In APPIA-GULP-PRODE 2002, Madrid (Spain)

A Configuration Framework for Distributed Logic Applications

Jesús Correas Fernández
Francisco Bueno Carrillo

CLIP Group, Technical University of Madrid
Campus de Montegancedo
28660 Boadilla del Monte
Madrid - Spain
{jcorreas,bueno}@fi.upm.es

Abstract. This paper proposes a configuration framework to develop and deploy large distributed logic applications. The proposal extends the concept of "active modules" used in the Ciao Prolog development system, isolating code from configuration and deployment issues. This approach allows local development of distributed applications, disregarding architectural and topological issues, and later configuration (and re-configuration) of the application execution structure without changing the source code. This paper also presents the design guidelines of the proposed architectural structure of distributed applications that use this framework.

Keywords: Logic Programming, Distributed Systems, Configuration Languages.

1 Introduction

As computer systems are able to afford more complex problems, the interest in the development of distributed applications grows, since the needs for computational resources are increasing continuously. In another scenario, the ubiquity of network connected devices with growing processing power imposes the use of this kind of applications, generalizing their use in almost any area. Nevertheless, programming distributed systems is a rather complex task.

This paper proposes a configuration framework based on the module system and distributed programming infrastructure developed on Ciao, a next-generation logic programming development environment that includes a full-featured module system that considers modular incremental compilation, global analysis and language extensions, in addition to the functionalities the modularization provides to the programmer [CH00].

Our proposal is an extension of the "active modules" feature of Ciao [CH95,CH96,CH01]. An *active module* is just a module which will run as a separate process. Thus, an active module process acts as a server for the predicates exported by the module. A module is declared to be *active* in the client module, and is compiled in a special way to become "active". It is then run separately.

Our aim is to separate the development part from the configuration part of a distributed system, and centralise the configuration in a single point. In this envisioned framework, the programmer does not need to take care about which parts of the system will be executed remotely, or how to define component interfaces. Once the system meets the functional requirements (or while programming), different configurations can be tested with no changes in the source code. Of course, after selecting a specific configuration, the source code may be fine tuned in order to improve the remote components communication. To accomplish these tasks, the configuration language must be expressive enough to face possibly multiple configurations of a complex system.

Our target developer focuses first in the program that solves the problem faced, and afterwards in configuring it so that it runs distributedly. If the application components are distributed in nature, the programmer can simply map them into modules of the program and continue the development. Our target applications do not require dynamic redistribution or reasoning about its own distributed execution. Thus, there is no provision in our framework for goal-level distribution language constructs.

In the following section, we present the proposed structure of a distributed application developed using our approach, and in the next section our configuration language and its features. In section 4 implementation issues are discussed, and section 5 shows several issues related to programming distributed applications. Finally, section 6 presents related work, and section 7 our conclusions and future enhancements.

2 Distributed Application Architecture

The applications developed using our approach will work using standard architecture and protocols, based mainly on the use of name servers, that provide component creation and handle physical addresses.

The whole distributed platform is composed of a number of networked computers (nodes) on which a name server is running and listening on a communication port. These (node,name server) pairs are named *places*. Every name server knows the locations of the others, to connect with them if an application requests execution of an active module in any other node. For the moment, we abstract away on how to set up this scenario (this is discussed in Section 4). With this basic architecture in mind, we present in the following the actions that are involved in the execution of a distributed application via the name server network. These are represented in Figure 1 and are as follows:

1. When a distributed application starts, it connects with its local name server to get the physical addresses of the name servers it needs for execution. To

