A LAYERED ARCHITECTURE FOR
GROUP-ORIENTED PARALLEL AND
DISTRIBUTED LOGIC PROGRAMMING

JOSE C. CUNHA, PEDRO MEDEIROS, FERNANDA
BARBOSA, RUI MARQUES AND JOAO LOURENCO

This paper reports on our experimentation with the design of a layered
software architecture for the support of parallel and distributed execution
of Logic Programming Systems, how it offers a set of concurrency and
communication abstractions that are integrated in an extended Prolog,
and how this is used to support a high-level group-oriented distributed
logic language.

1. INTRODUCTION

The work we describe in this paper is part of an ongoing project [Cun94] which aims
at the development of an integrated parallel and distributed programming environment
for a heterogeneous network with multicomputer nodes. Several programming
language models and parallel programming tools will be supported in the environment,
on top of a common software platform that interfaces with the underlying
operating systems and hardware architecture. The main supported languages are
C, C++, FORTRAN and Prolog, extended with constructs for the specification of
parallelism and distribution, which are suitably adapted to the each base language

Address correspondence to Universidade Nova de Lisboa, Departamento de Informática,
P-2825 Monte da Caparica, Portugal, e-mail: jcc@ft.unl.pt
This research was supported in part by Programa Ciência and project PROLOPPE of Pro-
grama PRAXIS XXI; project PADIPRO - DEC European External Research Contract P-005
semantics, and embedded into its corresponding abstract machine. More ambitious user interface languages are also being envisaged for the environment, e.g., a visual programming language.

The basic motivation for this project was to get the ability of exploiting parallelism in a wide spectrum of application domains, ranging from AI and Robotics, Neural Networks, Computational Fluid Dynamics, and Simulation for Environmental Sciences. These experiments are being conducted on top of a heterogeneous hardware platform, a LAN with several types of UNIX machines and two SUN-hosted multicomputer nodes (each with 16 Transputers, under the control of proprietary operating and runtime systems).

Here we focus our discussion on the design and implementation issues of the logic programming components that we have been developing for this environment.

After about fifteen years of research in this field, the basic motivations still remain more or less unchanged: to get significant speed-ups in logic program execution for complex problem solving, and to benefit from the potential declarativity of logic programming dialects as a form of providing transparency at the user level. The latter motivation aims at a programming style which focuses on the logical aspects of the application (e.g., problem decomposition and forms of process cooperation), instead of on the low-level control required by parallel and/or distributed execution.

We argue that explicit specification of parallelism and distribution at the user level is currently required by a large diversity of applications, both in the scientific and industrial communities, concerning the exploration of logic languages for the modeling of distributed multi-agent systems. Examples can be drawn from multiple areas, such as expert systems, distributed intelligent data base systems with advanced knowledge processing capabilities, automated reasoning systems, intelligent robotics systems. Common to these applications is the need of some inference or reasoning capability, coupled with a requirement for an adequate model to express interactions among multiple autonomous (and possibly dynamically evolving) components of an application.

There have been many distinct proposals for new programming models using this approach. However, they have suffered from several relevant criticisms, e.g., very complex concurrency semantics definitions; only suited to specific problem patterns; unable to conciliate the fulfillment of logical completeness with efficient implementations. Another important criticism concerns the lack of structuring abstractions for the development of large scale applications. This is a consequence of most of the approaches being based on the Horn Clause Logic or on the Prolog language models which do not provide such kind of abstractions.

The current state of art clearly shows that the answer to all of those problems is not easily found by providing a single, supposedly unifying programming model, and so a lot of research is still to be made in this direction. Our approach consists of providing a two-layered abstract architecture for parallel and distributed logic programming:

- **Low-level**: this programming layer keeps the Prolog language programming style, and extends it towards a parallel and distributed setting, in the form of explicit interface control predicates.

- **High-level**: this layer supports high-level language models which typically extend classical forms of Horn Clause logic with constructs for parallelism,
communication and non-determinism, that are introduced through rigorous semantics definitions in the setting of the corresponding extended logic framework.

While the low-level offers the possibility of programming using the Prolog style, it also allows some degree of control of operating and distributed system abstractions that may help in the process to processor mappings, and thus contribute to more efficient implementations. On the other hand this layer provides a basic platform upon which one can possibly implement specific high-level models as the ones mentioned for the above layer.

