Multiparadigm Model Oriented to Development of Grid Systems

Jorge Luis Victória Barbosa[1], Cristiano André da Costa[1], Adenauer Corrêa Yamin[2], and Cláudio Fernando Resin Geyer[3]

[1] Informatics Department, University of Vale do Rio dos Sinos
São Leopoldo, RS, Brazil
{barbosa,cac}@exatas.unisinos.br
[2] Informatics Department, Catholic University of Pelotas (UCPel)
Pelotas, RS, Brazil
adenauer@ucpel.tche.br
[3] Informatics Institute, Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre, RS, Brazil
geyer@inf.ufrgs.br

Abstract. Multiparadigm approach integrates programming language paradigms. We propose *Holoparadigm* (*Holo*) as a multiparadigm model oriented to development of grid systems. Holo uses a logic blackboard (called *history*) to implement a coordination mechanism. The programs are organized in levels using abstract entities called *beings*. First, we describe the principal concepts of the Holoparadigm. After, the principles of a language based on the Holoparadigm are presented. Besides, we propose the *Grid Holo* (*GHolo*), a platform to support the multi-domain heterogeneous distributed computing of programs developed in Holo. GHolo is based on object mobility and blackboards. This distributed model can be fully implemented on Java platform.

Keywords: Multiparadigm, Mobility, Blackboard and Grid Systems.

1 Introduction

Several programming language paradigms were developed to make computer programming more effective. There is no ideal solution since each one has advantages and disadvantages. Multiparadigm approaches mix two or more basic paradigms trying to get a more powerful and general solution and to overcome the specific limitations of each paradigm taking advantage of its most useful characteristics. Several multiparadigm languages and environments have been proposed as for example [3, 12, 18, 20, 23]. Each paradigm has sources of implicit parallelism, for example, AND parallelism and OR parallelism in logic programming [4, 27]. Another example is object-oriented paradigm that allows the exploitation of inter-object parallelism and intra-object parallelism [9, 21]. The multiparadigm approach integrates paradigms. So, it also integrates their parallelism sources. In this context, interest in automatic exploitation of parallelism in multiparadigm software has emerged. The development of distributed software using multiparadigm models has received attention of the scientific community [9, 13, 14, 21, 25] with some systems considering mobility, heterogeneous hardware and cluster architectures.

In this paper we propose *Holoparadigm* (*Holo*) as a multiparadigm model oriented to development of grid systems. A logic blackboard (called *history*) implements the coordination mechanism and a new programming entity (called *being*) organizes the several encapsulated levels of beings and histories (multi-domains). A new multiparadigm language (*Hololanguage*) implements the main ideas introduced by Holoparadigm. Besides, we propose a platform to support the distributed execution of programs developed in Holo. This platform is called *Grid Holo* (*GHolo*). GHolo has a heterogeneous network as physical execution environment and is based on object mobility, blackboards and multi-domain organization (tree of beings). A prototype was implemented using Java [17] and special libraries to support mobility (Voyager [28]) and blackboards (Jada [10]).

The paper is organized in five sections. Section two presents the Holoparadigm and the Hololanguage. In section three is proposed the Grid Holo. Section four describes related works. Finally, section five draws some conclusions and presents directions for future works.

2 Holoparadigm and Hololanguage

Being is the main Holoparadigm abstraction. There are two kinds of beings: *elementary being* (atomic being without composition levels) and *composed being* (being composed by other beings). An elementary being (figure 1a) is organized in three parts: *interface*, *behavior* and *history*. The interface describes the possible interactions between beings. The behavior contains actions, which implement functionalities. The history is a shared storage space in a being. A composed being (figure 1b) has the same organization, but may be composed by others beings (*component beings*).

Each being has its history. The history is encapsulated in the being. In composed being, the history is shared by component beings. Several levels of encapsulated history can possibly exist. A being uses the history in a specific composition level. For example, figure 1c shows two levels of encapsulated history in a being with three composition levels. Behavior and interface parts are omitted for simplicity.

Automatic distribution is one of the main Holoparadigm goals. Figure 2 exemplifies a possible distribution of the being presented in the figure 1b. Besides that, the figure presents the mobility in Holo. The being is distributed in two nodes of the distributed architecture. The history of a distributed being is called *distributed history*. This kind of history can be implemented using DSM techniques [24] or distributed shared spaces [2, 10, 11].

