PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

A Distributed Programming Environment based on Logic Tuple Spaces

Paolo Ciancarini
Dipartimento di Informatica
University of Pisa - Italy
E-mail: cianca@di.unipi.it

David Gelernter
Dept. of Computer Science
Yale University - USA
E-mail: dhg@cs.yale.edu

Abstract

In this paper we describe PoliS, a coordination model based on Multiple Tuple Spaces. PoliS addresses the specification and coordination of logically distributed systems. We show that it can be used as a basic model for designing distributed and rule-based software development environments. In fact, PoliS has been used in the design of Oikos, a distributed software development environment. It has been specified and implemented using Extended Shared Prolog, a parallel logic language that smoothly combines the PoliS approach, to deal with concurrency and distribution, with Prolog, to deal with rules and deduction. Such a combination of blackboard-based communications and logic programming provides a powerful framework in which experiments about different environment architectures can be performed and evaluated.

1 Introduction

The concept of *software development environment* is a key issue in software engineering. Logic programming was proposed as an interesting technology for designing and implementing innovative environments since the first FGCS conference [Furukawa *et al.*, 1984]. However, only recently a theory for abstractly studying and comparing different software development environments has been developed [Perry and Kaiser, 1991]. Perry and Kaiser introduced a hierarchy of classes of software development environments. Their hierarchy is roughly based on the number of programmers involved and includes four classes: individual, family, city, and state. Each class is characterized by three interrelated components: policies, mechanisms, and structures. *Policies* are the strategies and the constraints imposed on the programmer by the environment; *mechanisms* are the tools supported by the environment; *structures* are the objects on which mechanisms operate.

The main contribution of this paper is the definition of an abstract paradigm for modeling and implementing a software development environment at the "city" level. The paradigm is called *PoliSpaces*, because it is based on Multiple Tuple Spaces [Gelernter, 1989, Matsuoka and Kawai, 1988], and it is shortened to *PoliS*, from the Greek word for "town". Using the terminology introduced by Perry and Kaiser, our model allows programmers to express different coordination policies simply and consistently, giving to the environment designer a powerful tool for structuring distributed software development environments.

Our proposal is twofold. Firstly, we define an abstract coordination model that can be used as a tool in the design of a distributed software development environment supporting activities by many agents. A coordination model is a set of mechanisms for expressing and controlling distributed activities [Ciancarini, 1990b, Carriero and Gelernter, 1991]. The activities themselves can be expressed in any sequential language; their interaction with respect to other activities is defined using the coordination model. To make clear the coordination issues, we have introduced Extended Shared Prolog (ESP for short) [Bucci *et al.*, 1991], a parallel logic language based on PoliS.

Secondly, we show how a software development environment can be specified using ESP. The idea is that the environment enforces protocols that specify goals, duties, and constraints of the agents involved in the software development process. We show how ESP can be used to specify simple programming environments corresponding to simple software development processes. The power of this method has been tested in the design of Oikos, a fully-fledged distributed environment [Ambriola *et al.*, 1990b]. Oikos offers a number of services giving some basic facilities, like access to databases and private workspaces, activation of shells, etc. ESP can be used to reconfigure and customize the environment.

The paper is organized as follows: Section 2 describes PoliS. Section 3 introduces 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.

2 PoliSpaces: A Model for Coordination

Intuitively, a *PoliSpace* is like an abstract town where there are many places; in each place many agents cooperate. In the town many activities take place simultaneously, mostly independently; however, they are ruled by 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).

Formally, a PoliSpace is a distributed system that is a collection of *tuple spaces.* A *tuple space* is a multiset of tuples: a *tuple* is simply a sequence of fields. More precisely, in PoliS three concepts are important: tuples, agents, and places.

- A *tuple* is a structured data object that is a sequence of values. It is produced by some agent in some space, and it remains there until some agent consumes it. A tuple can be "copied" (read) or "consumed" (read and deleted) only by an agent included in the same place. 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 [Gelernter, 1985], or a mechanism based on unification, as in a logic language.

- An *agent* is an execution thread, *i.e.,* it is an abstraction of a running program completely independent of other agents. An agent is contained in a particular place and is able to perform some operations on the tuples that it contains. The semantics of an agent can be described as follows: an agent looks continously for some tuples; when they are found, it executes a computation consisting of instructions written in some sequential programming language; finally, it creates new entities (tuples or places). The sequential language chosen for programming the internal working of the agent is left outside the scope of the model as a degree of freedom, so that agents written in many different sequential languages can coexist.