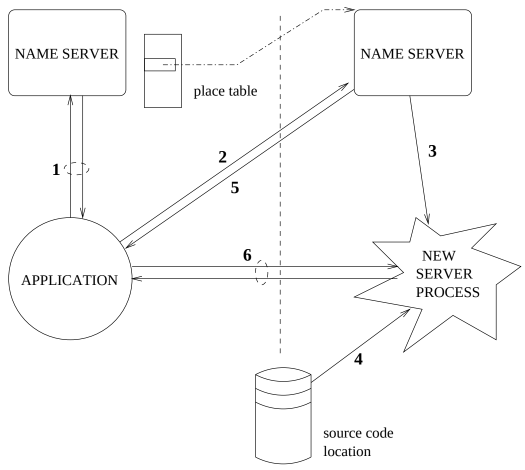

Fig. 1. Distributed application architecture

do this, it uses a predefined port number (that the programmer does not need to know because is fixed and known by the local LP system on which both application and name server are running) to communicate with the name server.
2. The application requests the creation of server processes for its active modules to the corresponding (remote) name servers. The application attaches to this request the location where the source code of the active module is stored (this location must be accessible from any node that participates in running the application).
3. Each name server then creates a new server process to accomplish the application request.
4. The newly created server process loads dynamically the active module code from the source code location, and prepares it to receive requests.
5. The remote name servers return to the application the physical address of the newly created server process, so it can start requesting goals.
6. Once the application has the physical address of the active module process, requests to it the execution of goals. The active module server process executes them and returns the results (and possibly the application requests backtracking to get more results). This action is repeated for each goal that must be executed by the active module.
7. When the client side of the application terminates, the active module processes terminate their execution accordingly.

The previous action list reflects a pure client/server relation between the client part of an application and a server component (using intermediate name servers). This scheme is a simplified version of the actual communication. The mechanism may be more complicated if one active module calls predicates of another active module, and the later calls back predicates of the former, creating a call cycle. The execution mechanism is not reduced to a client/server communication. There are several complex problems that the implementation has to deal with to build a generic architecture. They are discussed in Section 4.

3 Distributed Configuration Language

To describe complex distributed systems a expressive enough configuration language is required. The basic elements we need to use are the following:

`application(Name,Main,SourceAddr)` defines the application name and a source address (usually an URL or file system directory) where the application code is accessible from all the nodes that will execute any application component. Using a common place for every node avoids copying the code, making the application maintenance easier (only one location must be updated when a new version is deployed). `Main` declares the module that contains the application starting code. This is the *main* module of the application, which identifies the application itself. It is this object to which `Name` refers as a logical name.

`place(Place,ServerAddr)` defines a *place of execution*: a name server address. This is the link between the logical component architecture and the physical application deployment. The second argument reflects the name of a networked system.

`active_module(Module,Place,Mode)` is used to associate an active module to a place. When a predicate of `Module` is called (from any other module of the application), the corresponding goal is executed in `Place`. The third argument indicates whether the server process that executes `Module` will share it with other applications (with `Mode` instantiated to `public`) or with just other modules of this application (`Mode` instantiated to `private`). The latter will create a different server process for each application that uses the active module.

For managing complex applications, the configuration language should be powerful enough to handle a variety of options, multiple configurations, flags, and so on. In short, a complete language with a declarative look is desirable, since the configuration problem is itself declarative: what one wants is a language to declare the distributed structure of the application. The best solution for this is then to use for configuration the same language than for programming. Thus, we propose the above to be predicates, so that they can be used flexibly for configuration files.

For example, let us suppose we have an application for querying a database using a natural language interface. This app could be composed of (among others) four major modules: a module that handles the dialogue with the user (`dlg_processor`), a module for natural language processing (`nl_processor`), another module to deal with the queries to the database (`db_query_manager`), and a main module that performs initial tasks and starts the dialogue. A possible implementation could be as follows:

```
:- module(nldi,[main/0]).

:- use_module(dlg_processor,
              [new_dialogue/1]).
main:-
       ...application setup...,
       repeat,
         new_user(User),
         new_dialogue(User),
         fail.
```

```
:- module(dlg_processor,
          [new_dialogue/1]).
:- use_module(nl_processor,
              [nl_process/2]).
:- use_module(db_query_manager,
              [db_query/2]).
new_dialogue(User):-
       repeat,
         get_user_sentence(User,Sent),
         nl_process(Sent,Semantics),
         db_query(Semantics,Result),
         conform_response(Result,Resp),
         respond_user(User,Resp),
         fail.
```

```
:- module(nl_processor,
          [nl_process/2]).

nl_process(Sentence, Phrase):-
    ...
```

```
:- module(db_query_manager,
          [db_query/2]).

db_query(Sem,Result):-
    ...
```