The higher layer aims at promoting a more clear and transparent model supporting a more abstract view of the distributed or parallel system, such that application decomposition and forms of cooperative computing can be expressed without regard for the low-level details of real execution.

In summary our current approach aims to establish a well-defined system-level layer accessible from regular Prolog programs, and provide that layer as a basic platform for the user to develop his own higher-level abstractions. In the long run we hope this will contribute to the development of a reasonable diversity of special-purpose high-level toolkits which may be available for particular application domains. An interesting aspect of this approach relates to the provision of low-level support (but accessible at the extended Prolog layer) for monitoring the parallel and distributed execution. We are using an event-based approach that collects relevant information (and stores it using a meta-format, i.e. user-definable, for the event traces), to be analyzed and processed, at a high-level, according to the semantics of a specific model and depending on the intended user view: for performance evaluation, for visualization of the dynamic program behavior, or for debugging.

The organization of the paper is as follows. In section 2 we outline the complete architecture: identify its layers, and define the requirements and fundamental abstractions provided by each layer.

Section 3 discusses a higher level layer where specific logic agent structuring abstractions are identified: process grouping, message-based communication and shared views among group members. This is one of our current approaches for exploiting group-oriented abstractions, but as explained above other abstractions and/or specific language models are amenable to implementation in this layer.

Section 4 discusses basic mechanisms for concurrency and communication which are offered by the low-level extended Prolog layer: processes and threads, ports and term queues and briefly discusses the implementation issues towards the support of the above mentioned structuring abstractions. Section 5 briefly refers prototype implementations of the aspects which were discussed in 4. Finally the paper presents some conclusions drawn from our experimentation and points ongoing research directions.

2. A MULTI-LAYERED SOFTWARE ARCHITECTURE FOR PARALLEL PROCESSING

The global architecture of the environment is organized into several functional layers but these do not impose a rigid hierarchy in the sense that any upper layer may directly access functions from a non-adjacent lower layer:
- High-level parallel and distributed logic layer
- Low-level extended Prolog layer
- Virtual system layer (PDVM)

As the two upper layers have already been presented (and are further detailed in sections ahead), in this section we summarize the purpose of the Virtual system layer or Parallel and Distributed Virtual Machine Layer. This supports the intermediate level system platform which provides an unified model of concurrency, communication and synchronization abstractions, and system support for process grouping with several semantics concerning consistency preservation, causality and uniformity in the event orderings.

The PDVM layer has two main goals:

- to provide a suite of primitives that are accessed by any upper layer in the environment, in order to implement the extensions to the high-level base languages as mentioned in section 1. This protects such language extensions from any dependence upon the existing heterogeneity at the hardware and operating system levels, unless specific visibility of low-level features is explicitly desired by the language or tool developer. This visibility concerns some degree of monitoring and control of real program execution, including mapping, load-balancing, performance and debugging issues. For example, an application-specific topology may be specified to be supported by a configurable multicomputer architecture, or specific process to processor mappings may be requested to this layer.

- to map the basic model of interface primitives onto each particular hardware and operating system real environment.

The details on the PDVM layer supported abstractions are beyond the scope of this paper. As a final note to this section we should emphasize that our main goal is not to promote yet another parallel and distributed programming interface, but instead to provide a flexible testbed which may be used to experiment with improved models for programming languages and development support tools.

3. DESIGN OF A HIGH-LEVEL GROUP-ORIENTED DISTRIBUTED LOGIC LANGUAGE

In this section we discuss the basic concepts that are behind the two levels of structuring for a parallel and distributed logic language. We do not discuss here the language syntax and semantics in detail [Bar94] [CB95]. These structuring levels are currently integrated into a specific distributed logic language that is being developed by the group and which is an example of the kind of high-level abstractions that can be implemented on top of the low-level layer.

Besides general principles such as modularity, information hiding and reutilization of components, the need for program structuring is due to the great complexity of concurrent computations with multiple processes evolving in an autonomous way but cooperating through well-defined interaction points. Three structuring levels are considered: modules support the structuring of the clause space, agents support
encapsulation of knowledge and computation, and groups support the structuring of the space of agents.

Agents are program units which encapsulate a well-defined internal knowledge that is specified through a set of imported modules (finite sets of immutable clauses), and have a well-defined communications interface, as well as a specific behavior that may exhibit some internal concurrency to the agent. The following components define an agent:

- The Clause context: this specifies local knowledge to the agent and may be interpreted as the program that is associated with the agent.

