporal*, a *classic*, a parenthesized *ptf*, an *atom*, a *ptf* can be universally or existentially quantified inside *range* over some variables;
- a *context* is a PTL formula that has a pattern like: $ptf \ \in \ C$ (space C), $ptf \ \in \ \star C$ (all C spaces ), $ptf \ \in \ \& C$ (some C spaces), or $ptf \ \in \ \% C$ (exactly one C space), these because in a specification it can be more than one instance of the same space;

| | | |
|---|---|---|
| ptf | ::= | context \| |
| | | temporal \| |
| | | classic \| |
| | | (ptf) \| |
| | | atom \| |
| | | $\forall$ i range (ptf) \| |
| | | $\exists$ i range (ptf) |
| range | ::= | $\epsilon$ \| |
| | | $\in$ [min,max] |
| context | ::= | ptf $\in$ C \| |
| | | ptf $\in$ $\star$C \| |
| | | ptf $\in$ &C \| |
| | | ptf $\in$ %C |
| atom | ::= | tuple |

| | | |
|---|---|---|
| temporal | ::= | $\star\mathbf{X}$ptf \| |
| | | &$\mathbf{X}$ptf \| |
| | | $\star$(ptf $\mathbf{U}$ptf) \| |
| | | &(ptf $\mathbf{U}$ptf) \| |
| | | $\star\Diamond$ptf \| |
| | | &$\Diamond$ptf \| |
| | | $\star\Box$ptf \| |
| | | &$\Box$ptf \| |
| | | ptf $\leadsto$ptf |
| classic | ::= | ptf $\wedge$ptf \| |
| | | ptf $\vee$ptf \| |
| | | $\neg$ptf \| |
| | | ptf $\Rightarrow$ptf |

**Table 4.** PTL syntax

- 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 a 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 at least a path ptf' will be eventually true*";
- a *classic* is a PTL formula with classical logic operators;
- an *atom* is simply a tuple.

The model checker (PoliMC) 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 graph whose nodes are spaces and edges are labeled with tuples produced/consumed in the parent spaces, then it parses PTL formulae and builds syntactic trees, finally it can start to perform model checking. Its algorithm follows the guidelines given in [7].

## 5 Analysis of the Specification

PoliS specification language allows the completely separated specification of the coordination aspects of a system from the computational ones: the specification of functional aspects is limited to functions $f$ in the rules. For instance, the rule $IN$ (Table 3.1) specifies the operation of invoicing an order defining the modification of the tuple $Order$, and the functional operation of calculating the difference between the quantity $q$ of the product in the stock and the required quantity $q1$ (indicated on the top of the arrow of the rule).

The multiple spaces based model encodes in a modular way the different components of the system (e.g. the $Stock$ is a space). If the requirements had said it, a particular space containing all the orders (i.e. an abstraction of a commercial office) would have been defined.

The input from the environment is seen in PoliS as inheritance of tuples by the main space from an hypothetic external space: it is a natural way to encode embedded systems.

The parallel handling of tuples is suitable for this study: it simulates the parallel handling of the orders (or of the updating of the product quantities) by different employees in a company.

It is possible to modify the specification creating different agents (the employees) that handle orders (or updates the quantities). The agents can be modeled in PoliS as spaces where the operations of invoicing orders can take place. In this way the process of invoicing in a company can be specified taking into account the personnel availability and the different roles.

PoliS allows also the splitting of the stock in different sub-stocks containing different kind of products (one single product per stock or similar products in the same stock): multiple spaces would be added and rules would help in the search of the right stock for an order).

The parser helps in finding syntax errors and wrong constructs or sentences. After having checked the specification against these errors the model checker tries to build the graph of the possible evolutions of the system. Obviously, while using a model checker, you have to limit the scope of the variables to a finite set of values in order to generate finite graphs [7].

We have tried to verify some liveness properties on the two versions of the Invoicing System. The model checker helped us in detecting some errors in our reasoning.

The first error we found was in the specification. The model checker revealed some difficulties in building the graph representing the system evolutions: it tried to generate negative numbers for the quantities $q$ of the products. The model checker helped us in debugging the specification and we found out that we did not put the constrain $\mathbf{ask}(q_j \leq q_i)$ on the specification of the rule $IN$ (invoice) (Table 3.1). This condition checks that the quantity of product in the stock is larger than the ordered one: only under this condition an order can be invoiced.

After having successfully built the graph of the two specifications we tried to verify the property (1) on the first system:

$$\begin{aligned} \forall p, q, id(\text{``}Order\text{''}, id, p, q, pending) \in StartContext \leadsto \\ (\text{``}Order\text{''}, id, p, q, invoiced) \in StartContext \end{aligned} \tag{1}$$

That is, if an order has to be invoiced, it will eventually be invoiced ($\leadsto$ stands for "leads to"). However, the model checker verified that property (1) is false. In fact we have to ensure that the product requested ($p$) is present in the stock at least in quantity $q$:

$$\begin{aligned} \forall id, p, q, \exists k((\text{“}Order\text{”}, id, p, q, pending) \in StartContext \wedge \\ (\text{“}Product\text{”}, p, q+k) \in Stock) \leadsto \\ (\text{“}Order\text{”}, id, p, q, invoiced) \in StartContext \end{aligned} \tag{2}$$

By the way, that condition is not enough yet: the model checker still found out that it is false. The reason is that there can be other orders for the same product that can be invoiced before the order $id$, and that do not leave in the stock enough quantity of product $p$ to let the order $id$ be invoiced.

