Coordinating Rule-Based Software Processes with ESP

Paolo Ciancarini

Technical Report UBLCS-93-8
April 1993
The University of Bologna Laboratory for Computer Science Research Technical Reports are available via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS in compressed PostScript format. Abstracts are available from the same host in the directory /pub/TR/UBLCS/ABSTRACTS in plain text format. All local authors can be reached via e-mail at the address last-name@cs.unibo.it.

UBLCS Technical Report Series

93-7 Distributed Programming with Logic Tuple Spaces, by P. Ciancarini, April 1993.
93-8 Coordinating Rule-Based Software Processes with ESP, by P. Ciancarini, April 1993.
Coordinating Rule-Based Software Processes with ESP

Paolo Ciancarini

Technical Report UBLCS-93-8
April 1993

Abstract

ESP is a language for modeling rule-based software processes that take place in a distributed software development environment. It is based on PoliS, an abstract coordination model that relies on Multiple Tuple Spaces, i.e., collections of tuples a la Linda. PoliS extends Linda aiming at the specification and coordination of logically distributed systems. ESP (Extended Shared Prolog) combines the PoliS mechanisms to deal with concurrency and distribution, with the logic programming language Prolog, to deal with rules and deduction. Such a combination of a coordination model and a logic language provides a powerful framework in which experiments about rule-based software process programming can be performed and evaluated.

1. e-mail: cianca@cs.unibo.it
1 Introduction

The theme of software process modeling has been recently addressed by several specific conferences, see for instance [1,21,25]; for a survey, see [19]. This interest stems from the fact that activities involved in software development are so complicate, expensive and error-prone that it seems necessary to completely specify the development process, in order to gain control of it and improve its quality. A software process program should formally define the activities that are carried out in the development of a software project, providing guidance to the agents involved and controlling the overall evolution of the project status [33].

In this paper we devote our attention to the fact that a software process takes place necessarily within a software development environment. The environment supports and coordinates the interactions of the project members, allows the use of programming tools, and monitors the evolution of the project documents.

We will show that the environment can be designed with the goal of explicitly modeling and coordinating the process itself. In fact, we suggest that a natural way to model a software process consists of introducing a suitable abstract machine defining the environment in which the software process is developed and a suitable programming language able to control the process evolution. Our guiding principle is that, in order to specify a software process, we must clarify the coordination model [13] that has to be used by all the agents, i.e., we must make explicit the communication mechanisms that are at the basis of the interaction protocols used by the participants of a software project. We identify the coordination model with the abstract architecture of the environment that supports the execution of the process itself.

The main contribution of this paper is the definition of an abstract paradigm for modeling coordination of activities that take place inside a software development environment. The paradigm is called PoliSpaces (or PoliS), because it is based on Multiple Tuple Spaces [23]. PoliS allows a software process designer to model a software process that takes place inside a multi-user, distributed development environment. In modeling the activities that form the software process our approach is similar to that used in MARVEL [8,9] and Merlin [24,34]: each activity that takes place in the environment, as prescribed by the process, is governed by rules. With respect to these systems, we put emphasis on both the coordination mechanisms that allow us to combine the rules that govern the software process, and on the declarative language that allows to express the rules themselves.

Our proposal includes three steps. First, we define an abstract coordination model. A coordination model is a set of abstract mechanisms for expressing and controlling sets of sequential activities [13]. In our approach, each activity can be expressed in any sequential language; its interactions with respect to other activities is defined using the coordination model. Our coordination model – PoliS – is based on Multiple Tuple Spaces, i.e., collections of tuples manipulated with the operations offered by the Linda coordination language [22].

Second, in order to be able to express rule-based software processes, we use logic programming to take advantage of its inferential and relational capabilities. We introduced elsewhere the parallel logic language Shared Prolog [11], that combines Prolog with a Linda-like tuple space. Here, we use an extension of Shared Prolog that is better suited for distributed programming. In fact, Extended Shared Prolog (ESP for short) is a logic language that combines the features of PoliS for distributed programming with the declarative and inferential capabilities of Prolog.

Finally, we show how ESP can been used to model a software process that takes place inside a distributed software development environment. The idea is that the software process is
explicitly defined by rules that form coordination protocols; these govern the activities inside a distributed environment. The rules specify goals, duties, and constraints that the agents involved in the software process have to fulfill. We suggest that it is the environment that imposes constraints and supports coordination protocols among the users, and that controlling the environment means governing the software process.

The concepts of distributed development environment and rule-based software process as well as their interplay are naturally modeled in PoliS and easily implemented in ESP. In fact, we show how ESP can be used to design simple programming environments corresponding to simple software processes.

The power of ESP can be tested using Oikos, a rule-based development environment able to execute software process models written in ESP [2,4]. Oikos offers a number of standard services giving some basic facilities, like access to databases and private workspaces, activation of shells, etc. The ESP language can be used to implement a desired software process.

The paper is organized as follows: Section 2 introduces PoliS as a model for distributed programming. Section 3 describes Extended Shared Prolog, a programming notation based on PoliS. Section 4 shows how ESP can be used in the design of simple software development environments and processes. Section 5 summarizes the main design principles underlying Oikos. Finally, Section 6 contains some comparisons with related work.

2 PoliS: A Model for Coordination

Intuitively, a multiuser software development environment can be seen like an abstract town where there are many places. Several agents are distributed in the places; they cooperate producing documents that either remain in the same place or are sent to another place. In the town many activities are performed simultaneously, mostly independently; however, all of them are ruled by laws and policies that are constraints that are either physical (e.g., the available resources, like space and time) or abstract (e.g., a set of laws that prohibit some behavior). We suggest that a software process designer that has to set up a multiuser software development environment should design it as a polispace, i.e., as a distributed system that is a collection of named tuple spaces.

The concepts of tuple and Tuple Space are borrowed from Linda; PoliS extends Linda allowing for a multiplicity of Tuple Spaces. More precisely, in PoliS three concepts are important: tuples, agents, and tuple spaces.

2.1 Tuples

Tuples are sequences of variables and values. As in Linda, values depend on the chosen sequential programming language used for programming agents. However, in PoliS a number of basic value types, as well as lists of these values, are allowed.