- The Communication context: this specifies the interface that this agent makes available to other agents in its outside environment. It defines a set of predicate names which may be invoked by other agents.

- The Behavior: this specifies how the agent reacts to the invocations of its interface predicates, as well as how it manages a possibly defined internal state which is kept under the supervision of an eternal process.

At this level of structuring, agents have unique names in a global name space. They communicate through several forms of remote interface predicate invocation, with a synchronous semantics. One-to-one or one-to-many communication schemes are supported by the language.

Forms of dynamic reconfiguration of multi-agent systems are very important in several classes of problems where agents cooperate by temporarily enrolling themselves in joint "task forces". So we have defined a dynamic structuring unit, at the program level, which is based on groups of agents. A group encapsulates a set of agents which are therefore hidden to the outside environment in order to perform a collective task in a transparent way. A set of well-defined interface predicates is provided for communication with a group.

Agents may dynamically enter and leave a group and so benefit from its attributes. Agents within a given group may share a common state - the group state - which is defined as a finite set of ground atomic formulas. They also share a set of immutable internal predicate definitions - the group clause context - which are known only to the group members and are used to support internal group functionalities. The group view is defined by some information that is kept consistent across all group members, and includes several components: the group current membership, the group state, and the group history, i.e. the set of events which are perceived by the group members in a uniform way. Relevant group events are the invocation of a group interface predicate, changes in the group membership, and changes in the group state.

4. A LOW-LEVEL EXTENDED Prolog LAYER FOR PARALLELISM AND DISTRIBUTION

This layer provides several primitive mechanisms as extensions to Prolog that may support the flexible construction of higher-level abstractions for parallelism and distribution control. We regard this as a low-level layer as these extensions are defined in terms of a set of Prolog system predicates.
The following description of the functionalities provided by this layer is necessarily succint due to space limitations. Further detail is the subject of more specific papers on this layer [CMC87][Mar92][Lou94].

4.1. Processes and Threads

A process provides the support for execution of Prolog goals. It has the following associated components:

- The process name: this uniquely identifies the process in the universe of processes within a given computation.

- The process program: this is a set of Prolog clauses; it only differs from a conventional Prolog program in that it includes additional system predicates for concurrency control.

- A set of input/output channels: these channels establish the communication of the process with the outside environment; they allow connections to conventional i/o devices of the terminal and file types, or they may be connected to communication devices known as interface ports which are described in a subsection below.

- A set of internal threads: each thread corresponds to a virtual Prolog executor which is responsible for a Prolog computation, given a top goal and the process program, following a sequential execution model, and using depth-first search with backtracking.

- A set of internal term queues: each queue is a FIFO-based communication device, used for term passing among the threads which are local to a particular process. They also allow to establish several forms of thread synchronization.

The rationale behind this model is briefly explained in the following. A process plays, in a certain sense, a role of a structuring device in a distributed computation. As a matter of fact, it encapsulates the execution contexts for, possibly concurrent, evaluation of goals, performed by the threads which are defined within the process. Although each such thread has a private execution context concerning flow of control and work areas, they all share the same set of program clauses (the enclosing process program) as well as the set of input/output channels and internal term queues which are associated with the process.

Processes and threads are dynamically created by the invocation of the following predicates (several options are omitted for simplicity):

process_spawn(+Input, +Output, -ProcessId)
Create a new process with associated Input Output channels. Returns an unique process identifier in ProcessId.

process_kill(+ProcessId)
Kill the process identified by ProcessId.
thread\_spawn (+\ TopGoal, −\ ThreadId)

Creates a new thread of control in the current process to solve $\text{TopGoal}$. Returns a unique local thread identifier $\text{ThreadId}$.

thread\_kill (+\ ThreadId)

Kill the thread identified by $\text{ThreadId}$.

Initially there is a single process, corresponding to a conventional Prolog session. So the functionality of a conventional Prolog environment is fully preserved, as each process has by default a single initial sequential thread. Whenever a new process is created, system resources are allocated and an initial thread starts execution within the process, running a sequential Prolog executor cycle, that is, it hangs on the specified input device waiting for a top goal. Typically, the first thing to do is to send a consult directive with a file specification, through that input device, in order to make the initial thread get to know (and internal install) the process program. It is possible to specify terminal names, file names or port names for the Input and Output arguments of a new process.