This code just takes account of the functionalities of the application, not addressing any distribution related issues. In order to turn it into a distributed app, we only need to use a configuration file. For instance:

```
:- use_package(ams).

main(nondist).
main(local) :-
    common(localhost,localhost).
main(production) :-
    common(natlang_node,database_node).

common(NL,DB) :-
    place(NL,_),
    place(DB,_),
    application(nldi,nldi,"http://www.clip.dia.fi.upm.es/src"),
    active_module(nl_processor,NL,shared),
    active_module(db_query_manager,DB,shared).
```

This sample file contains three possible configurations of the previous application (named `nldi`). Different configurations are represented as separate clauses of `main/1`, which is the standard startup predicate of any Ciao Prolog program, and the arguments of which are the parameters given to the invocation of the program when run. The first clause just configures the app to be executed as a non-distributed program (as if there were no configuration file). The configuration named `local` prepares the application to be run in a distributed fashion, but keeping the active modules in the local machine (this configuration may be useful to test distributed components with no need of distributed infrastructure at all). Finally, the "`production`" configuration sets up the code to be executed in three different machines (the local machine will run the code of the local part of the application, while `nl_processor` and `db_query_manager` will be executed in `natlang_node` and `database_node` respectively).

This example also shows that every Prolog goal may be added to the configuration file, making it quite powerful (e.g., using auxiliary predicates such as `common/2`, library predicates, or even other Prolog modules).

The first line indicates that this source file uses the `ams` Prolog language extension ("package" in Ciao), that brings the semantics of a configuration file. It thus declares that this file is a configuration file for AMS (Active Module Service). Operationally, this means that when this file (let it be `config.pl`) is executed, the code of the application modules will be set up so as to guarantee the selected configuration. The user can select the desired configuration calling:

`config nondist` or `config local` or `config production`

In this way (helped substantially by the language extension features of Ciao), it is made possible that the configuration language be the programming language itself. The configuration file is just a *precompiler* for the application, which is run by the standard interpreter of the Ciao system.

4 Implementation Issues

The proposed architecture and configuration features involve a number of issues that will raise during implementation. The following sections include what we understand that are the most important items to deal with.

4.1 Setting Up a Name Server Network

The architecture on which the configuration framework is based requires name servers to perform many important operating tasks. In section 2 we have shown the main actions name servers have to do when an application uses active modules. Although in the previous configuration language definition the name server addresses are embedded in the configuration file of an application, address-independent application code could be quite interesting: applications could be configured with no physical network addresses; these addresses would be given to the local name server, in order to establish dynamically the physical server addresses of the application (the actual nodes in which it would be executed). This feature requires the interconnection of the name servers that may be used for executing an application, making up a *name server network*. This network is also very important to enable public active modules, as mentioned in Section 4.5.

To build a name server network, every participant must know about the others: their addresses and what services each member offers. This can be easily made including a communication protocol between name servers. When starting a new name server, an already existing name server address could be provided to connect it to the existing name server network. This name server then sends to the newly created name server the information it has about the network structure. It sends also the new name server to the rest of the network.

This scheme is very useful for future enhancements of the framework. As an example, a name server could offer a certification service (as a public active module) that guarantees the active module code safety, either checking enclosed certification information or analysing the active module code if it has not been certified yet. When any name server receives an active module creation request, checks its code safety using that certification service. Name servers can also share interesting information about modules and applications, propagating it across the network and enabling dynamic application reconfiguration, in response to node and network load, system crashes, and so on. Name server network should become an element of central interest in the operational part of this framework.

4.2 Setting Up the Application

The configuration code presented in Section 3 must be interpreted before the application is compiled, behaving just like a precompiler, in order to prepare the application modules that are declared to be active so that they are executed as separate processes and communicate with the main application process.

The adopted solution is to create (for each active module) new code that replaces the original module code. This new code only performs the communication between the caller module(s) and the active module process. This *dummy* module contains the same exported predicates than the original module, but they just call remotely the actual predicates residing in the active module. A similar approach was taken when designing the active module extension of Ciao.