Example:
For instance, these are tuples:

```
(do,edit(program1))
('Paolo', programmer, Program)
(program1, [module1, module2, module3])
```

UBLCS-93-8
Some useful conventions: a tuple is always parenthesized; identifiers that are not quoted and begin with capital letters denote variables; other identifiers denote atoms; square brackets denote lists. Tuples denote themselves; they are simply data objects that exist in a tuple space, produced by an agent and possibly in the future consumed by an agent. The scope of variables inside tuples spans only for the tuple to which they belong. This means that each tuple inside a Tuple Space is completely independent from other tuples.

Each tuple is produced by some agent in some tuple space, and it remains there until some agent consumes it. Access to a tuple is associative, i.e., it is done “by contents”. The particular access mechanism chosen is a degree of freedom: e.g., PoliS can accommodate either a mechanism based on typed pattern matching, as in Linda, or a mechanism based on unification, as in a logic language.

2.2 Agents

Abstractly, agents are execution threads, i.e., an agent is a process executing some program. As in Linda, an agent is represented by a so called active tuple. An important difference between Linda and PoliS is that in the latter an agent executes a program contained in a tuple of a special kind, called program-tuple. The program is written in a sequential programming language, enriched with a set of tuple space operations.

2.2.1 Tuple Space Operations

An agent can use the following tuple space operations for interacting with the Tuple Space it lives in:

- associative test of a tuple contained in the agent’s Tuple Space;
- associative consumption of a tuple from the agent’s Tuple Space;
- asynchronous creation of a Tuple Space or a tuple.

Most of these operations are borrowed from Linda. There are two different “flavors” for the test and consumption operations—they can be either blocking or not-blocking, and two not blocking operators—one for the creation of a tuple, and one for the creation of a new tuple space.

The blocking test operation in Linda is written \texttt{read(Tuple},\texttt{schemata)}\texttt{), the non-blocking test operation is written \texttt{readp(Tuple},\texttt{schemata)}\texttt{), the blocking consumption operation in is written \texttt{in(Tuple},\texttt{schemata)}\texttt{, and the non-blocking consumption operation is written \texttt{inp(Tuple},\texttt{schemata)}\texttt{. A Tuple},\texttt{schemata} is a tuple containing formal arguments, i.e., wild cards that match any actual argument inside a tuple contained in the Tuple Space.

Actually \texttt{readp} and \texttt{inp} are not considered part of the “standard” definition of Linda. We maintain them in PoliS, because agents need maximum generality in accessing tuple spaces, and the success/failure semantics of \texttt{readp/ inp} will be useful when using logic-based control structures.

An agent can output either a tuple or a Tuple Space (i.e., it creates a new tuple space). The former operation is written \texttt{out(Tuple)} in case of local writing, \texttt{out(Tuple)@TupleSpace} in case of outside writing; the latter one is written \texttt{tsc(TupleSpace)}.

The target of output operations is always a Tuple Space, specified using its name. If no name is specified, by default the Tuple Space of the agent is used. What happens if an \texttt{out} operation targets an external Tuple Space that does not exists? PoliS is consistent with the Linda semantics: \texttt{out} is a non-blocking operation (i.e., the agent that issue it does not wait for any result or error code), that never fails. Communications among tuple spaces are supported.
by a meta Tuple Space where undelivered tuples remains deposited; whenever a tuple space comes into existence, the undelivered tuples “pop up” in the tuple space.

If an agent needs to be certain that a message has gone its way, it must explicitly use a protocol. For instance, it could send the message and an agent that sends back an acknowledgment.

Finally, we note that an agent can test or consume tuples representing other agents. This is useful to build agents that schedule agents. Tuple spaces cannot be operands either for testing or for consuming, because the obvious semantics for such operations (copy a whole tuple space, delete a whole tuple space) should necessarily manipulate the global state of a Tuple Space, in contrast with the asynchronous nature of its internal activities.

2.2.2 Program-tuples

A Linda program can invoke arbitrary sequences of Tuple Space operations. PoliS agents follow instead a fixed protocol for invoking tuple operations. Syntactically, the program that an agent executes is written inside a program-tuple.

(Heading : (Test; Consume; LocEval; Out))

The Heading is a Tuple Schemata. Test, Consume, and Out are sequences of tuple operations, whereas LocEval is a sequential computation that has no side effect in the Tuple Space to which the agent belongs. An agent is activated when the Tuple Space contains both a program-tuple and a normal tuple matching the heading in the program-tuple; the latter is consumed.

The second component of a program-tuple is also called a pattern. Executing a pattern, an agent will do the following actions:

- it reads something from its tuple space using any number of test operations; actually the test operation in PoliS has a broader semantics than read in Linda: a number of predefined tests on the Tuple Space are allowed, depending on the chosen type system for tuple arguments. Some useful predefined tests are: relational (binary) predicates, a var/1 predicate to check if an argument inside a tuple is a variable, and a thisplace/1 predicate returning the name of the tuple space in which an agent is located.
- it deletes some tuples using any number of consumption operations.
- it executes a “local evaluation” that has no effect on the Tuple Space insofar as no operations on the Tuple Space itself are allowed; this local computation is expressed in a sequential programming language;
- it outputs the results in a number of tuple spaces it “knows”; these outputs can consist of tuples or tuple spaces;
- at the end of the sequence the agent “dies”, terminating its thread of evaluation; however, we can specify an ever-lasting agent by inserting among its outputs the creation of a copy of itself.