Mobility [16] is the dislocation capacity of a being. In Holo, there are two kinds of mobility: *logical mobility* (being is moved when crosses one or more borders of beings) and *physical mobility* (dislocation between nodes of distributed architectures). Figure 2 exemplifies two possible mobilities in the being initially presented in the figure 1b. After the dislocation, the moveable being is unable to contact the history of the source being (figure 2, mobility A). However, now the being is able to use the history of the destiny being. Here, physical mobility only occurs if the source and destiny beings are in different nodes of the distributed architecture (it is the case in

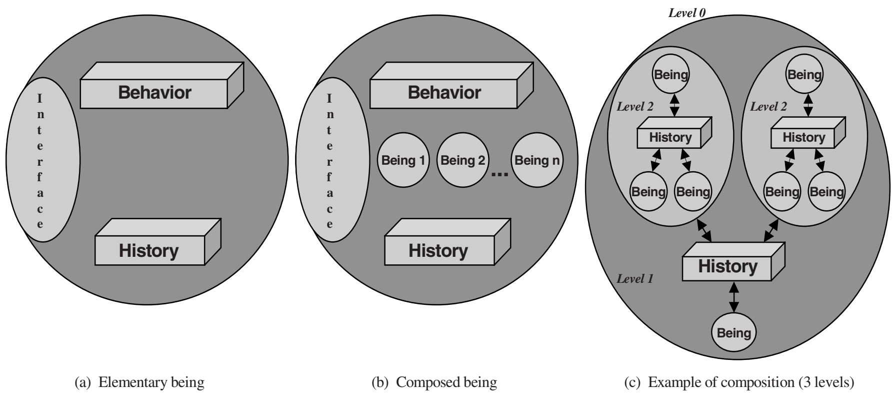

Fig. 1. Being organization

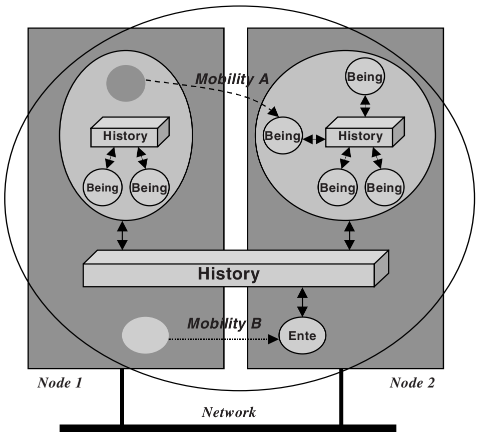

Fig. 2. Distributed being and mobility

our example). Logical and physical mobilities are independent. Occurrence of one does not imply in the occurrence of the other. For example, the mobility B in the figure 2 is a physical mobility without logical mobility. In this example, the moved being does not change its history view (supported by the blackboard). This kind of situation could happen if the execution environment aims to speedup execution through locality exploitation.

The coordination model used in Holo is based on the *blackboard architecture* [22] (figure 3a). A blackboard is composed by a common data area (blackboard) shared by a collection of programming entities called *knowledge sources* (KSs). Control is implicit in the blackboard access operations. The read and write operations in the blackboard are used to communication and synchronization between KSs. This kind of control is called *implicit invocation*. A composed being architecture is similar to the blackboard architecture, since several components share a common data area. In Holo, KSs are beings and the blackboard is the history. Since blackboard implicit

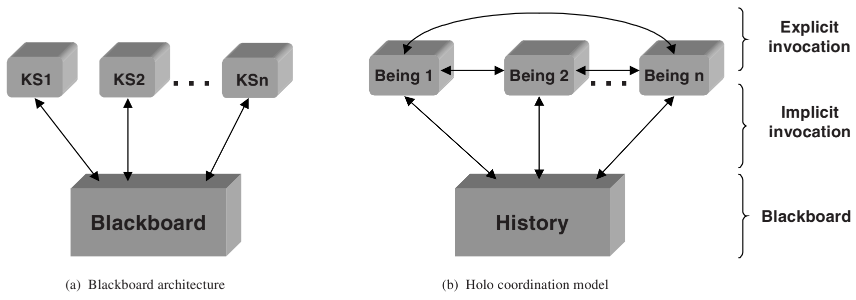

Fig. 3. Holo coordination model

invocation has several limitations, explicit invocation was introduced in the coordination model. Explicit invocation is any direct call between entities. So, the beings influence the others using the history, but can change information directly too.