Then, we rewrote the (2) as:

$$\begin{aligned} \forall id, id', p, q, q' \exists k((\text{“}Order\text{”}, id, p, q, pending) \in StartContext \\ \wedge (\text{“}Product\text{”}, p, q+k) \in Stock \wedge id \neq id' \wedge \\ (\neg(\text{“}Order\text{”}, id', p, q', pending) \in StartContext)) \leadsto \\ (\text{“}Order\text{”}, id, p, q, invoiced) \in StartContext \end{aligned} \tag{3}$$

Then, the model checker verified (3).

We then tried to verify a property on the second system (Table 3). In this system input from the environment is accepted, so we had to take into account the updating of the products stocked (see Table 3: rule $NEWQUANT$):

$$\begin{aligned} \forall id, id', p, q, q' \exists k((\text{“}Order\text{”}, id, p, q, pending) \in StartContext \wedge \\ (\text{“}Product\text{”}, p, q+k) \in Stock \wedge id \neq id' \wedge \\ \star(\neg((\text{“}Order\text{”}, id', p, q', pending) \in StartContext)\mathbf{U} \\ \neg((\text{“}Order\text{”}, id, p, q, pending) \in StartContext)) \\ \leadsto((\text{“}Order\text{”}, id, p, q, invoiced) \in StartContext \\ \vee(\text{“}Order\text{”}, id, p, q, canceled) \in StartContext) \end{aligned} \tag{4}$$

That is, if an order of product $p$ is to be invoiced, and there is enough quantity of product in the stock, and a new order for the same product is not accepted until (**U** stands for until) the order $id$ is to be invoiced, then, eventually the order will be invoiced or canceled.

In a previous proof session we tried to verify that, at these conditions, the order will eventually be invoiced. However the model checker verified that this property was false: we had to take into account also order cancellations.

## 6 Conclusions

We have introduced a coordination language (PoliS) and used it to specify the Invoicing System. Due to the simplicity of the case study, the specification is quite simple and immediate.

PoliS offers a different approach with respect to languages like Z [11] or VDM. These languages are property oriented and have a declarative approach. Z is a very expressive notation and strongly typed: type checking helps in dealing with large specification documents where type errors are more frequent. PoliS is a type-less language, the parser and the model checker detect syntax errors and verify temporal properties, however no type checking can be performed on the specification. On the other hand Z language hardly specifies dynamics of a system: many enhancements and integrations with other notations have been tried in order to allow dynamic aspects to be specified [8, 10, 1, 4]. PoliS emphasizes the behavioral aspects of a system, highlighting the rules configuration.

The PoliS operational model helped us in understanding the dynamics of the Invoicing System: Case 1 consists of a simple specification containing a single rule ($IN$) that helps in invoicing the orders decreasing the quantities of products in the stock. In Case 2 the environment has a role and the input of new orders and new quantities of products are considered. The specification is more complex than in Case 1: there are new rules that handle the input and the output with the environment.

Some questions have risen from both the specifications. from the Case 1 specification: "what happens if an order asks for a non-stocked product? Or for a quantity of product larger than the one stocked?". From Case 2: "What happens if a quantity of non-stocked product arrives? Or if the environment asks to cancel a non-present order?". The model checking technique helped us in inferring some properties on our specifications and in increasing confidence in the dynamics: trying to proving properties, helps in finding comprehension errors and features of the model (see sect 5 for details).

As future work we are devising an enhancement of the notation in order to deal with typed data. We are also studying how the PoliS language could be used to specify mobility based systems, considering several different paradigms of mobility. We are investigating if the PoliS model could fit for analysis of security issues and if the model checker can help in verifying security properties on a system.

**Acknowledgments.** The authors would especially like to thank Francesco Franzè for his helpful comments and suggestions.

## References


1. M. Benjamin. A Message Passing System. An example of combining Z and CSP. In J. Nicholls, editor, *Proc. 4th Z Users Workshop (ZUM)*, Workshops in Computing, pages 221–228, Oxford, 1989. Springer-Verlag, Berlin.
2. N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.
3. P. Ciancarini. PoliS: a Programming Model for Multiple Tuple Spaces. In C. Ghezzi and G. Roman, editors, *Proc. 6th ACM/IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pages 44–51, Como, Italy, October 1991. IEEE Computer Society Press.

4. P. Ciancarini, S. Cimato, and C. Mascolo. Engineering Formal Requirements: an Analysis and Testing Method for Z Documents. *Annals of Software Engineering*, 3:189–220, 1997.
5. P. Ciancarini, F. Franzé, and C. Mascolo. A Coordination Model to Specify Systems including Mobile Agents. In *Proc. 9th ACM-IEEE Int. Workshop on Software Specification and Design (IWSSD)*, page (to appear), Japan, 1998.
6. P. Ciancarini, M. Mazza, and L. Pazzaglia. A Logic for a Coordination Model with Multiple Spaces. *Science of Computer Programming*, (to appear), 1997.
7. E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
8. A. Evans. Specifying and Verifying Concurrent Systems Using Z. In M. Bertran, T. Denvir, and M. Naftalin, editors, *Proc. FME'94 Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 366–380. Springer-Verlag, Berlin, 1994.
9. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
10. L. Lamport. TLZ. In J. Bowen and J. Hall, editors, *Proc. 8th Z Users Workshop (ZUM)*, Workshops in Computing, pages 267–268, Cambridge, UK, 1994. Springer-Verlag, Berlin.
11. J. Spivey. *The Z Notation. A Reference Manual.* Prentice-Hall, 2 edition, 1992.