- A *place* is a named multiset of tuples (in this paper we will use as synonyms for "place" the terms *tuple space* and *blackboard*). Places are containers in the sense that the universe of tuples and agents is partitioned in a number of places. Places can be dynamically created by agents. A place is both a computing space and a communication channel, *i.e.,* a shared data structure on which agents read and write data; in fact, an agent can produce a tuple inside a place and it has access to every tuple in its own place. An agent cannot directly read the contents of an external place.

The PoliS model is enforced by a notation whose syntax is informally described below.

2.1 Places

A place is a named multiset of tuples. Syntactically, we will write places as braced sequences of tuples.

Example: For instance, we write

```
place1{ (a) (b,X) }
```

to describe a place named `place1` containing two tuples. □

An interesting feature of PoliS places is that they have *names.* Agents can send tuples outside their own place using the name of another place. The name system of places is an interesting design choice that has been left out of PoliS: it is another degree of freedom, just like the choice of the matching mechanism to access the tuples, and the sequential language for expressing local computations. For instance, in ESP the names are structured: they are paths in Unix-like style.

2.2 Tuples

Tuples are sequences of variables and values. Values obviously depend on the chosen sequential component, *i.e.,* the sequential programming language adopted for agents. However, in PoliS a number of basic value types, as well as lists of these values, are allowed. Tuples denote themselves; they are simply data objects that exist in a place, produced by some agent and possibly in the future consumed by some agent. An important topic is the scope of variables inside tuples contained in a place: the scope of these variables spans only the tuple to which they belong. This means that each tuple inside a Tuple Space is completely independent from other tuples.

2.3 Agents

Abstractly, agents are execution *threads, i.e.,* an agent is a process executing some program. Syntactically, an agent is represented by a tuple and executes the program contained in another (special) tuple, called *program-tuple.* An agent can use the following *abstract tuple operations* for its interaction with the landscape it lives in:

- associative *test* of a tuple contained in the same place the agent is;
- associative *consumption* of a tuple from the same place the agent is;
- asynchronous *creation* of a place or a tuple inside the landscape the agent knows.

These operations are borrowed from Linda. Actually, Linda offers an intuitive syntax for PoliS operations, introducing two different "flavors" for the test and consumption operations (they can be either blocking or not-blocking), and two not blocking operators for the creation of entities. The blocking *test* operation in Linda is written `read(`*Tuple_schemata*`)`, the non-blocking *test* operation is written `readp(`*Tuple_schemata*`)`, the blocking *consumption* operation is written `in(`*Tuple_schemata*`)`, the non- blocking *consumption* operation is written `inp(`*Tuple_schemata*`)` (a Tuple_Schemata is simply a tuple containing variables, *i.e.*, wild cards that match any actual argument inside a tuple contained in the Tuple Space).

The *creation* operation is written as `out(`*Tuple*`)` in the case of tuples, and `Name.tsc()` in the case of places (in this paper we assume no structure on the set of names of Tuple Spaces).

An agent can output any of these entities:

- a tuple; the operation is written `out(Tuple)` in case of local writing, `name.out(Tuple)` in case of outside writing;
- a Tuple Space (*i.e.*, it creates a new place); the operation is written `name.tsc()`.

The destination of such operations is always a place. The target of an `out` operation is specified using a record-like notation. If no target is specified, the Tuple Space of the agent is used by default. What happens if an `out` operation targets an external tuple space that does not exists? PoliS tries to follow the Linda semantics: `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. Thus, communications among places are supported by a meta Tuple Space where undelivered tuples remain deposited; whenever a place comes into existence, the undelivered tuples "pop up" in the tuple space.

If an agent needs to be certain that a message arrived somewhere, it must explicitly use some protocol. For instance it could send the message and an agent that, upon arrival in the target Tuple Space, sends back an ack.

Finally, we note that an agent can *test* or *consume* tuples representing other agents. Such operations are useful to build agents that schedule agents. Places cannot be operands neither for testing nor for consuming, because the obvious semantics for such operations (test a whole place, delete a whole place) should necessarily manipulate the global state of a place, sharply contrasting with the asynchronous nature of its internal activities.

PoliS agents have a reactive semantics defined by a fixed protocol of tuple operations. The basic protocol is the following (we borrow some syntax from regular expressions: with *op** we intend a sequence of indefinite length of tuple operations):

*test**; *consume**; *loc_eval*; *out**

Syntactically, such a protocol is written inside a program-tuple.