Figure 3b shows the Holo coordination model. History is a logic blackboard, i.e., the information stored is a group of logic terms. Interaction with the history uses two kinds of Linda-like [8] operations: *affirmation* and *question*. An affirmation puts terms in the history, like asserts in Prolog databases. Moreover, a question permits to consult terms from the history. A consult does a search in the database using unification of terms. A question is *blocking* or *non-blocking*. A blocking question only returns when a unifying term is found. Therefore, blocking questions synchronize beings using the implicit invocation. In a non-blocking question, if a unifying term is not found, the question immediately fails. Besides that, a question is *destructive* or *non-destructive*. A destructive question retracts the unifying term. The non-destructive one does not remove it.

Hololanguage (so-called Holo) is a programming language that implements the concepts of Holoparadigm. A program is composed by descriptions of beings. The language supports logical mobility and concurrency between actions of a being. Besides that, the Hololanguage permits both kinds of blackboard interaction proposed by the Holoparadigm. Five kinds of actions are supported: (1) *logic action* (LA) is a logic predicate; (2) *imperative action* (IA) is a group of imperative commands; (3) *modular logic action* (MLA) contains several logic actions encapsulated in a module; (4) *modular imperative action* (MIA) encapsulates several imperative actions; (5) *multiparadigm action* (MA) integrates logic and imperative actions.

Actions are composed using an *Action Composition Graph* (ACG). Following the Peter Wegner's opinion [29] about the impossibility of mixing logic and imperative behaviors, we have created the *Action Invocation Graph* (AIG). This graph determines the possible order of action calls during a program execution. MAs, IAs and MIAs call any action. LAs and MLAs only call LAs and MLAs. Therefore, there are two regions of actions during an execution, namely, imperative and logic regions.

If an execution flow goes in the logic region, the only way to return to the imperative region is finishing the flow (returning the results asked from the imperative region). This methodology eliminates many problems, which emerge when logic and imperative commands are mixed (for example, distributed backtracking [5]). We believe AIG is an important contribution to the discussion presented by Wegner [29, 30].

3 Grid Holo

Holo to grid systems is called Grid Holo (GHolo). The being multi-level organization (see figure 1c) is adequate to modelling heterogeneous multi-domain distributed systems. Holo was created to support implicit distribution, i.e., automatic exploitation of distribution using mechanisms provided by basic software (compiler and execution environment). Holoparadigm abstractions are hardware independent. However, the model is dedicated to distributed architectures. When the hardware is distributed, there are two main characteristics to be considered: (1) *mobility support*: it is necessary to implement the physical mobility treatment when there is a move to a being located in another node; (2) *dynamic and hierarchical history support*: distribution involves data sharing between beings in different nodes (distributed history). There are several levels of history (hierarchical history). Besides that, access history is adapted during the execution to support the mobility (dynamic history).

GHolo is the software layer that supports the grid distributed computing of programs in Holo. It creates support to physical mobility and dynamic/hierarchical history in a grid. GHolo project is based on a structure called *Tree of Beings* (HoloTree, see figure 4). This structure is used to organize a being during its execution. The tree organizes the beings in levels. A being only can access the history of the composed being to which it belongs. This is equivalent to access the history of being localized in the superior level. A logical mobility is implemented moving a leaf (elementary being) or a tree branch (composed being) from the *source being* to the *destiny being*. The Jada spaces [10] are used to support the change of context. After the mobility, the being moved has direct access to the destiny being's space.

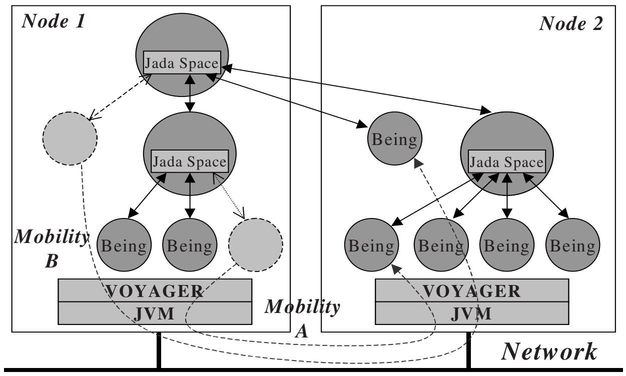

Fig. 4. Grid Holo (distributed HoloTree)

The figure 4 presents the GHolo architecture to the being initially shown in the figure 2. The figure shows the HoloTree distributed in two nodes. The changes to both mobilities of figure 2 are demonstrated. Each being is implemented using an object and a Jada space (*history*). GHolo initialization involves the creation of a *Voyager-enabled program* [28] (Voyager environment) in each node that will be used. Since Voyager executes on the Java Virtual Machine each node will also have a running JVM. During the initialization, a *Table Environment (TE)* indicates the nodes that will be used by GHolo. During a program execution, if a logical mobility results in a physical mobility, it executes a *moveTo* operation in Voyager (mobility A, figures 2 and 4). When a physical mobility is realized without logical mobility (mobility B, figures 2 and 4), the tree of beings does not change, but a *moveTo* operation is realized. This kind of mobility does not have any kind of relation with the program. It is a decision of the environment to support a specific functionality (load balancing, tolerance fault, etc). GHolo does not support this kind of decision yet, but there is on going work in this subject [31].