What is the computing model underlying the computation of an agent? The idea is that agents are stateless and reactive, i.e., they compute when a “molecule” of tuples can be built inside the Tuple Space. A molecule is composed of a program-tuple, a normal tuple matching the first component of a program-tuple, and all the tuples to be consumed as specified by the consume section in the program-tuple. The agent “reacts” to its environment, “burning” the molecule, and creates new entities as specified in the create section. This semantics is studied in [17].
Example:
An ever-lasting chemical reaction can be seen in this Tuple Space containing two table tennis players:

\{(a, b) (ping) ((a) : (in(ping); out(pong); out(a))) ((b) : (in(pong); out(ping); out(b)))\}

Agent a begins building a molecule with tuple (ping); it consumes that tuple and produces tuple (pong) and a copy of itself (a). Then it is the turn of agent b that reacts and consumes tuple (pong) producing tuple (ping) and a copy of itself (b), and so on, either forever or until something from outside comes to alter this “chemical solution”. For instance, suppose that an external agent sends a new (ping) tuple in the above Tuple Space; as soon as the new tuple is noticed by agent a, the two agents are no longer serialized.

We remark that program-tuples are tuples, i.e., they can be freely added or removed from a Tuple Space, thus providing an indirect means of manipulating agents.

Example (remote evaluation):
If an agent a in tuple space source needs to be sure that some output reached a tuple space dest, it can use the following protocol:

\{\begin{align*}
\text{((a): (out(msg)@dest; out(b)@dest; out((b):(read(msg); source.out(ack_msg))@dest))}
\end{align*}\}

i.e., the agent sends the message msg and a “representative” b, that in the destination Tuple Space tests for the presence of the message and sends back an ack message.

2.2.3 Non deterministic evaluation
The agent semantics outlined above is very simple, because it can be defined as an input-output relationship independent from the global environment, i.e., the tuple space. An agent is “reactive”, i.e., it reacts to its tuple space in the sense that the set of testing and consuming operations contained in its program can be thought of as a firing pattern that activates a computation (the local evaluation) that finally modifies the tuple space itself.

Actually this semantics is even too simple, since we have problems in specifying some interesting situations. For instance, situations where it is impossible to give to an agent all the input it needs at the start of its computation increase the verbosity of programs. Moreover, mutual exclusion relationships are very boring to specify, because they must be handled explicitly defining a “semaphore” tuple for each set of agents in mutual exclusion. We can define non deterministic agents simply by putting in the Tuple Space a set of program-tuples whose headings match.

Example:
The program-tuples describing the behavior of an agent a that can either edit, or compile, or execute a file, are the following:

\{\begin{align*}
\text{((a) : (in(edit(File)); do_edit(File, New); out(New))}
\text{((a) : (in(compile(File)); do_compile(File, Object); out(Object))}
\text{((a) : (in(exec(File)); do_exec(File, Results); out(Results))}
\end{align*}\}

Such a set can be compared to the Hoare’s CSP guarded command, or to an object defined by a set of methods, or to a logic predicate defined by a set of clauses. This is yet another
extension of PoliS with respect to Linda, that does not offer any non-deterministic operation. Non determinism is useful to specify complex tasks that are partitioned in several alternative subtasks.

2.3 Tuple Spaces
A Tuple Space is a named multiset of passive or active tuples. Syntactically, we will write tuple spaces as sequences of tuples between curly brackets.

Example:
To describe a tuple space named workspace containing three tuples, we write what follows:

```plaintext
workspace{ (file,1) (file,2) (do_edit,file,1) }
```

Another difference with respect to Linda is that in PoliS each Tuple Space has two attributes: a name, and a set of invariants.

2.3.1 Names of Tuple Spaces
An important feature of PoliS tuple spaces is that they have a name. Agents can send tuples outside their own tuple space using the name of another tuple space. Tuple space names can be freely passed as arguments of tuples, so that it is possible to dynamically build complex communication flows.

The name system of tuple spaces is a degree of freedom in PoliS, just like the choice of the sequential language for local computations, and the type system chosen for tuples. This means that PoliS does not offer any specific name system. A programming language based on PoliS has to choose its own name system for tuple spaces. If the names of the tuple spaces form a flat structure (e.g., they are pointers), all tuple spaces are independent and scoping rules are very simple: tuple space names form a global environment. If the names are structured, the tuple spaces are structured as well: e.g., if the names are paths in Unix-like style, the tuple spaces form a tree similar to the Unix file-system (this name system is used in Oikos; see 5).

2.3.2 Invariants
Whereas agents are ephemeral and stateless, a Tuple Space can be seen as an object that is persistent and has a state [41]. A Tuple Space is not a passive entity, a mere repository of tuples or a channel for messages. There is a way of controlling the activities that take place inside a Tuple Space. In fact, for each Tuple Space we can define one or more invariants, i.e., constraints that must hold for all the Tuple Space life span. Whenever an invariant is violated, the Tuple Space stops all activities and terminates. A “tuple space garbage collector” could now claim all the resources allocated to the Tuple Space.

Invariants are defined inside special program-tuples where the heading is substituted by the keyword invariant.

```plaintext
(invariant : (Test; Loc_Eval; Out))
```

The test section defines a condition on the Tuple Space that, when verified, terminates the Tuple Space itself. The Loc_Eval and the Out sections can be used to compute and communicate any results to other tuple spaces.
Example:
The following is an invariant:

\[
\text{invariant: (read(validated\text{(design)}); out(start\text{(coding}@\text{manager})))}
\]

when tuple \text{validated\text{(design)}} is produced, the tuple space terminates, communicating
tuple \text{start\text{(coding)}} to tuple space named \text{manager}.

The invariant concept is not present in Linda. In PoliS it is useful to specify the intended
semantics of a Tuple Space: when the condition specified by the invariant is verified, a “result”
has been obtained; it can then be passed to some other Tuple Space.

ESP: A Parallel Language that combines PoliS and Prolog

PoliS could accommodate any sequential language for expressing local computations inside
agents. For instance, C-Linda can be considered an instance of PoliS where the sequential
language is C, tuples are built using the C data types and a unique tuple space is allowed. In
Linda there are no constraints on the use of tuple operations (i.e., we can use in a program
\text{in}, \text{read}, and \text{out} in any order). However, we note that any sequence of Linda operations
can be split in a number of subsequences such that each subsequence begins with \text{read/in}
operations and terminates with \text{out} operations.