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

The `Heading` is a normal tuple. Instead, `Test`, `Consume`, and `Out` are actually sequences of tuple operations, whereas `Loc_Eval` is a sequential computation that has no side effect on the place to which the agent belongs. An agent is activated when the place contains both a program-tuple and a normal tuple matching the heading in the program-tuple. The second component of a program-tuple is also called a *pattern*. Executing a pattern, an agent will do the following actions:

- it reads associatively something from its place using any number of *test* operations; actually the PoliS *test* operation has a broader semantics than `read` in Linda: a number of predefined tests on the place are allowed, depending on the chosen type system for tuple arguments. Some useful general predefined tests are: relational (binary) predicates, a `var` predicate to check if an argument inside a tuple is a variable, and a `self` predicate returning the name of the place in which an agent is located.
- it deletes some tuples using any number of *consume* operations.

When an agent has finished testing and deleting tuples from the place, it "reacts" and starts a computation that ends by creating some new objects in the landscape.

- it executes a "local evaluation" that has no effect on the place and is invisible from outside the agent insofar as no operations on the place are allowed; this local computation is expressed in a sequential programming language,
- it outputs the results obtained in a number of places it "knows": these outputs can consist of tuples or places;

- 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.

Which is the computing model underlying agents' computations? The idea is that agents are stateless and reactive, *i.e.*, they compute when a "molecule" can be built inside the Tuple Space. A molecule is composed of a program-tuple, a normal tuple matching the first field 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 as a result creates new entities as specified in the *create* section. This "chemical" model is also used in GAMMA [Banatre and LeMetayer, 1990].

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 which can react and consume tuple (pong) to produce 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. □

Even if the relationship among places, agents, program-tuples, and local evaluations can look slightly contrived, actually their relative meaning is quite simple: a place defines an AND-parallel computation of agents; an agent executes the computation defined by a program-tuple; the agent reacts to the contents of its place with a local evaluation followed by the creation of new entities, either tuples or places.

3 ESP: A programming notation based on PoliS

PoliS is a coordination model that could accommodate any sequential language as sequential component for local computations inside agents. For example, 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 place is allowed for every program. In Linda tuple operations inside agents are not structured (*i.e.*, you can have in, read, and out in any order), but any sequence of Linda operations can be split in a number of subsequences such that each begins with read/in operations and terminates with out operations.

PoliS is a model for designing coordination of distributed systems. It is introduced as a paradigm for explorative distributed programming, and can be considered a useful prototyping model for distributed applications. In order to explore its usefulness for this task, we have defined Extended Shared Prolog (ESP), a programming language that embeds its main features.

ESP is a logically distributed extension of the parallel logic language Shared Prolog, which is a logic language that uses the blackboard model for interprocess communication [Brogi and Ciancarini, 1991]. With some approximations, Shared Prolog can be considered a logic counterpart of the Linda family of physically distributed programming languages [Gelernter, 1985]. The main difference is that Shared Prolog gains in expressive power with respect to Linda by exploiting unification and backtracking during synchronization with the blackboard (Linda uses pattern matching, and no backtracking is allowed). ESP generalizes Shared Prolog allowing multiple blackboards using a hierarchical name system.

3.1 Theories

An ESP program is composed of a set of theories. Each *theory* has the following syntactical structure:

> **theory** $name(V1, \ldots, Vn)$:-
> **eval** $pattern_1 \# \ldots \# pattern_k$
> **with** *Prolog_program*

A theory is identified by a name and zero or more arguments V_i that are logic variables that scope over the patterns. The *theory interface* follows the keyword eval and includes a number of patterns, separated by the symbol #; the *theory implementation* is the Prolog program that follows the keyword with. If we compare ESP with other languages for programming in the large, the set of patterns of a theory can be considered the interface of a module, while the Prolog program is the private implementation of the module.

Logic patterns are clauses that include *test* and *consume* operations, a *loc_eval* that is a goal to be evaluated with respect to the *Prolog_program*, and finally some *out* operations. For simplicity and consistency with the logic paradigm, test operations are written as goals, whereas consumption and creation operations are put between braces.

```
Test {Consume} — Goal {Success} fail
{Failure}
```

The combination of Test and Consume operations is a *guard*: when such a guard is satisfied, *i.e.*, when all its test and consume operations are completed, the pattern commits 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-set is produced, else the Failure out-set is produced. Thus, an ESP pattern is similar to a Concurrent Prolog clause (but ESP clauses are failure-free), and a theory corresponds to the definition of a CP predicate.