Internal threads to a process are created by invocation of the above predicate within that specific process. Each thread starts by evaluating its given top goal, and may use communication predicates to interact with other threads (using term queues in the same process - see subsection 4.2) or other processes (using their interface ports -see 4.3), or other devices (using the corresponding I/O channels).

Each process has its own thread scheduler. All threads have the same priority and the thread scheduling policy, is, by default, preemptive with timeslicing.

4.2. Intraprocess communication

In conventional low-level thread-based models, the common address space of the enclosing process allows communication through shared-memory, e.g. threads may share global variables defined by the process. In a logic-oriented model several possible interpretations may be based on this concept:

- To define a shared set of clauses within each process which is accessible to the threads through additional predicates. These predicates are variants of the typical assert/retract predicates, coupled with some kind of thread synchronization control, or they may be inspired in a Linda-like model [Bro90][Cla93][BJ93]. We prefer this latter approach, in agreement with multiple experiences, but we find this is a more attractive model to offer at the higher-level layer, possibly as a toolkit (or even embedded into the semantics of a language model).

- To introduce non-logical global shared variables within each process, and provide some classical form of thread synchronization, e.g. using mutual exclusion locks. This approach is not so attractive from a logic-oriented programming style.

Threads in the same process can communicate through devices named term queues. A term queue has a name with a restricted visibility to the threads of the corresponding process. It can be manipulated by the following primitives:
queue_create(+QueueName)
Create a new term queue with name QueueName.

queue_put(+QueueName, +Term)
Asynchronously insert term Term in queue QueueName. There is a “rendez-vous” version of this predicate.

queue_get(+QueueName, -Term)
Retrieve term Term from queue QueueName. Blocking and non-blocking versions are provided.

queue_delete(+QueueName)
Delete queue QueueName.

queue_empty(+QueueName)
Succeed if the queue QueueName has no terms; fail otherwise.

queue_select(+QueueNameList, -QueueName, -Term)
Non-deterministically retrieve a term from one of the queues in QueueNameList. Returns the retrieved term in Term as well as the queue name from where it was retrieved.

In our experimentation, we are assessing the mentioned primitives for definition of processes, threads and term queues for the programming of applications where multiple autonomous reasoning agents may concurrently evolve, each agent being represented by an individual process. Possibly, many applications just require each process to be single-threaded and compatible with a conventional sequential Prolog executor - and this is our option by default. In such applications, one only needs to provide a form of interprocess communication so that individual processes may cooperate - see section 3.3 below. However, if one so desires, internal concurrency may be exploited within each process. This allows a more clear programming model in general situations where internal deductions (or i/o actions) performed by a process are amenable to concurrent execution. The role of term queues in such situations aims at supporting forms of tight and efficient thread cooperation.

4.3. Interprocess communication

Threads from a process can send messages to ports. Ports are created associated with a process such that only its threads can receive from those ports.

Ports have unique global names in a name space that is associated with a set of processes. By default, this is the universe of processes that are defined by the program (but see more on this below, concerning process groups). This means that, once a given port is created, every process in that referred set may communicate to that port. However, except for the port creator, all other processes are only allowed to send terms to a port. We use ports as well-defined entry points to a given process. We argue this is a good approach to promote modularity in a dynamically evolving system, because it supports the definition of an input interface that may be used by the processes to contact each other.

The primitives for port-based communication are:

port_create(+PortName, -Port)
Create a port with name $\text{PortName}$. Returns a local unique port identifier in $\text{Port}$.

$\text{port_open(+PortName, -Port)}$
Open the port with name $\text{PortName}$ for writing. The port must already exist.

$\text{port_close(+Port)}$
Close the port $\text{Port}$ previously opened for writing.

$\text{port_delete(+Port)}$
Delete the port $\text{Port}$ previously created.

$\text{port_send(+Port, +Term)}$
Asynchronously send the term $\text{Term}$ to port $\text{Port}$. A “rendez-vous” version is also available.

$\text{port_receive(+Port, -Term)}$
Receive a term from port $\text{Port}$ and return it in $\text{Term}$. Blocking and non-blocking versions exist.