In order to explore the effective usefulness of PoliS in the design of rule-based software
processes and environments, we have introduced ESP, a programming language that combines
PoliS with Prolog. ESP is actually an extension of the parallel logic language Shared Prolog
[11], that is a logic language that uses the blackboard model [31] for organizing interprocess
communication. With some approximations, Shared Prolog can be considered a member of
the Linda family of physically distributed programming languages [22]. Shared Prolog gains
in expressive power with respect to Linda by exploiting unification and backtracking when an
agent accesses the tuple space (Linda uses pattern matching, and no backtracking is allowed).
ESP generalizes Shared Prolog with multiple tuple spaces.

3.1 Theories

An ESP program is composed of a set of modules called theories. Each \textit{theory} has the following
syntactic structure:

\[
\text{theory name}(V_1, \ldots, V_n):- \quad \text{\% theory heading}
\]

\[
\text{eval \text{rule}_1\# \ldots \# \text{rule}_k} \quad \text{\% theory interface}
\]

\[
\text{with Prolog\_program} \quad \text{\% theory implementation.}
\]

A \textit{theory} is a module identified by a name and zero or more arguments \(V_i\) that are logic
variables that scope over the rules. The theory interface follows the keyword \textit{eval} and it
includes a number of rules (also called “patterns”), separated by the symbol \#; the theory
implementation is the Prolog program that follows the keyword \textit{with} (the reader not familiar
with Prolog will find a good introduction to it and its terminology in [37]).

Example:
The following is a theory that includes only one rule. It defines an agent that compiles a file
when it finds in its tuple space the tuple \text{compile\text{(File)}}; the result of the agent activation

is that either a success tuple is sent to blackboard test or, if the compilation fails, a failure message to blackboard user.

theory compiler:-
  eval
  \{compile(File)\} % consume
  \rightarrow
  call_compiler(File), % Prolog goal
  \{compiled(File)test\} % Success_Out
  fail \{report_error(File)user\} % Failure_Out
  with
  call_compiler(File):- ... % Prolog program
  % special predicate that invokes a compiler
  % it fails if compilation fails

An ESP theory including a set of rules is similar to a CSP guarded command: when one of the rules can fire, the agent executing the theory commits to that rule; if more than one rule can fire, one is chosen non deterministically.

ESP rules are clauses that include: a Test section, that formally is a goal to be evaluated with respect the current contents of the Tuple Space; a Consume section, that describes a multiset of tuples to be consumed; a loc evalu section, that is a Prolog goal to be evaluated with respect to the Prolog program; and finally some out operations. To make explicit the fact that they have side effects on the Tuple Space, consumption and creation operations are put between curly brackets.

Test \{ Consume \} \rightarrow Goal \{ Success_Out \} fail \{ Failure_Out \}

The combination of Test and Consume sections is a guard: when such a guard is satisfied, i.e., if the test is satisfied and consume operations can be completed, the rule is fired, that means that the tuples to be consumed are really deleted and the Prolog goal is evaluated. To deal with the possibility of a failure of such a Prolog goal, creation operations are partitioned in two sets separated by the keyword fail: if the goal evaluation succeeds the Success_Out is produced, else the Failure_Out is produced.

Example:
In the following rule, the first line is the Test section; the second line is the Consume section. This rule states: “if the Tuple Spaces contains tuple file(F) and does not contain tuple reserved(F,\_), and tuple check_out(P,F) can be found and deleted, then produce tuple reserved(F,by(P)) in this tuple space and tuple file(F) in the tuple space named by the value bound to P”.

file(F), not reserved(F,\_)
\{check\_out(P,F)\}
\rightarrow
\{reserved(F,by(P)), file(F)\_P\}

Communications with agents in other tuple spaces are denoted by simply adding the address of the destination tuple space to the tuple to send. If the destination tuple space has not yet been created, the tuple simply waits for such a creation; this consistently extends the Linda semantics of out operations to the PoliS framework.
Finally, a few remarks on the Prolog program that follows the keyword with. All Prolog built-ins can be used, except for predicates that alter the program, like assert and retract. This guarantees that agents are stateless, as required by PoliS semantics. Any state information must be stored in the tuple space.

3.2 Logic tuple spaces
An ESP tuple space, also called a blackboard, is a named multiset of logic tuples; a logic tuple is simply a Prolog term.

Blackboards can be dynamically created by agents simply using an activation goal of the form \texttt{tsc(name\{contents\})}. An activation goal specifies the name of the tuple space and possibly its initial contents.

**Example:**
A goal that creates three tuple spaces; the tuple space \texttt{bb_of_PM} initially contains the tuple \texttt{develop, change, and test(Unit)}.

\[
\texttt{?- \{tsc(bb_of_COD), tsc(bb_of_PM\{develop\_change\_and\_test(Unit)\}), tsc(bb_of_PT)\}}
\]

In general, the execution of an ESP program builds a set of tuple spaces. The set grows when a tuple space is created, and shrinks when a tuple space terminates, i.e., when an invariant fires.

**Example:**
A goal that associates two invariants with the tuple space \texttt{bb_of_PM}. The tuple space terminates itself when either the tuple \texttt{end\_work(Id)} or the tuple \texttt{abort\_work(Id)} it is found in its contents.

\[
\texttt{?- \{invariant(end\_work(Id))@bb\_of\_PT, invariant(abort\_work(Id))@bb\_of\_PT\}}
\]

3.3 Evaluation of logic agents
An ESP Tuple Space can be seen as a programming construct that encapsulates a parallel evaluation of logic agents. Logic agents are represented by active tuples; a tuple is active if it matches the heading of a theory. An agent executes rules that read or delete tuples from its tuple space; the result of the evaluation of a rule normally consists of the creation of some tuples in some tuple space.

A notable feature of ESP is that control flow of \texttt{test} and \texttt{consume} operations is ruled by backtracking. Each \texttt{test} or \texttt{consume} operation either is successful or fails; a failure activates backtracking to the preceding operation.

**Example:**
Suppose we have the following subset of a Tuple Space:

\[
\{\ldots, deliver(Amy), spec(1), spec(2), impl(2), \ldots\}
\]
If an agent commits to the rule: spec(X), \{deliver(X), impl(X)\} → \{mod(X)\}
then tuple deliver(X) is consumed with X = 2. The resulting tuple space subset is the following:

\{
..., \text{spec}(1), \text{spec}(2), \text{mod}(2), ...
\}

A formal semantics for this behavior is given in [11].