When compiled, the module(s) that call(s) an active module will be linked with the dummy module instead of the original module. During execution, all calls to active module predicates are redirected by this module to the process that acts as the active module server. It is this dummy module which is in charge of recording the process server address (returned by the name server that created it) and redirect to it all calls to the exported predicates.

It is also a precompilation task to prepare things so that the application is started as configured when run. To accomplish this, dummy modules contain code for communicating with the local name server and requesting the activation of the corresponding active module server process in the designated place. For this purpose, `initialization` directives are used, so that the activation of the processes occurs when the application is invoked. The active modules themselves do not need any change to work in a separate process server, because the process server already includes all the code to do the communication work.

And finally, the execution of the configuration file should also make a copy of the active module source files in the public source code location given in the `application` declaration. This location should be accessible from the nodes where the active modules will actually run.

4.3 Running the Application

Starting from the guidelines shown in section 2, the main issue to face about running a distributed application is to decide where the actual network addresses are located. The previously defined distributed configuration language needs to specify network addresses for source code location and execution nodes. This approach have an important drawback: if the network changes for any reason, or the application source code is moved to another network, the configuration file will produce incorrect results. To solve this problem, the configuration file can be split in two different files: one for configuring the application, but leaving addresses uninstantiated (as `place/2` calls in the previous example), and the other to reflect the relation between places and addresses. The latter file must be provided to the local name server, in order to update its application tables. This improvement keeps the configuration information centralised in two files. In addition, when the network structure changes the application does not need to be recompiled, as the network structure is not embedded in it. Name server handling of network addresses of its local applications is also interesting to enable dynamic changes (for example, for load-balancing). Nevertheless, if the actual addresses are stored in the name server, unique place and application names must be provided to the name server for differentiating them from other applications. A name space must be established.

Other important issue is that of separating different runs of the same application. If the framework is not aware of this issue, stateful active module processes will be shared between several runs of the same application, producing unpredictable results. Our solution is that active modules are attached to a run of an application: the initialization of the application execution starts up a (separate) server for the active module. Servers are shared only if the user declares so, which is done via the `Mode` in `active_module/3`. Thus, we extend modes to include also `shared`, allowing to share a server with any run of the same application.

4.4 What Does an Active Module Consist of?

Although the concept of active module has not been defined precisely, in the previous sections we have considered that it is alike the concept of distributed component in other contexts. However, the main difference with components is that in this framework there are no special definitions for the component interface: the interface of an active module is just the interface of the underlying module.

This point of view implies an important pitfall to solve, not existing in other approaches: deciding which code will be executed in the remote place, aside from the active module code itself. That is, if the active module uses other modules, deciding where to execute their code. If these modules are also defined as active modules, they will be executed in the places specified in the distributed configuration file. The problem arises when there is no explicit declaration on where to execute the code of a module used by an active module. The solution to this problem has several alternative approaches:

- The first approach is to execute the code in the same place that the caller (active or not) module is executing. Although it could be a good solution, local execution has an important drawback. If the called module is being used by different active modules, or by the start up part of the application (the "client" side) and an active module, a copy of the called module will be created in each process space. If the called code contains state, different state will be kept in different process spaces. This approach cannot be accepted directly because may change the semantics of the application.
- Next alternative could be to convert automatically the called module in an active module. This approach solves the problem found in the previous approach. However, there is no information about which place the new active module should be executed in. And, more important, the final application structure will not reflect the intended structure specified in the configuration file.
- The third approach is to perform a previous program analysis to detect how the modules not explicitly declared as active modules can be distributed between the different components of the application. For example, if a module is used only by an active module, that module could very well be loaded in the process space of the active module instead of converting it into another active module. This analysis should work on the module dependency graph

in order to detect useful subsets of modules that could successfully be loaded locally in an active module process space.
- Next alternative consists in loading all the modules not declared explicitly as active modules in the start up part of the application (the "client" side). If an active module uses any other (non active) module, it will use the "client" side as an active module in turn. The "client" side must then be loaded from the begining of the application as an active module itself. This is the most conservative approach because it supposes that the application will run mostly in the local process, and only sporadicly it will need an active module.
- Finally, a variation of the previous alternative can take advantage of the structure of the low-level communication mechanism with active modules, implementing a callback protocol. If a predicate of an active module is called, and its execution makes a call to a predicate of another module (which is not active, and thus its code is loaded in the caller module process space), it can use the existing communication to call back the later predicate to the caller module. The approach is the same as before, but in this case there is no need of starting the "client" side of the application as an active module: the existing communication is used in duplex form, allowing calls in both directions, thus making the communication protocol more complex.