$\text{port_select(+PortList, -Port, -Term)}$
Non-deterministically receive a term from one of the ports in $\text{PortList}$. Returns the received term in $\text{Term}$ as well as the port identifier from where it was received.

By default, each port acts as a passive device in the sense that it requires an explicit invocation of the above primitives in order to access/modify its state. There is however the possibility of associating a given port with a user-defined predicate that will be activated as soon as a message is delivered at that port. The execution of this predicate is performed in the context of a thread that is created within the corresponding process. This mechanism allows the asynchronous handling of inter-process communication events. For greater flexibility, a given port may operate in the default mode or in this so-called urgent mode, depending on the invocation of the two predicates below:

$\text{port_urgent(+Port, +Goal)}$
Define predicate $\text{Goal}$ as a handler for the arrival of terms at port $\text{Port}$. $\text{Goal}$ must be a predicate with arity one and the received term is passed as its argument.

$\text{port_disable(+Port)}$
Disable any handler previously set by $\text{port_urgent/2}$ to port $\text{Port}$, i.e. revert to the default port mode, as of its creation.

### 4.4. Process groups

The group abstraction supports the dynamic definition of process groups. Each group is a set of processes that define a logical unit in the sense that all its members perceive the same event orderings as a whole. Relevant group events are communication events and group membership modifications. In the design of these group functionalities we were inspired in the Isis system [KBS87][BJ87][Bir91], and at a first stage we have simply designed and implemented an interface to that system, accessible from the Prolog language [CG91][Mar92]. Then the design evolved towards our port-based model, but our work still relies on many aspects which are
common to the Isis framework. Due to space limitations we just provide a summary of the supported functionalities.

The group has a unique name and defines a new level of organization in the previous flat universe of processes. This has implications concerning the visibility of the port names which become now restricted to the scope of the groups to whom each process belongs.

A group is created by the predicate \texttt{group:crea}te that specifies the group name and several group attributes, namely its interface with the outside environment. A process joins a group by invoking a predicate called \texttt{group:jo}in/2 where the first argument specifies the group name (a Prolog atom) and the second argument specifies several options not detailed here. A process leaves a group by invoking a predicate called \texttt{group:leave}/1.

Communication within a group may take place by having the processes invoking the above predicates, for point-to-point communication. They can also invoke other communication predicates which provide term multicasting to the processes in the same group, more exactly to corresponding sets of ports: forms of atomic uniform broadcast, and of causal broadcast are supported, based on corresponding functionalities of the Isis system. All group members have the same view of the current group membership, in the sense that each member sees the same sequence of group modification and communication events. This is a very nice aspect which allows processes to dynamically enter or leave groups without originating inconsistent views.

An external process may communicate with a group, by sending a term to special purpose group interface ports whose definitions specify the internal group behavior concerning message delivery to group members.

\subsection*{4.5. Comments on the suitability of the low-level layer}

The low-level extended Prolog layer has been under an incremental design process for some time, and is still evolving. We currently feel it provides reasonable flexibility for the development of high-level models or specific parallel logic programming toolkits. The first example of use of a previous version of this layer \cite{CMC97} was for the implementation of prototypes of the Delta Prolog language \cite{PN84,CFP89,CMCP92}. Further experimentation confirmed this claim, as we have used this layer to implement a toolkit supporting parallel evaluation and distribution of Prolog goals, with user-defined strategies for requesting and producing solutions to the mentioned goals \cite{Lou94}. Other experiments which have used the low-level layer include applications in Robotics and in the control of industrial processes, where several processes (some of them written in Prolog some in C) cooperate in the supervision of numerical control machines \cite{BCGa90}.

Processes, threads, ports and groups are very good abstractions to experiment with the design and implementation of parallel and distributed logic languages which include forms of static or dynamic structuring of the program components. We also are using this layer to support the abstractions in the group-oriented distributed logic language that we briefly discussed in section 3. The implementation relies on the definitions of suitable mappings from the high-level language abstractions onto the low-level layer:

- Agents in the language map onto processes in the low layer; internal concur-
rency within an agent is supported by the threads in the respective process; internal communication and synchronization is supported by term queues and scheduling control.

- Interface predicates of each agent are mapped onto ports of the corresponding process, such that the remote invocation is converted into the sending of a message (term) to a port, which subsequently triggers the invocation of a predicate that supports the intended agent behavior.