As a simple example, we show a simple theory including one pattern: it defines an agent computing a value as a function of some input, or outputs an error tuple if the evaluation fails.

```
theory agent(State) :-
eval
    {tuple(Input)} % consume
    →
    f(Input, State, Output, NewState)
    {tuple(Output), agent(NewState)} % Success
    fail {error(f(Input,State)), agent(State)}
with
     f(I,S,O,NS):- ... % Prolog_program
```

3.2 Agents

Logic agents are represented by active tuples; they react to the presence of other tuples in their blackboard. They can read and delete tuples from their blackboard: they answer by writing tuples in any blackboard they know. The relation between input and output is defined by a Prolog program (with a slight abuse of language, we will say sometime that the behavior of an agent is defined by a theory). Several agents with the same theory can be active at the same time, in the same blackboard or in different ones.

A notable feature of ESP is that control flow of *test* and *consume* operations is ruled by backtracking. Each *test* or *consume* operation either is successful or fails; a failure activates backtracking to the preceding operation. The formal semantics of such a mechanism has been studied in [Brogi and Ciancarini, 1991].

3.3 Blackboards

For reasons that will become clear in Sect.5, the name system chosen for ESP blackboards defines a hierarchical system. In fact, blackboard names are paths in the style of a Unix file system. Such a hierarchy is not limiting the communication patterns among the agents, 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.

Blackboards can be dynamically created by agents simply outputting an *activation goal* that specifies a number of agents. This is the syntax of an activation goal.

```
?- child{agent_1, ... , agent_n } @ parent.
```

This goal creates a blackboard named *child* as offspring of blackboard *parent*.

In general, the execution of an ESP goal builds a tree of blackboards. Syntactically, a blackboard is a multiset of tuples that are Prolog terms. Semantically, a blackboard defines an AND parallel evaluation that transforms the contents of the blackboard itself. The actors of such an evaluation are logic agents, whose evaluations are defined by Prolog programs.

4 Programming with ESP

The activity of programming with ESP consists of building distributed systems; this topic has been explored in another paper [Ciancarini, 1990a]. Here we will show how ESP can be used as a specification and design language for software development environments.

4.1 A Tiny Programming Environment

A rule-based distributed software development environment can be easily specified in ESP. Rule-based software development environments have recently become popular [Barghouti and Kaiser, 1990] because they can be used to support process programming, *i.e.*, the activity of specifying multiagent software development.

A very simple programming environment can be set up including an editor and a compiler. Suppose we have to specify a software development process that consists of editing a file, then compiling it as soon as the editing by a programmer is terminated; Fig.1 depicts such an environment as a PoliSpace. If the compilation gives no errors, the object program has to be invoked and executed using some test data.

In order to build the ESP program that implements such a PoliSpace we need three theories: one for an editing agent, one for a compiler agent, and one for an executing agent. We show the code for the compiler theory.

```
theory compiler:-
eval
    {compile(File)}
    →
    call_compiler(File),
```

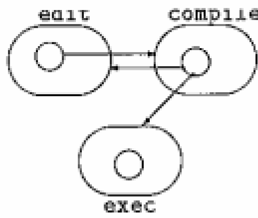

Figure 1: *A PoliSpace Coordinating a Simple Programming Environment*

```
    {compiled(File)}@exec
    fail  {do_edit(File)}@edit
with.
    call_compiler(File):- ...
     % invoke Prolog-Unix envelope for cc
     % fails if compilations fails for errors
```

Such a theory is called *envelope* because they incapsulate external software tools [Kaiser *et al.*, 1987]. 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.*, success or failure).

This minimal programming environment enforces a simple *edit-compile-exec* programming model. Admittedly, something similar is not difficult to do with sophisticated editors like GNUEmacs, however in ESP distribution and remote evaluation are very easy to deal with. Moreover, it is easy to specify different interaction paradigms. For instance the three agents `editor`, `compiler`, and `executor` are easily integrated in a unique blackboard, or can be separated in different blackboards, as in Fig.1, aiming at enforcing distribution and protection.

4.2 A Multiuser Environment enforcing an Access Protocol

A software project is composed of a set of modules on which a team of programmers operate. The updated public version of the whole project is stored within a *main database*. Users can access the main database in *read mode*. It is not possible to directly change the main database. The core of the environment is a reserve/deposit access protocol to 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, the 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.