3.4 ESP implementation

ESP is a language that evolved as an extension of Shared Prolog. There are currently two distributed implementations, one stand alone [12] and one based on Linda [18]. The first one is implemented part in C and part in Prolog. The C layer implements a socket-based Meta Tuple Space that supports a number of communication primitives usable by Prolog processes. Each Prolog process can handle either several blackboards or several agents. In the Linda based implementation, the Meta Tuple Space is a C-Linda program that (remotely) forks one “worker” for each new logical tuple space. Such a representative communicates via sockets with a Prolog process that handles one logic tuple space. For other details concerning the ESP implementation, see the references.

3.4.1 Interaction with the user

ESP can be used for parallel programming just like Linda: a typical parallel application is a program that needs massive computation power to do a search inside a number of distributed knowledge bases. Linda is of course much more efficient, but ESP can be more easily used for symbolic applications. A good example is a program that coordinates several agents cooperating to solve a code according to the rules of Mastermind [14]. Our benchmarks show that the current ESP distributed implementation is at least as efficient as some parallel implementations of production systems [18].

However, the real use intended for ESP is in the design of programs that coordinate and interact with a number of users. For instance, we have developed a program that coordinates four players that intend to play bridge over a network. Each player is connected to a different workstation. The program distributes the cards to the players, handles each trick and keeps the score. Other programs in this class that have been developed are a simple distributed agenda to coordinate committee meetings, an e-mail system, and a financial simulation in which some users buy and sell stocks in a number of simulated stock markets. These distributed applications make large use of the interactive capabilities of Prolog: it is possible to launch, even remotely, special logic tuple spaces that directly communicate with a terminal. These are called shells; a user then can produce or consume tuples within the shell he is connected to.

In principle, through his blackboard a user can access the whole universe of blackboards, sending message tuples or even active tuples that open new shells on these blackboards. In practice, his interaction with the blackboard system can be constrained by the rules that define the shell.

Example:

The following theory defines a shell that enables a user to put commands in a tuple space if the tuple space contains the tuple access_granted(User) and the commands are syntactically correct:
4 Process Programming with ESP

![Diagram of Polispace]

**Figure 1. A polispace coordinating a simple programming environment**

```prolog
theory simple_shell(User):-
  eval
    access_granted(User)
    ->
    read_command(Command)
    {tuple(Command), simple_shell(User)}
    fail {simple_shell(User)}
with
  read_command(Command):-
    % check legality of Command; uses Prolog I/O predicates
```

This theory shows how user inputs can enter in a polispace. The I/O model used in ESP is based on Prolog I/O built-in predicates.

4 Process Programming with ESP

The activity of programming with a Linda-based logic language has been explored elsewhere [16,17,38]. In this section we will show how ESP can be used as a language for modeling simple software processes and the related development environments.

4.1 A Tiny Programming Environment

A very simple software process takes place in a programming environment including an editor and a compiler. We specify a naive software development process that consists of editing a file, then compiling it as soon as the editing by a programmer is terminated. If the compilation gives no errors, the object program has to be invoked and executed using some test data. Fig.1 depicts such an environment as a polispace.

In order to build the ESP program that implements such a polispace we need three theories: one for an editing agent, (actually a shell for a user), one for a compiler agent, and one for an executing agent. These theories use predicates that are called *envelopes* in [26], because they encapsulate external software tools. Envelopes are useful to introduce non-declarative
operators inside a declarative framework, because they are able to call standard Unix tools via some system predicates that return a logic result \(i.e.,\) either success or failure.

```prolog
theory editor(Editor, Tty):-
  eval
  thisplace(User)
  \{do\_edit(File)\}
  \rightarrow
  call\_editor(Editor, Tty, File),
  \{compile(User, File) \& compile\}
with
  call\_editor(E, T, F):- ...  % invoke envelope for editor E
theory compiler:-
  eval
  \{compile(User, File)\}
  \rightarrow
  call\_compiler(File),
  \{compiled(File) \& test\}
  fail \{do\_edit(File) \& User\}
with
  call\_compiler(File):- ...  % invoke envelope for cc
  \% fails if compilations fails for errors
theory tester(Datafile):-
  eval
  test\_data(Datafile) \{compiled(File)\}
  \rightarrow
  do\_run(File, Datafile, Result),
  \{tested(File, Result)\}
  fail \{test\_error(File)\}
with
  do\_run(File, Datafile, Result) :- ...  % invoke Prolog-Unix envelope for object File
```

This minimal programming environment enforces a simple edit-compile-test programming model. The user interacts only with his tuple space; actions in other tuple spaces are coordinated by the ESP program.

### 4.2 A Multiuser Environment enforcing an Access Protocol

A software project is composed of a set of modules developed by a team of programmers. A software process designer has to design a software development environment that enforces the following policy. The updated public version of the whole project is stored within a main database. Users can access the main database in read mode only. Coordination is based on a reserve/deposit access protocol for the main database which guarantees mutual exclusion and consistency: the main database always contains a consistent and updated version of the project. To modify the contents of a module, a user must reserve the module to gain write access. Obviously, at any time a module can be reserved just once. A reserved module is copied into the user database, where the user can modify it at will. While a reserved module is being edited in a user database, other users can access in read mode the old public version
of the module stored in the main database. When the changes to the module are completed and tested, the user will deposit the new version back into the main database. The updated version is then readily accessible by all other users.

A polispace realizing such an environment is showed in Fig.2.

