Merging Logic Programming into Web-based technology:

a Coordination-based approach

Enrico Denti, Antonio Natali**, Andrea Omicini**

Dipartimento di Elettronica, Informatica e Sistemistica
Università di Bologna - Viale Risorgimento, 2 - I-40136 Bologna, Italy
Tel. + 39 51 6443087 / 644 3021** - Fax + 39 51 6443073*
E-mail: {edenti,anatali,aomicini}@deis.unibo.it
WWW: http://www-lia.deis.unibo.it/Staff/

Abstract. Current WWW technology is becoming the de-facto standard platform for groupware applications, yet it provides virtually no effective coordination capabilities. New applications, instead, demand higher-level middleware services, with intelligent behaviours, deductive capabilities, and effective coordination. In this work we discuss a possible extension to the current Web-based architectures whose aim is to support the integration between logic-based and conventional components, and to introduce the concept of programmable communication abstraction. Within the framework of the current Java-based architecture, logic programming is used as a base for Linda-like coordination. Thus, the coordination medium may be interpreted as the theory of communication, and exploited for reasoning over communication and for introspection over the overall system state. Logic programming is also used to define *reactions* to communication events, which allow the communication abstraction behaviour to be extended and tailored to achieve specific coordination policies.

Keywords: World-Wide-Web, Internet, Java, Multi-Agent Systems, Agents, Interaction, Coordination, Tuple Spaces, Programmable Coordination Media, Extensible Coordination Abstractions

1. Introduction

Both in the industry and in the academic realities there is a growing demand for groupware and co-operative distributed-work applications, such as videoconferencing, remote work, remote people training and education, whose needs are far beyond simple document exchange. By "providing the resulting illusion of several cooperating applications working together" [1], the World-Wide Web and the Internet "are rapidly becoming the de facto standard and hardware platform" for such geographically distributed groupware applications [1,2].

The success of the WWW as a support *environment* is mainly due to the simplicity of its three basic support mechanisms - the HTTP protocol for data access, the HTML mark-up language for page description, and the URL global naming scheme - and to its open architecture, which provides a way to integrate further technologies - such as CGI scripts [4,5], the Java language [8,9] with its Abstract Window Toolkit, browser scripting languages like Javascript [6], MIME-types and browser helper applications - in the basic framework.

However, the forthcoming application areas are much more demanding then the current WWW implementation can provide. In fact, new added-value interactive services such as remote banking, insurance, health care, revenue tellers, and public information access systems (for cultural events, train and flight schedules, on-line reservations, hotel booking, road condition inquiry, etc.) require higher-level services (sometimes referred to as *middleware*) offering new abstractions so as to compose new and old programs in more complex software ensembles - including, in particular, facilities to easily access geographically distributed information spaces, personal communication tools, and to interoperate with heterogeneous Data Base Management Systems (DBMS).

For all these reasons, solutions are emerging trying to extend the current WWW middleware capabilities so as to support the integration between WWW and other technologies. Component technology is dramatically altering the way software systems are built: in particular, Java is becoming an architectural model, rather than just a language, for Internet-based applications, providing the WWW with the architectural reference which is currently lacking. In addition, its role is becoming central in supporting the rapid development of new, hybrid applications, where very heterogeneous (Java-based, Visual Basic-based [13]) components may be asked to cooperate within the same ensemble. With respect to this issue, the JavaBeans technology [10], currently under definition, is likely becoming the *de facto* standard for dynamic, distributed component-based applications, as shown by the development of bridge software allowing Visual Basic ActiveX controls to interoperate with JavaBeans components [11] and by the embodiment of the CORBA/OpenDoc architecture [5].

Despite its intrinsically distributed nature, however, the WWW as we know it now lacks both the coherent conceptual framework and the *coordination* capabilities ([3]) which are essential to create an effective platform for groupware and co-operative information systems. In its current definition, also Java lacks a general coordination model intended as (quoting [3]) "the glue that binds separate activities into an ensemble", although efforts are being made to define global communication abstractions and models - the *Javaspaces* [26-30] - based on Linda-like [7] *tuple spaces*, so as to help filling this gap. In fact, the Remote Method Invocation (RMI) package [14], available to support remote object method invocation on objects executing on the name space of a remote host, still relies on a basic point-to-point communication protocol, thus providing no real higher-level coordination abstractions [3,35]. The same applies to CORBA [5], another widely-adopted distributed object model platform. Actually, it seems that most approaches currently used to extend middleware capabilities are aimed more to reuse existing metaphors, technologies and tools to quickly port applications to the Internet platform, rather than to introduce really innovative abstractions specifically designed for the features of this platform.