4 Related Works

There are other multiparadigm implementations over distributed environment. I^+ model [21] supports the distribution of objects, which implement methods using functions (functional classes) and predicates (logic classes). The implementation is based on the translation of functional classes into Lazy ML (LML) modules and translation of logic classes into Prolog modules. The distributed architecture is a network of Unix workstations using 4.3 BSD sockets to implement message passing. The runtime environment was initially implemented using C language, Quintus Prolog and LML. In a second phase, programs were only translated into C language. I^+ does not focus mobility. In addition, none kind of shared space is supported between objects.

Ciampolini et al [9] have proposed DLO, a system to create distributed logic objects. This proposal is based on previous works (Shared Prolog [7], ESP [11] and ETA [2]). The implementation is based on the translation of DLO programs into clauses of a concurrent logic language called Rose [6]. The support to Rose execution is implemented on a MIMD distributed memory parallel architecture (transputer-based Meiko Computing Surface). The runtime environment consists of a parallel abstract machine that is an extension of the WAM [1]. This proposal does not support mobility and it is not applied on a network of workstations. DLO does not support levels of spaces.

Oz multiparadigm language [20] is used to create a distributed platform called Mozart [25]. Oz uses a constraint store similar to a blackboard and supports the use of several paradigm styles [15, 20]. Besides, Mozart has special support to mobility of objects [13] and distributed treatment of logic variables [14]. Mozart distributed architecture is a network of workstations providing standard protocols such as TCP/IP. The runtime environment is composed by four software layers [25]: Oz centralized engine (Oz virtual machine [19]), language graph layer (distributed algorithms to decide when to do a local operation or a network communication), memory management layer (shared communication space and distributed garbage collection) and reliable message layer (transfer of byte sequences between nodes). The physical mobility supported by Mozart is completely transparent, i. e., the system decides when to move an object. None kind of logical mobility is used. The shared spaced supported by Mozart is monotonic and stores constraints, while Holo being's history is a non-monotonic logic blackboard that stores logic tuples (terms). In addition, Mozart does not provide levels of encapsulated contexts composed by objects accessing a shared space.

Tarau has proposed Jinni [26], a logic programming language that supports concurrency, mobility and distributed logic blackboards. Jinni is implemented using BinProlog [5] a multi-threaded Prolog system with ability to generate C/C++ code. Besides that, it has special support to Java, such as a translator allowing packaging of Jinni programs as Java classes. Jinni is not a multiparadigm platform. In addition, Jinni does not work with logical mobility and with levels of encapsulated blackboards.

So, Oz and Jinni have a kind of mobility. In addition, they can be executed over network of workstations. However, we believe that the support to hierarchy of spaces as proposed by Holo is an innovation.

5 Conclusion

We have proposed the use of Holo to multi-domain heterogeneous systems. Holoparadigm concepts estimulate the grid programming. Besides, we proposed the Grid Holo (GHolo), namely, an environment to grid computing that automatically manages the tree of beings distribution.

One important aspect of Holo is the coordination model, which simplifies the management of mobility. For this coordination, the Holo model uses a logic blackboard while GHolo proposes the use of spaces to implement it. Another important concept is the autonomous management of mobility. Holo model does not deal with physical distribution so mobility is always at logic level, i.e., between beings. GHolo execution environment can define what kind of mobility is necessary: a logical or a physical one. A logical mobility requires changes in history sharing, while physical also involves objects mobility.

Future works will improve our proposal. One ongoing work [31] aims to propose a dynamic scheduling of distributed objects, which can be directly used in GHolo. Optimizations over initial execution kernel are also under development.

References

1. Ait-Kaci, H. Warren's Abstract Machine – A Tutorial Reconstruction. MIT Press, 1991.
2. Ambriola, V.; Cignoni, G. A.; Semini; L. A Proposal to Merge Multiple Tuple Spaces, Object Orientation and Logic Programming. Computer Languages, Elmsford, v.22, n.2/3, p.79-93, July/October 1996.
3. Apt, R. et al. Alma-0: An Imperative Language that Supports Declarative Programming. ACM Transactions on Programming Languages and Systems, New York, v.20, September 1998.
4. Barbosa, J. L. V.; Vargas, P. K.; Geyer, C. GRANLOG: An Integrated Granularity Analysis Model for Parallel Logic Programming. Workshop on Parallelism and Implementation Technology (constraint) Logic Programming, London, 2000.
5. Bosschere, K.; Tarau, P. Blackboard-based Extensions in Prolog. Software – Practice and Experience, v.26, n.1, p.49-69, January 1996.
6. Brogi, A. AND-parallelism without shared variables. Seventh International Conference on Logic Programming. MIT Press, p.306-324, 1990.