The code of the theory user\_db\_manager, that handles the user’s requests in a user database, is the following:

```
theory user\_db\_manager(Dbmain):-
  eval
    thisplace(Udb)
    \{check\_in(File, Dbmain)\}
    \rightarrow
    \{check\_in(File,Udb)\#Dbmain\}
#
    thisplace(Udb)
    \{check\_out(File,Dbmain)\}
    \rightarrow
    \{check\_out(Udb,File)\#Dbmain\}
```

Since a user can issue in his blackboard either a checkin or a checkout command, there is a rule for each command, that simply transmits a tuple to the main database blackboard. This tuple space contains two agents that guarantee the consistency of the main database: the checkin\_manager and the checkout\_manager, that execute the following theories:

```
theory checkout\_manager:-
  eval
    file(F), not reserved(F,\_)
    \{check\_out(F,F)\}  \% read request for file F by programmer F
```
The first rule reserves the requested file; the other rules handle different error situations.

theory checkin_manager:-
eval
not file(F)
{check_in(F,P)} % create request for file F by programmer P
→ {file(F)}, {created(F)0P} % request granted
#
file(F), reserved(F,by(F,P))
{check_in(F,P)} % modify request for file F by programmer P
→ call_file_server(modify, F, P)
{modified(F)0P} % request granted
#
file(F), not reserved(F)
{check_in(F,P)} % cannot create: a file F does exist
→ {error(file_exists(F))0P} % request not granted
#
file(F), reserved(F,by(F,OP)), P ≠ OP
{check_in(F,P)}
→ % cannot modify: file F is reserved
{error(is_locked(F,by(0P)))0P} % request not granted
with
call_file_server(modify, F, P):- ...
% asks file system to get new file F from P's workspace

The first rule create a new file; the second rule modifies a file, invoking by an envelope an operating system service; the other rules handle different error situations.

4.3 A Software Process Program Fragment

In October 1990, the organizers of the Sixth International Software Process Workshop suggested a software process problem to the workshop participants [27]. The problem specifies a process in natural language; it concerns the development, change, and testing of a defective software module. Such a process is decomposed in a number of phases: designing, coding, testing, and management of change. There is an authority, CCB, in charge of the project; there is a programmers’ team PT coordinated by a project manager PM.
We show how a software process designer could model part of such a process in ESP. We start creating three blackboards: one for the CCB, one for the PM, and one for the PT. The following goal sets up the initial distributed environment in which the process will take place.

?- \{tsc(bb_of_CCB),
    tsc(bb_of_PM\{develop_change_and_test(Unit)\}),
    tsc(bb_of_PT)\}

For blackboard bb_of_PT, we specify the termination condition using the following invariants:

?- \{invariant(end_work(Id)@bb_of_PT), invariant(abort_work(Id)@bb_of_PT)\}

This means that blackboard bb_of_PT will terminate its activities when either tuple end_work(Id) or tuple abort_work(Id) is found.

The following ESP theory controls the overall process from the viewpoint of the project manager PM.

theory develop_change_and_test(Unit):-
  eval
  change(requirements(Unit),Id), \{authorization(Id)\}
  \rightarrow
  \{schedule_and_assign_tasks(Unit), start_work(Unit,Id)@bb_of_PT\}
# \{success(Id)\}
  \rightarrow
  \{end_work(Id)@bb_of_PT\}
# \{cancel_change(Id)\}
  \rightarrow
  \{abort_work(Id)@bb_of_PT\}

The theory develop_change_and_test(Unit) includes three rules: the first one describes the activation of the process, that starts when the CCB issues the authorization tuple and the specification of the changes for the target unit. The project manager responds to this "stimulus" by starting the schedule_and_assign_tasks phase, that is defined by another ESP theory. The other two rules state the possible terminal conditions of the process.

theory schedule_and_assign_tasks(Unit):-
  eval
  \{change(requirements(Unit),Id), project_plans(Plans)\}
  \rightarrow
  update(Plans,NewPlans)
  \{project_plans(NewPlans),
    mail(Assignments)@bb_of_PT,
    change(requirements(Unit),Id)@bb_of_PT\}
with
  update(Plans,NewPlans):- ...% definition of the new scheduling of tasks.

For brevity we do not give a complete process program. A detailed solution to this problem can be found in [3].
4.4 Discussion

The examples we have given in this section are admittedly not complex. Moreover, some of their features are also offered by some existing and well known tools. For instance, something similar to the first example is not difficult to do with the `make` tool or writing a script for a sophisticated editor like GNU Emacs, whereas the second example captures only the very basic mechanisms of a tool like `SCCS`.

However, these examples show that in ESP distributed computations are very easy to deal with. For instance, it is easy to specify different allocation policies. In the first example the three agents editor, compiler, and tester can be integrated in a unique blackboard, to be allocated on one workstation, or else they can be put in different blackboards, aiming at enforcing distribution and protection. Moreover, combining different software processes is also not difficult: the following goal sets up an environment that combines the program of the first example with the program of the second example:

```prolog
?- {tsc(maindb{checkin_manager,checkout_manager}),
   tsc(user1{user_db_manager(maindb), editor(module1,tty1)}),
   tsc(user2{user_db_manager(maindb), editor(module2,tty2)}),
   tsc(compile{compiler}),
   tsc(test{tester(data1),test data(data1) })}.
```

More important, we have chosen these examples in order to show that in ESP it is possible to program and control several levels of coordination:

- the overall structure of the software development environment, defined as set of named tuple spaces devoted to distinct activities;
- the activities encompassing several tuple spaces, defined by the communication flows of tuples that are sent from a tuple space to another;
- the coordination of agents inside a tuple space, defined by the multiset of tuples that are active in such a tuple space;
- the concurrent interactions among different agents in the same tuple space, defined by the semantics of tuple operations;
- the tasks that an agent can perform, defined by the rules of the theory that it executes;
- the actions that an agent has to perform for each task on which it is activated, defined by the knowledge base of the theory that it executes.

We believe that software process programming is a difficult and complex undertaking mainly because there are all these different coordination levels that have to be taken into account and amalgamated in a harmonious assembly. Although ESP is based on a small number of language mechanisms, it can be used to program all the above levels of coordination in a declarative way. In fact, ESP keeps from the logic programming paradigm the view of computation as proof search. Thus, we obtain for free a formal semantics for the local (i.e., non related to the tuple space) evaluations of agents. Admittedly, ESP moves away from pure logic programming to rely on a different, more expressive framework. In fact, ESP enriches logic programming with a set of coordination mechanisms for agent and tuple space creation, and for inter-tuple space communications.