On the other hand, many forthcoming application areas will require built-in mechanisms to perform inferential activities. Reasoning capabilities would be clearly welcome, for instance, in remote banking applications (to provide extra information to the customers, and to monitor the system state and/or consistency), in the health care field (where expert-systems features are often used to support diagnoses), in train/flight schedule services (to find connections respecting customer-defined constraints, to infer customer habit information to be used to improve the service, etc.), in on-line reservation systems (to suggest alternatives, to find best price/service combinations, etc.), in road-condition monitoring systems (here again, to suggest alternatives routes, to check for expected critical conditions), and so on.

Due to its well-known features with respect to knowledge representation and handling, Logic Programming is quite a natural choice to provide the extra intelligence demanded by such situations: the question is under what form it may be integrated and properly exploited in the WWW framework. In fact, it is very unlikely that the current WWW technology and architecture may radically change their reference models, languages and tools as a consequence of the integration with Logic Programming. Rather, Logic Programming will provide, at least in the short time, a contribution only if properly integrated within the current WWW architecture. Moreover, deductive capabilities alone would not suffice to make such an integration really useful and effective: coordination capabilities are - for the reasons outlined above - at least as important as deductive features.

Coherently, two main approaches can be followed. The first one consists of providing the logic-based support as a "backstage" service, adding intelligence to the system without changing the existing architecture, by inserting an ad-hoc component. The other one, instead, is based on the idea of exploiting coordination models to introduce an architecture providing a bridge between components with inferential capabilities (henceforth, *logic-based* components) and conventional components.

Since the second alternative is more general than the first, our choice is to start from the component model, and to introduce a new software layer, acting both as a high-level communication device (from the viewpoint of conventional components) an as a knowledge base (from the viewpoint of logic components).

In fact, agent coordination strategies are often the result of the overall coordination / competition among individual agents, and are often difficult to understand and implement based on point-to-point communication. Thus, our main goal is to extend the current WWW architectures with a new (abstract) component which offers a higher abstraction level for object interaction, in order to overcome the low-level, event-based programming model of the Web while providing a more appropriate support for inferential activities.

However, the introduction of a new component providing a common reference language as well as interaction capabilities between logic-based and conventional components is not enough, as one key issue is how to impose different coordination strategies on top of the message-passing paradigm. For this reason, the software component supporting the new layer should also be dynamically "programmable" so as the logic of interaction, which is usually spread among the interacting agents, can be embedded into the coordination medium. As already shown in [19,20], this choice can greatly improve the design and maintenance of Web-based systems.

In short, the desired software layer should be designed according to a reference coordination model which allows logic-agent to reason over the available, private or common, knowledge and non-logic agents to access such a knowledge as conventional messages. From the application viewpoint it should provide a program interface making it possible for logic-based components to perform deductions in a time-dependent environment while offering at the same time an extendible behaviour in order to accommodate any required coordination policy.

A suitable starting point to achieve these goals is represented by the ACLT coordination model [16,17,18,19,20] which already meets several of these requirements. Moving from the observation that an interactive systems is (or should be) something more than the simple sum of its parts [24], ACLT adopts Linda[7]-like tuple spaces as its basic communication devices, but enhances them under several aspects.

First, the global (distributed) communication medium is structured in such a way that it can directly constitute the knowledge-base for logic-components. Named, logic tuple spaces are used as communication channels, allowing deductive activities to be performed through a set of *demo* primitives, defined according a coherent notion of logic consequence in a time-dependent environment [16,17]. However, non-logic components (i.e., components written in C, C++, Java, or other imperative languages) can still access the communication channels in a traditional way, viewing them as conventional tuple spaces with no extra features. So, each component (*agent* in ACLT terminology) can access the communication channels at its own perception level.

Second, ACLT raises observability from tuples to *operations on tuples*, thus shifting the observability level from simple state observability (also provided in Javaspaces by the *notification* concept [26]) to communication operations observability, which provides the *dynamic* view of the system required in distributed, complex systems (for instance, for monitoring and debugging purposes) [31,32] and also constitutes a powerful mechanism to build the heterogeneous, coordinated, introspective component-based systems which are in the aims of the Web-based technology.