- Groups of agents are mapped onto groups of processes, and their interface predicates are mapped onto the group interface ports of the low level layer.

- Shared state by the members of a group is supported by hidden group managers that preserve the consistency of the corresponding set of formulas, relying upon the above mentioned consistent event ordering guarantees of the low-level layer.

5. CURRENT PROTOTYPES OF THE LOW-LEVEL LAYER

We have been for some time developing prototypes that extend Prolog with primitives for parallelism and distribution. These extensions are defined in terms of system predicates. The experimentation with these prototypes has contributed to the definition of the extended Prolog layer.

Each prototype implements several aspects of the described model for the extended Prolog layer. In the following description we give a succinct description, in order to illustrate our experimentation with the multiple components of the model.

5.1. Single-threaded Prolog programs with port-based communication

These prototypes support a system of communicating sequential Prolog processes. Each process corresponds to an instance of an interpreter (an extended C-Prolog interpreter in the first versions [CMC87] [SC90]) or an instance of a WAM-based machine [Dia90]. The latter approach is the one being used in the most recent versions [LB91] [Lda94]. Extensions for process control and port-based communication were included in all of these prototypes but they were implemented on top of distinct operating system and hardware environments, ranging from UNIX monoprocesors, UNIX LAN, an in-house developed PC-hosted Transputer-based parallel architecture [BCGa90] [SC90], a distributed-memory multicomputer (Meko Computing Surface CS-1) under the control of the Trollius operating system [BRDM90].

5.2. Multi-threaded WAM-based prototypes

Two prototypes were developed to experiment with the flexibility of a multi-threaded Prolog abstract machine, based on modifications to the WAM [War83]. Both prototypes run under a UNIX (single-process) environment:

- A Delta Prolog abstract machine: the SB-Prolog system [Deb88] was extended with threads with the specific purpose of supporting new basic instructions as required by the semantics of the Delta Prolog language [CC92].
• A Multithread WAM: the NanoProlog system [Dia90] is a minimal Prolog implementation, developed specially for research and teaching purposes, however keeping performance as a primary issue. We have extended it towards the support of a self-contained, more flexible prototype that supports threads and term queues [Mar92] according to the extended Prolog model.

5.3. Distributed Prolog processes

Several prototypes have been developed to support Prolog interfaces to existing parallel or distributed programming interfaces:

• Prolog/Isis interface [KBS87] [CG91]: The use of this interface has shown us the usefulness of group-oriented abstractions in a logic-oriented model.

• Prolog/PVM interface [MC95]: The PVM system [G892] [Sun90] supports parallel programming on heterogeneous computer networks. It offers communication and synchronization facilities at task (process) level and allows a certain degree of control at the physical (machine and networking) level which have inspired some of the functionalities of the low-level model. Both prototypes have played important roles in the support of the previously described high-level and low-level models. Besides, they offer full access to the Isis and PVM functionalities, from within a Prolog program, allowing the exploitation of a broad range of applications for parallel and distributed Prolog systems.

6. CONCLUSIONS

We have discussed a layered approach to the design and implementation of a software architecture supporting parallel and distributing logic programming systems. Due to our long term research in parallel and distributed logic programming, we found a strong motivation to design a flexible system platform that could hide the peculiarities of the hardware and operating system architectures. The design of the PDVM layer is still ongoing work, as it is very difficult to conciliate the power of PDVM abstractions with efficient implementations [MC94]. The two-level extended PrologG layer further eases the task of implementing higher-level logic-oriented layers, besides providing concurrency primitives that may be used on their own. Concerning the high-level layer, our current work focus on the complete design and implementation of the mentioned structuring abstractions, upon the services offered by the lower layers.

In future work we will continue to improve the three layers in the following directions:

• to add monitoring, performance evaluation, visualization and debugging tools at all levels;

• to assess the two upper layers with realistic AI applications in two current projects: PROLOPPE (funded by JNICT the portuguese research agency) and PADIPRO (funded by an DEC grant);
• to assess the PDVM layer concerning its flexibility, performance and adaptability by adapting it to other heterogeneous hardware platforms a Parsytec MultiCluster machine and a Digital Alpha-based farm (PADIPRO project).
On the other hand we will assess PDVM against its use for the support of distributed programs composed of non-logical languages (namely extended C++ and Fortran) and logic components.

REFERENCES