Although ESP is a powerful and expressive language for programming coordination, the goal of managing the definition and evolution of the software process would be impossible to attack without the support of a specialized environment. Apart from the definition and execution of ESP programs, such an environment should allow the integration between a
software process and the tools it needs, offering a repository of services that can be used as building blocks of complex software processes. Such an environment is Oikos.

5 Oikos

Oikos is an experimental rule-based software development environment that supports process programs written in ESP [2]. Oikos provides a number of predefined facilities to model rule-based software processes over a distributed system. The overall approach consists of offering mechanisms that can be easily composed using ESP, which can be used to model different software processes.

In Oikos a software process is an ESP program that coordinates a dynamic collection of agents cooperating in a distributed environment modeled as a hierarchy of blackboards. The hierarchy offers a natural way of structuring a software process; conceptually, it is similar to the contractual hierarchy in IStar [20], but in IStar the nodes of the tree are documents, whereas in Oikos they are tuple spaces. In fact, the blackboard hierarchy is used to reflect the decomposition of the software process in subprocesses, according to a top-down refinement strategy. The hierarchy is not really constraining the communication flows among the participants of a software process, since blackboard names can be exchanged in tuples, and an agent can put tuples in any blackboard, provided that it knows the name of the destination. Therefore, highly dynamic communication patterns can be set up, even connecting blackboards at different levels of the hierarchy, if this is convenient.

Here we give only a short description of the current system architecture; the interested reader will find more details in [2,5,4].

5.1 Services in Oikos

A key concept in Oikos is the service, that is is a predefined subsystem that accepts requests from clients. Conceptually, a service offers access to shared resources according to a given set of rules. Shared resources include software products, tools, the process program itself, etc.

Formally, each service is defined by an ESP program that specifies the protocol of interaction with the service itself, i.e., which tuples must be put into its blackboard to submit a request. There are a number of standard services, that in Oikos play the role that primitive operators and data types play in a programming language. We discuss here only two Oikos services: the database service DBS and the user interface service UIS.

In order to store information on the documents of a software process, Oikos offers the service DBS that interfaces a database management system. Currently the DBS is Salad, a deductive database management system based on a logic language [15]. The DBS accepts scheme definition requests, queries, and transaction requests; it also enforces access rights. One of the first duties of the software process manager is to set up the project database by defining its scheme, describing properties and relations among the different kinds of documents. Oikos provides a few predefined schemes, for the reusable documents related to Oikos itself, e.g., those containing the service specifications and those related to the tools already available at start up.

Documents are actually stored in files, and are manipulated in a blackboard only via their identifiers. When it is necessary to operate on a document with a tool, e.g., an editor, the document is retrieved from the underlying file system.
Users accesses Oikos through shells managed by the User Interface Service (UIS). The user can see the contents of its blackboard in a window; his action must comply the rules defined in the UIS specification. UIS is a service because several different shells can coexist. A shell offers a flexible way to monitor a software process, since the user can activate it on a blackboard, looking at the tuple flow, and even saving some tuples to record the evolution of the software process [30].

Fig. 3 shows what the user sees when he is connected to the ESP programming environment under XWindows (the ESP programming environment is called EXPO [5]). Conceptually, the user opens a window through which he can see the contents of a blackboard, and also put and remove tuples in such a blackboard.

In fig.3 the leftmost top window shows the current contents of the blackboard. A number of buttons allow the manipulation of the window. The leftmost bottom window is used to edit “shell patterns”, i.e., rules that are inserted and evaluated “on the fly” when the user is interested in some events that happens in this blackboard, typically when some tuples appear or disappear. The rightmost top window shows the history of user actions. The rightmost bottom window shows the name of the blackboard, and three buttons used to manipulate the windows.
6 Comparison with related work

5.2 Oikos Implementation

Oikos has been written mostly in ESP. Fig. 4 shows the architecture of the Oikos prototype that has been implemented at the University of Pisa on top of a local network of Sun workstations. ESP provides the basic mechanisms for physical distribution and dynamic activation of communicating processes.

The three main layers of the Oikos architecture are shown in Fig. 4: the EXPO programming environment, which is written in ESP and provides an XWindows-based user interface [5]; a collection of separate processes, that implement a distributed ESP run-time system; the underlying operating system, that is Unix. The hierarchy of blackboards is represented by nested boxes. The processes in the second layer are depicted by circles: an ESP process is the local interpreter of the ESP language, and there are as many of them as machines in the network, eager to interpret pieces of the ESP program. For a detailed exposition see [12,5].

6 Comparison with related work

First we compare ESP with other coordination languages, then we compare it with other rule-based process programming languages.
6.1 Comparing ESP with other coordination languages

ESP is a language that combines Linda-like coordination with Prolog-like computation. It is an extension of Prolog in the same sense that C-Linda is an extension of C. In comparison to Linda, the main difference is that ESP is based on multiple tuple spaces. Moreover, it gains expressiveness thanks to the “magic” of logic programming. However, ESP is not a mere Prolog-Linda dialect, because of the structure of the clauses. Such a structure, i.e., Test and Consume–Compute–Output makes ESP similar to parallel logic languages like Flat Concurrent Prolog and GHC [36]. The most important difference with respect to these languages is that ESP tuple spaces have a state that agents manipulate by adding and deleting tuples; FCP and GHC operate on shared streams in a monotonic way, i.e., they can only add new data. Moreover, ESP is a true extension of Prolog, whereas FCP and GHC have no sequential component.

There are now other parallel logic languages that are based on a notion of logical tuple space. A programming notation similar to ESP is Swarm [35], a specification language based on UNITY that has been introduced for studying the programming logic underlying tuple space communications. ESP is more tightly related than Swarm to logic programming, and it has many constructs for programming in the large, that are currently missing in Swarm. Linear Objects (LO) [7] is an object oriented language based on a concept of multiple tuple spaces that communicate by broadcasting. Apart from this, there are many similarities between ESP and LO, and we are studying them in a unified semantic framework [6].