We are currently working towards the last solution. However, as future work on this concern, another solution to this issue can be the evolution of the distributed configuration language to provide a framework for defining compositional relations between modules, so structured components can be expressed. It could be done simply as a generalization of the module system of the language. Although such alternative is out of the scope of this proposal, it would be an extremely powerful tool for large logic applications.

4.5 Public Active Modules: Towards Agents

Lastly, a quite interesting implementation issue is to provide public active modules, shared by any application that uses them. A public active module can be started as a stand-alone application that publishes its interface in the name server network. Each application that uses that active module just connects with the already existing active module process server, instead of creating a new process server for it.

For example, a natural language processor could start in a network node publishing its interface in the local name server. When an application needs the use of a natural language processor, it simply declares this as another module in the program, with the corresponding interface. The programmer does not care at all about whether this will be distributed or not. Transparently to the programmer, when the application is executed, it requests to its local name server the service of natural language processing. The name server then provides it the address of the public active module (received from the name server network), instead of creating a new natural language processor process. The goals for natural language processing will be executed in the existing process space, thus taking advantage of such a complex process, without any burden on the programmer.

This could be the first stage in the implementation of a public agent that broadcasts its services on the network. The application that uses such service only needs to know the application program interface of the public agent. In a more sophisticated approach, application and agent could use a standard language for communicating agents. In any case, a key issue in public agents of this kind is the specification of a global naming scheme that allows to address univocally each of the possible distributed agents.

5 Programming distributed applications

Existing distributed application development systems allow building distributed apps specifying in the client code the modules to be used remotely, specifying the location of those servers, compiling the components properly, and registering and starting the component servers before the client application starts execution. This approach can be very useful in applications that involve a small number of components and in which the client and the server are clearly separated. However, as the complexity of the applications to be built grows the problems that have to be faced up (and that are not directly related to the programming task itself) spend a considerable part of the development effort. The main pitfalls are discussed in the following.

First, when a distributed application or prototype is being developed, the source code of the client modules must reflect the distributed nature of the system. This fact has two consequences: first, the general structure of the distributed system has to be set before the programming phase begins, since later architectural changes generally convey a very expensive reprogramming effort; and second, this implies the need of an at least minimally configured distributed application to do the tests of the developed components as they are programmed. Although minor changes in the code can be made to test parts of component code without the need of distributed configuration, this technique does not reflect the behaviour that the actual system will have and may be error prone. From the programmer point of view, a distributed development framework is desirable in which the applications can be programmed locally, testing as soon as possible the functional requirements of the application, and then configuring externally the deployment of components that compose the application.

Second, the nature of the component is usually embedded in the component code itself. On the contrary, we think that the development of a component should be completely independent of its *nature*; that is, whether it will be a piece of software executed locally, or it will work within a remote component server. This independence with respect to usage is also a programmer's need to develop distributed applications appropriately. Current development frameworks divide the interaction between components in “clients” and “servers” in a classical perspective. It should be interesting to consider “requesters” and “providers” in a cooperative programming model, in a way that requesters can be in turn providers of their own providers. This approach gives a more flexible framework. In addition, this independence allows a given module to act as a normal module when used in an application, and as an active remote module when used in another application.

The third problem that can arise when programming distributed applications in-the-large concerns the specification of the system topology: how the components are distributed, and how every component knows the physical addresses in the network of other components it needs to do the job. Currently a number of methods are available, several of them embedding the structure into the code. But when developing a complex application or prototype, the deployment structure may also be complex, and changes in the topology may be usual during the development phase (changing the network nodes where components will reside, grouping several components in a single node –and possibly merging them in a single component– or even splitting one component in several distributed pieces). A key factor in the flexibility of a distributed development framework is the ability to change the structure of the application with no traumatic reconfiguration effort. A step forward concerning this problem is to centralise the topological configuration in a single point of the application.