7. Brogi, A.; Ciancarini, P. The Concurrent Language, Shared Prolog. ACM Transaction on Programming Languages and Systems. New York, v.13, n.1, p.99-123, January 1991.
8. Carriero, N.; Gelernter, D. Linda in context. Communications of the ACM, v.32, n.4, p.444-458, 1989.
9. Ciampolini, A.; Lamma, E.; Stefanelli, C; Mello, P. Distributed Logic Objects. Computer Languages, v.22, n.4, p.237-258, December 1996.
10. Ciancarini, P.; Rossi, D. JADA: A Coordination Toolkit for Java. http://www.cs.unibo.it/~rossi/jada, 2003.
11. Ciancarini, P. Distributed Programming with Logic Tuple Spaces. New Generating Computing, Berlin, v.12, n.3, p.251-283, 1994.
12. Hanus, M. The Integration of Functions into Logic Programming from Theory to Practice. Journal of Logic Programming, New York, v.19/20, p.583-628, May/July 1994.
13. Haridi, S. et al. Programming Languages for Distributed Applications. New Generating Computing, v.16, n.3, p.223-261, 1998.
14. Haridi, S. et al. Efficient Logic Variables for Distributed Computing. ACM Transactions on Programming Languages and Systems, v. 21, n.3, p.569-626, May 1999.
15. Henz, M. Objects in Oz. Saarbrüchen: Universität des Saarlandes, May 1997. (PhD Thesis)
16. IEEE Transactions on Software Engineering, v.24, n.5, May 1998. (Special Issue on Mobility)
17. Java. http://www.sun.com/java, 2003
18. Lee, J. H. M.; Pun, P. K. C. Object Logic Integration: A Multiparadigm Design Methodology and a Programming Language. Computer Languages, v.23, n.1, p.25-42, April 1997.
19. Meh, M.; Scheidhauer, R.; Schulte, C. An Abstract Machine for OZ. Seventh International Symposium on Programming Languages, Implementations, Logics and Programs (PLIP'95), Springer-Verlag, LNCS, September 1995.
20. Muller, M.; Muller, T.; Roy, P. V. Multiparadigm Programming in Oz. Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor of Prolog, 1995.
21. Ng, K. W.; Luk, C. K. I+: A Multiparadigm Language for Object-Oriented Declarative Programming. Computer Languages, v.21, n.2, p. 81-100, July 1995.
22. Nii, H. P. Blackboard systems: the blackboard model of problem solving and the evolution of blackboard architectures. AI Magazine, v.7, n.2, p.38-53, 1986.
23. Pineda, A.; Hermenegildo, M. O'CIAO: An Object Oriented Programming Model Using CIAO Prolog. Technical report CLIP 5/99.0 , Facultad de Informática, UMP, July 1999.
24. Proceedings of the IEEE, v.87, n.3, march 1999. (Special Issue on Distributed DSM)
25. Roy, P. V. et al. Mobile Objects in Distributed Oz. ACM Transactions on Programming Languages and Systems, v.19, n.5, p.804-851, September 1997.
26. Tarau, P. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. PAAM'9, The Practical Applications Company, 1999.
27. Vargas, P. K.; Barbosa, J. L. V.; Ferrari, D.; Geyer, C. F. R.; Chassin, J. Distributed OR Scheduling with Granularity Information. XII Symposium on Computer Architecture and High Performance Computing, Brazil, 2000.
28. Voyager. http://www.recursionsw.com/products/voyager/voyager.asp, 2003
29. Wegner, P. Tradeoffs between Reasoning and Modeling. In: Agha, G.; Wegner, P.; Yonezawa, A. (eds.). Research Direction in Concurrent Object-Oriented Programming. Mit Press, p.22-41, 1993.
30. Wegner, P. Why interaction is more powerful than algorithms. Communications of the ACM, v. 40, n. 5, p.80-91, May 1997.
31. Yamin, A. C. ExEHDA: Execution Environment for High Distriubted Applications. PPGC/UFRGS, 2001. (PHD proposal)