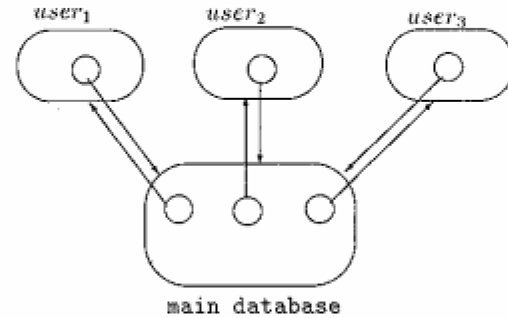

Figure 2: *A PoliSpace Coordinating a Multiuser Environment*

The PoliSpace realizing such an environment is showed in Fig.2.

```
theory user_database_manager:-
eval
    self(Udb), {check_in(File, Dbmain)}
    —
    {check_in(File,Udb)}@Dbmain
#
    self(Udb), {check_out(File,Dbmain)}
    —
    {check_out(Udb,File)}@Dbmain

theory main__database_manager:-
eval
    file(F), not reserved(F,_), {check_out(P,F)}
    —
    {reserved(F,by(P)}, {file(F)}@P
#
    not file(F), {check_out(P,F)}
    —
    {error(nofile(F))}@P
#
    reserved(F,by(OP)), P ≠ OP, {check_out(P,F)}
    —
    {error(is_locked(F,by(OP)))}@P
#
    file(F), not reserved(F), {check_in(F,P)}
    —
    {error(file_exists(F))}@P
#
    file(F), reserved(F,by(F,OP), P ≠ OP,
{check_in(F,P)}
    —
    {error(is_locked(F,by(OP)))}@P
#
    not file(F), {check_in(F,P)}
```

```
→
{file(F)}, {created(F)}@P)
```

We show only the code of the theories `user_database_manager`, that handles the user's requests in a user database, and `main_database_manager`, which guarantees the consistency of the main database.

5 Oikos

Oikos is a distributed software development environment based on PoliS and written in ESP [Ambriola *et al.*, 1990b]. Oikos provides a number of standard facilities that can be easily configured using ESP itself. The overall approach consists of offering mechanisms that can be easily composed, in order to easily explore different environment designs.

The ESP blackboard hierarchy offers a natural way of structuring a software development environment. It is used to reflect its decomposition in subenvironments, according to a top-down refinement strategy. The blackboard hierarchy is not really constraining the communication patterns among the agents participating in a software development 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.

5.1 A Prototype Implementation of Oikos

The Oikos prototype has been implemented on top of a local network connecting some Sun workstations and a Vax mainframe. Oikos is written is ESP, that provides the basic mechanisms for physical distribution and dynamic activation of communicating processes. ESP itself is implemented partly in C and partly in Prolog [Bucci *et al.*, 1991]. The standard set of Prolog system predicates has been augmented with IPC mechanisms using Unix Internet sockets [Ambriola *et al.*, 1990a].

The three layers of the Oikos architecture are: the Oikos runtime support, which is written in ESP and provides escapes to the underlying operating system; a collection of separate processes, that implement a distributed ESP run-time system; the underlying operating system, UNIX in this case. 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 more detailed exposition see [Bucci *et al.*, 1991].

5.2 Oikos Services

Oikos provides a set of basic services. A *service* offers access to shared resources according to a given protocol. The public interface of a service specifies the protocol of interaction with the service, *i.e.*, which tuples must be put into its blackboard to obtain its service. For lack of space, we simply summarize the Oikos *standard services*, which play the role that primitive operators and data types play in a programming language. We discuss here the most meaningful only, *i.e.*, those that are fundamental in any software development process.

The Tool Kit Server (TKS), the Service and Theory Server (STS) and the History Server (HS) offer restricted access to databases of system data, *e.g.*, those modeling the predefined documents. A User Interface Service (UIS) is used to interact with running software process programs, whereas the Workspace Server (WS) allows users to run the tools and the executable products of the software process. The DataBase Server (DBS) offers unrestricted access to a general purpose project database, and is therefore used to set up specific project databases. Finally, the Oikos Run Time System (ORTS) can also be seen as a server offering essential services, like escapes to the underlying operating system. All these services, except ORTS, can be simultaneously activated several times in different blackboards.