Once the development phase has been completed and the topology of the application has been set, running complex applications may arise other kind of problems: how a distributed application can be easily set up. That is, component servers must be started in the proper systems and connected to the correct port numbers, and then the application itself must be started and everything must be checked to work smoothly.

Last, there are several other issues concerning complex application development that have to be taken into account, such as security and certification, isolation of several applications in a single node, and so on. These issues are out of the scope of this proposal, but we have to take them in mind when designing a distributed development framework.

The previous points reflect the problems that may arise when programming in-the-large, but also provide the key to solve them. During the prototyping/programming phase, the most important item to consider is the flexibility and independence from the final structure of the distributed system. The prototype/application should be developed disregarding the physical structure of the final system, and reflecting only the logical interaction between the modules that will compose the system. We talk here about modules instead of components because a component implies an external interface and a usually complex framework; as we have said before, during the development it is common to realize that a component should be built in a different way that it was initially designed. If we consider initially only modules –using the programming language notion of 'module'–, the only thing to do is just to reconfigure the component information. The best solution to provide this functionality is to make the underlying structure of the distributed system completely transparent to the programmer: the specification of active modules and locations must be detailed externally to the source code.

Changing the structure of a distributed system may then be a simpler task, and a very interesting feature of a development framework is the ability to try different system topologies for selecting the one that best fits the requirements of the final system. In this case, complex systems can be reconfigured easily if the physical structure is centralised in a single point: a *distributed configuration file*.

Finally, to start a complex system, the preferable option is to use *name servers* that do the job dynamically. The component servers can be then started in a generic way. This solution also centralises other important tasks such as security control and analysis, load balancing, and possibly other tasks related to the concept of *agency*, such as code security analysis.

6 Related Work

From low-level system calls highly dependent of the operating system to high-level "encapsulated" languages, there are a lot of proposals for developing distributed systems, some of them focusing on the system independence, as OSF DCE and CORBA, while others specifically designed for a language or operating system, as Java RMI or Microsoft DCOM. Other approaches include the use of specific programming languages that model more appropriately the problems involved in distributed programming.

At a higher level, there are frameworks that hide the complexities of distribution providing a programming environment which abstracts the actual network structure. Among them are ActorSpaces [AC93], and Sun JavaSpaces [Mic00], both more or less based on the Linda framework [JR89,CG89]. GNATDIST [YKP96] follows another completely different approach that in certain sense is similar to ours: it defines a configuration language very close to the programming language, in this case Ada. In the LP systems context, Jinni [Tar99] is an interesting scripting tool that provides a software architecture and logic programming extensions, centered in agents and in enabling computation mobility. Also a distributed version of Oz has been developed [SHS97]. Oz is a concurrent multi-paradigm programming language of which the distributed version provides a different semantics to the same language, in order to offer distribution transparency. In addition, it provides fault-tolerance capabilities. Finally, DRL [MDT97a] is a distributed real-time logic language that has inspired a language independent logic-based coordination model [MDT97b]. In this case, the distribution is explicitly made in the source code. The coordination model can be used for other non-logic languages such as Smalltalk or C. This is achieved using source to source compilers that translate the extended syntax to the target language.

Most of the former approaches implicitly constrain in one way or another the programming language or the programming task. Some of the language or system-independent approaches limit the expressive power of the language in constraining to the programming paradigm of the distributed framework (this is the case of CORBA with object orientation). Also, many approaches, even if language-independent, require the use of explicit constructs in the code of the program to deal with distribution (this is the case of Linda, Jinni, and DRL). The development of the program has to be done with the distributed nature of the application in mind.

We argue that this is not interesting, and moreover it is an innecessary burden on the programmer, when the distribution of the execution is independent of the problem solved by the application. In these cases, it is already too complex to program the solution to the problem to have to take care of distribution, in addition. Our proposal aims at simplifying development in these cases.

Our proposal follows the approach of, for example, GNATDIST [YKP96] and Occam [May87], in that there is a clear distinction between the configuration and the application parts of a software system, with different languages, although possibly similar. This similarity of languages is often only syntactic, but in our case we do use the same language (with additional –predefined– library predicates), thanks to the extensibility of Ciao.