This is the concept we exploited to make the communication space programmable. By allowing user-defined *reactions* to be triggered upon the occurrence of some given communication event(s) [18,19,20], ACLT makes it possible to extend of the execution of a communication operation (*in*, *out*, *read*, ...) in a non-intrusive way (i.e., leaving the semantics of the basic communication primitive untouched). Thanks to its reaction execution model, ACLT can make agents perceive all the effects of a single communication operation as a single-step transition of the communication abstraction state, so that the behaviour of the coordination medium can be made (to appear) as complex as desired.

Originally defined in 1995 [16] (following the previous work on the CARA model [36,37]), ACLT is not just a reference model, but also a working system, currently implemented on top of Sicstus Prolog [15]. So, the new software layer proposed in this work can be introduced, at least initially, as a proper set of interface components, which exploit the already-available system while providing the abstraction layers and the necessary technological bridges.

In the following of this work, we will discuss how such a hybrid technology can be exploited to provide Web-based application with both deductive and real coordination capabilities. To this end, next Section briefly summarises the current WWW technology from the coordination and the component-based technology viewpoints, Section 3 summarises the basics of the ACLT model, and Section 4 discusses our approach more deeply, also describing some possible application scenarios. Conclusions and final remarks are reported in Section 5.

2. Current WWW technology

Currently, a Web server can be seen as a multi-coordinated application, which exploits several, different communication protocols to coordinate itself with the other entities. Basic Internet browser requests are handled using (only) the HTTP protocol, but this provides a fixed behaviour which allows only static pages to be consulted, and is therefore insufficient for many purposes. In fact, people often require specific information, which has to be selected dynamically based on some input data inserted by means of HTML forms. In this case, form data are issued to the Web server via other protocols (POST or GET) [4], that the Web server must be able to handle, too.

Moreover, it often happens that the server has to interact with an external database, which means that further protocols have to be used. On many platforms, this interaction is achieved using operating-system facilities and communication models, such as ActiveX controls, OLE [25], or AppleEvent. In order to process input data and dynamically build the response page, the Web server must be equipped with an ad-hoc program: typically, this program is a script written in a language such as Perl [12], Visual Basic Script [13], Tcl/Tk [33] or platform-specific shell languages, that interacts with the Web server through the CGI (Common Gateway Interface) interface protocol [4,5]. One more protocol is then needed to communicate with the Webmaster - a case where, once again, operating-systems models are typically adopted, often properly extended (for instance, using the Microsoft Frontpage Extensions) so as to allow higher-level interaction based on more sophisticated site administration tools. Moreover, the Web server cannot maintain state information from one request to another (unless by using tricks such as writing it to some HTML page), as each request is translated into a single HTTP transaction: so, no information is available over the global coordination state.

Obviously, this basic scheme introduces a clear bottleneck, as it puts all the computational load onto the Web server. In particular, possible errors in HTML input forms would be discovered only after the form has been completely filled in by the user and analysed at the server side, consuming both server and user time and generating extra network traffic. As a further consequence, no active behaviours (animations, calculations,...) could be coded in HTML pages, since all activities would be always performed by the Web server.

So, the trend in current WWW technology is to move as much of the computational load as possible to the client side, helping to reduce the Web server load and response times, and to avoid any unnecessary task centralisation.

In the Java [8,9] approach, Internet-based applications (*applets* in Java terminology) are provided as separate Java programs, whose code is referenced inside HTML pages using a specific `<APPLET>` tag. So, HTML pages continue to describe the page appearance only, while the behaviour description is defined by the external applets executed by the Java Virtual Machine (JVM), which is available as an extension (Netscape Navigator’s plug-in, Internet Explorer’s ActiveX control, etc.) to most of the widely-used Internet browsers. The JVM is often so deeply integrated in the browser environment that the user may hardly distinguish it from the browser itself. However, it remains - both conceptually and physically - a separate component, which interacts with the browser according to the protocol defined for applet handling. Consequently, no special extension is required to the browser, except for the capability of handling the new `<APPLET>` tag by downloading the applet classes and passing them to the JVM for execution.

In the script language approach, instead, *behaviours* to be executed in response to user events are interleaved with conventional HTML code, so that pages describe other than just the page appearance and content. This is done by embedding simple event handlers directly in most HTML tags and by defining separate logical sections of the HTML page (delimited by the `<SCRIPT>` tag), inside which behaviours are encoded by means of script programs written - in principle - using any adequate script language (Javascript, Visual Basic Script,...). The browser must incorporate an interpreter for the selected script language(s): this means that no further issues are introduced from the interaction viewpoint, as all the extra work is performed by the browser component itself.