The user accesses Oikos through a special interactive service called User Interface Service (UIS). It is a service because several different UIS can coexist, and their definitions are ESP programs found in STS. A UIS shows the user the contents of its blackboard in a window, and acts according to the user's input. A UIS offers also 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 with the history server HS. UISs are the basic blocks of the role services, *i.e.*, those parts of the process program that allows users to interact with the software process.

For lack of space, here we do not show how Oikos is used in a real software development process. The interested reader can see the example contained in [Ambriola *et al.*, 1990b].

6 Conclusions

In this paper we have introduced PoliS, a coordination model useful for designing distributed systems. A programming notation based on PoliS, ESP, has been used to illustrate the design of Oikos, a distributed software development environment. The goal of the ESP/Oikos project is to assess the combination of the blackboard model with logic programming in the de-

sign of distributed programming environments. While the blackboard model is well known in Artificial Intelligence, its use in Software engineering is quite novel.

After completing the implementation of Oikos, our future plans include the study of the impact of different models of tool coordination in the definition of planning tools for assisting users in the software process, and the analysis of the role interplay in dealing with the software process itself.

Acknowledgements P.Ciancarini has been partially supported by C.N.R. Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, and M.U.R.S.T. The authors are very grateful to N.Carriero at Yale, for many discussions on Linda and PoliS, and to the Shared Prolog and Oikos groups in Pisa, including V.Ambriola, A.Bucci, T.Castagnetti, M.Danelutto, and C.Montangero.

References

[Ambriola *et al.*, 1990a] Vincenzo Ambriola, Paolo Ciancarini, and Marco Danelutto. Design and distributed implementation of the parallel logic language Shared Prolog. In *Proceedings of ACM Symp. on Principles and Practice of Parallel Programming*, volume 25:3 of *SIGPLAN Notices*, pages 40–49, 1990.

[Ambriola *et al.*, 1990b] Vincenzo Ambriola, Paolo Ciancarini, and Carlo Montangero. Enacting software processes in Oikos. In *Proceedings of ACM SIGSOFT Conf. on Software Development Environments*, volume 15:6 of *ACM SIGSOFT Software Engineering Notes*, pages 12–23, 1990.

[Banatre and LeMetayer, 1990] Jean-Pierre Banatre and Daniel LeMetayer. The gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[Barghouti and Kaiser, 1990] Naser Barghouti and Gail Kaiser. Modeling concurrency in rule-based development environments. *IEEE Expert*, 5(6):15–27, December 1990.

[Brogi and Ciancarini, 1991] Antonio Brogi and Paolo Ciancarini. The concurrent language Shared Prolog. *ACM Trans. on Programming Languages and Systems*, 13(1):99–123, 1991.

[Bucci *et al.*, 1991] Annamaria Bucci, Paolo Ciancarini, and Carlo Montangero. Extended Shared Prolog: A multiple tuple space logic language. In *Proceedings of the 10th Japanese Logic Programming Conference*, 1991.

[Carriero and Gelernter, 1991] Nick Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 1991.

[Ciancarini, 1990a] Paolo Ciancarini. Blackboard programming in Shared Prolog. In David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 170–185. MIT Press, 1990.

[Ciancarini, 1990b] Paolo Ciancarini. Coordination languages for open system programming. In *Proceedings IEEE Conf. on Computer Languages*, pages 120–129, 1990.

[Furukawa *et al.*, 1984] K. Furukawa, A. Takeuchi, S. Kunifujii, H. Yasukawa, M. Ohki, and K. Ueda. Mandala: A logic based knowledge programming system. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, pages 613–622, 1984.

[Gelernter, 1985] David Gelernter. Generative communication in Linda. *ACM Trans. on Programming Languages and Systems*, 7(1):80–112, 1985.

[Gelernter, 1989] David Gelernter. Multiple tuple spaces in Linda. In *Proceedings of PARLE '89*, volume 365 of *Lecture Notes in Computer Science*, pages 20–27, 1989.

[Kaiser *et al.*, 1987] Gail Kaiser, Simon Kaplan, and Josephine Micallef. Multiuser, distributed language based environments. *IEEE Software*, 4(11):58–67, 1987.

[Matsuoka and Kawai, 1988] S. Matsuoka and S. Kawai. Using tuple-space communication in distributed object-oriented architectures. In *Proc. OOPSLA '88*, volume 23:11 of *ACM SIGPLAN Notices*, pages 276–284, November 1988.

[Perry and Kaiser, 1991] Dewayne Perry and Gail Kaiser. Models of software development environments. *IEEE Trans. on Software Engineering*, 17(3):283–295, 1991.