In the LP community, comparable proposals are those of Jinni, Oz (Mozart), and DRL. The Jinni approach hides the complexity of network communication between places, and provides a way to move computation from one place to another, enabling a very powerful mobile agent platform. However, remote execution and mobility concepts (represented with operations as `set_host/1` `move/0`, `here/0`, and `there/0`) are visible to the programmer and are embedded in the code, whereas our focus is centered in hiding the distributed structure of the application itself. Distributed Oz has been designed extending the basic operations of the Oz language to provide a distributed semantics to the language constructs already existing in the centralised Oz version, although participant nodes must be declared on source code, due to the lack of a separated configuration language to declare the application network structure. Lastly, DRL also focuses on *execution units*, named *grains*, that can be called on separate processors, and the communication mechanism is based on *logic channels* to simulate a global shared address space. In our approach, the execution units are the program modules, which allows us to avoid the explicit distribution constructs that in DRL programs have to incorporate. In all these cases, the distinguishing aspect of our proposal is not the code distribution transparency but the separation between programming and configuration. Obviously, our approach does not take into account many issues that both Jinni and Oz do, because their and our objectives are different.

7 Conclusions and Future Work

We have proposed a framework and configuration language to express the distributed nature of large logic applications. Our proposal includes a number of features with multiple enhancements from previous implementations. First of all, it proposes the complete separation between application code and distribution configuration code. This allows system reconfiguration with minimum effort. Next, the application code is kept unmodified, and then can be tested and executed either locally or in a distributed way with no program code changes. Also, the distributed configuration language is not yet another language to learn. A configuration file is just a Prolog program with a minimal additional library. From the architectural point of view, a distributed application is a set of processes running on different nodes logically interconnected by name server processes that create and manage the application processes.

This innovative application configuration implies several implementation issues such as program transformation, analysis of automatic module distribution when distributed locations are not explicitly declared, and public active modules for sharing computational resources and providing a base for agent programming.

We are currently developing the proposal, and further enhancements may be necessary in the process. However, we can now identify a number of areas that may be extended in future work. First, it should be interesting to enhance the distributed configuration language with compositional component definition. The distributed architecture provides multiple working areas. First of all is to provide a security system to this distributed architecture. The use of a name server to create every active module provides a very powerful tool to add security capabilities to the proposed architecture. This single point of active module creation enables many other interesting enhancements. For example, a load-balancing system may be implemented, since name server processes communicate in the so called name server network. Name servers can also be extended to provide agency features and other agent capabilities (mobility, language independence, and so on).

References

[AC93] Gul Agha and C. J. Callsen. Actorspace: An open distributed programming paradigm. In *Proceedings 4th ACM Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices*, pages 23–32, 1993.

[CG89] N. Carriero and D. Gelernter. Linda in context, 1989.

[CH95] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid. Available from `http://www.clip.dia.fi.upm.es/`.

[CH96] D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.clip.dia.fi.upm.es/`.

[CH00] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[CH01] D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001.

[JR89] K. K. Jensen and G. E. Riksted. Linda, a distributed programming paradigm. Master's thesis, Department of Mathematics & Computer Science, University of Aalborg, Denmark, June 1989.

[May87] D. May. *Occam 2, Langage definition*. Prentice Hall, 1987.

[MDT97a] B. Rubio M. Díaz and J. M. Troya. Drl: A distributed real-time logic language. *Computer Languages*, 23(2-4):87–120, 1997.

[MDT97b] B. Rubio M. Díaz and J. M. Troya. A logic-based coordination model. In *Workshop on Logic-Based Composition of Software*, July 1997. ICLP Post-Conference Workshop.

[Mic00] Sun Microsystems. Javaspaces service specification, October 2000.

[SHS97] P. Van Roy S. Haridi and . Smolka. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[Tar99] Paul Tarau. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *PAAM'9*. The Practical Applications Company, 1999.

[YKP96] L. Nana Y. Kermarrec and L. Pautet. GNATDIST: A configuration language for distributed ada 95 applications. In *TRI-Ada*, pages 63–72, 1996.