With respect to other rule based programming languages, like for instance production systems like OPS5, ESP rules merge cleanly two basic evaluation mechanisms: forward and backward chaining. Backward chaining is used during the local computation of Prolog goals; a form of parallel forward chaining is used as basic mechanism for activating agents. A logic tuple space is also very similar to a blackboard [31], and in fact many ESP implementation issues can be discussed in the same way as for blackboard systems. A detailed discussion of ESP related implementation issues is contained in [12,18].

6.2 Comparing ESP with other rule-based process programming languages

A well known project that inspired Oikos is the rule-based software development environment MARVEL [8,9]. MARVEL follows an object oriented design paradigm, and relies upon a special rule-based language derived from production systems for AI applications. A software process is specified by three sets of specifications: the project ruleset describes the development process, the project type set specifies the project data, and the project tool set defines the interface with external tools.

This is an example of a MARVEL rule [9]:

```
calculate(?f:FILE):
    (?f.compile_status = NotCompiled)
    COMPILER compile ?f.contents ?f.object_code ?f.error_msg "-g"
    (and (?f.compile_status = Compiled)
        (?f.object_time_stamp = CurrentTime));
    (?f.compile_status = Error);
```

MARVEL rules have a name and include three sections: a precondition, an activity part, and a postcondition. The precondition is a condition to be evaluated with respect to the current status of the project database; if it is verified, the activity part is executed, that usually invokes...
a tool. At the end of the tool activity the postcondition specifies the effects of the rule on the project database. Such a rule could easily translated in ESP as follows:

\[
\text{\{compile(c\_file(File,notCompiled))\}} \\
\rightarrow \\
\text{do\_compile('cc -g', File, Object\_code, Error\_msg), time(CurrentTime)} \\
\text{\{c\_file(File,compiled), object\_file(File, CurrentTime)\}} \\
\text{fail} \\
\text{\{c\_file(File,error)\}} \\
\text{with} \\
\text{do\_compile(Compiler, File, Object\_code, Error\_msg):- ...}
\]

MARVEL puts emphasis on designing the interaction protocols used by a multiplicity of users that share a project database. In ESP more importance is put on the environment’s architecture design, aiming at formalizing the coordination model underlying users’ activities. A key issue in ESP is the use of the blackboard model in combination with logic programming.

The blackboard model of problem solving is well known in Artificial Intelligence [31], whereas in software engineering it has been exploited only in a few projects. For instance, Agora [10] uses a blackboard architecture for interprocess communication among heterogeneous software components. A more recent and relevant example is REBUS, where the blackboard is curiously called “whiteboard” [40]. REBUS is a process program written in APPL/A [39], an extension of Ada for software process modeling. The REBUS whiteboard is used through five ad hoc primitives that manipulate its contents. The whiteboard is used to coordinate the activities of a number of participants in the software process.

In our experience, the combination of the blackboard model with logic programming provides a powerful rule-based framework to specify and prototype distributed software development environments and the related software processes. The use of logic programming techniques and tools in software engineering is slowly gaining popularity. For instance, the Darwin project [28,29] develops a Prolog-based framework to support rule-based software engineering environments, called law governed systems. In Darwin the law is an explicit statement of a rule that must be followed by agents in the environment. Environments written in ESP can be considered as law-governed systems whose laws are stated in a parallel language.

The use of Prolog for software process programming was suggested for the first time in [32], but without introducing any concurrency mechanisms. A more successful project for a rule-based environment centered upon Prolog is Merlin [24,34]. Merlin can be used to model rule-based software processes using an extension of Prolog in which both forward chaining and backward chaining computations are possible. Merlin extends Prolog with a few imperative constructs, like for instance \text{CALL, INSERT and REMOVE}, that operate on clauses contained in the knowledge base. This is an example of a Merlin rule:

\[
\text{IF document(module, Object\_Name, to\_be\_compiled) } \\
\text{THEN } \\
\text{CALL(compiler, Object\_Name, Compile\_Status), } \\
\text{REMOVE(document(module, Object\_Name, to\_be\_compiled)), } \\
\text{INSERT(document(module, Object\_Name, Compile\_Status)).}
\]

Such a rule is easily translated in ESP as follows:

\[
\text{\{document(module, Object\_Name, to\_be\_compiled)\}} \\
\rightarrow
\]
More complex sequences including several \texttt{INSERT} and \texttt{REMOVE} commands in an arbitrary order could be rephrased in ESP using chains of rules.

Currently the Merlin project does not take into account concurrency and distribution issues. We believe that the PoliS coordination model would be a natural way to extend Merlin with features to deal with concurrency and parallelism.

7 Conclusions

In this paper we have introduced PoliS, a coordination model useful for designing distributed systems. A programming notation based on a combination of PoliS with Prolog, ESP, has been used as a language for software process modeling. ESP has been used in the design and implementation of Oikos, a rule-based software development environment that is distributed over a local network.

The ESP programming environment has been operational for three years. It has been used to implement a few distributed multiuser applications, like a referee for coordinating four bridge players, a financial simulation of stock exchanges, and a simple e-mail system. Some software processes have been simulated, but not fully enacted, like for instance the software process defined in section 4.3.

Our future plans include: the development of a graphic specification language to help in the development of software processes that can be successively modeled in ESP; the definition of planning tools for assisting users in the software process; the possibility of changing a software process program during its execution.

Moreover, we feel that PoliS and the whole coordination framework based on Multiple Tuple Spaces deserve to be more deeply studied. For instance, we are developing a programming logic for PoliS, aimed at developing a specification language suitable for distributed applications.

Acknowledgments The contribution given by the referees’ comments to the improvement of this paper is invaluable. The author is also very grateful to N.Carriero and D.Gelernter at Yale, for many discussions on Linda and PoliS, and to his PhD thesis supervisor, U.Montanari, for the suggestion of studying associative coordination mechanisms. Last but not least, the author thanks all the members of both the Shared Prolog and Oikos research groups in Pisa, especially V.Ambriola, A.Brogi, M.Danelutto, M.Gaspari, and C.Montangero.

This work has been partially supported by C.N.R. Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, and by the Italian Ministry for University and Scientific Research (MURST).

References

REFERENCES