So, the use of "script" programs and languages in this approach is inherently different from using them at the server side, where scripts are executed by separate interpreters interfaced to the Web server via the CGI protocol, which implies a completely different interaction scheme.

3. The basics of the ACLT model

The ACLT coordination model [16,17,18,19,20] is founded on the Linda [7] model, which introduces the notion of *generative communication* and promotes the separation between the *computation model* and the *coordination model* [3], based on a shared memory communication abstraction called *tuple space*. Here we will assume the Linda model as known, together with its most common extensions (like the non-blocking, predicate-version `in_noblock` - or `inp` - and `rd_noblock` - or `rdp` - primitives).

ACLT extends the basic Linda with the notion of multiple, *logic* tuple spaces [21,22,23]. A logic tuple space is a collection of first-order unitary clauses, which can be interpreted either as a simple communication device, or as a knowledge repository: the latter reading suggests that it can be used as a logic theory, where deductive activities can be performed over the current communication state. ACLT also introduces the notion of *reactive* tuple space [18,19,20], i.e. of a programmable communication channel, which can be dynamically extended to support arbitrarily complex behaviours while keeping the semantics of the basic communication primitives untouched.

3.1. Deductions over evolving logic theories

The first consequence of the twofold nature of ACLT communication devices - as conventional tuple spaces and as logic theories - is to induce a categorisation of agents. In particular, *logic* agents (for instance, Prolog agents) are aware of the nature of ACLT tuple spaces as logic theories and can perform logic proofs on them, while *non-logic* agents (for instance, C or Java agents) consider ACLT tuple spaces just as conventional communication channels, and can access them only in a traditional way, using pattern-matching instead of unification, as imperative languages provide no notion of variable binding.

When dealing with logic agents, the fundamental issue is how to provide a notion of logic consequence allowing the logic view of clauses as immutable truths to coexist with the intrinsically transient nature of knowledge in tuple spaces [16,17]. More precisely, the question is how to ensure *correctness* and *completeness* of the proof process.

Basically, ACLT addresses the correctness issue by introduces a knowledge classification scheme, distinguishing between *persistent* knowledge (which refers to the time-independent elements of the application domain) and *transient* knowledge (which refers to that part of the world which may evolve during agent life), and allowing logic proofs to be performed over persistent knowledge only. As a result, transient knowledge can be removed but, for this very same reason, not taken as a base for deductions, while persistent knowledge can be used for logic inferences, but, consequently, cannot be retracted. Since ACLT exploits predicates as modularity sources, each predicate in a tuple space is labelled - either implicitly or explicitly - to represent either transient or persistent knowledge.

The completeness issue is less critical, and is faced by providing a special set of logic proof primitives which are able to keep the new knowledge into account upon backtracking [16,17,18]. ACLT also addresses the *synchronisation* issue, which captures the idea that sometimes it may be "too early to deduce", so logic proof operations might need to be properly synchronised before actually starting deduction [16,17,18]. Here again, a special set of *hybrid* logic proof primitives is provided, which wait (synchronise) if no suitable knowledge is available when the logic proof starts.

3.2. The reaction model

ACLT reaction model is aimed to provide a simple yet powerful way to make the communication abstraction programmable and dynamically extensible, while keeping the semantics of the basic communication primitives untouched. The basic idea is to define a communication channel where communication events are made observable at the system level, thus raising the observability level from state changes observability (as in the case of conventional tuple spaces), to communication event observability - that is, from tuples to communication operations over tuples. Tuple spaces are then given the capability of triggering activities (*reactions*) when some specified communication events occur

[18,19,20].

To this end, any physical communication event can be associated with one or more logical events (each denoted by a unique name) by means of the special tuple `map(Operation, Event)`. Multiple logical events can be associated to the same physical event, and multiple physical events can correspond to the same logical event. Reactions are specified through tuples like `react(Event, Goal)`, where *Goal* is a conjunction of term predicates, state primitives (`current_op/1`, `current_tuple/1`, ...), and basic (non-blocking) communication primitives.

If multiple react/2 refer to the same logical *Event*, the corresponding *Goal*s are all executed as mutually-independent atomic actions, in a non-deterministic order [18,19,20], with a transaction semantics: so, a reaction succeeds only if all its subgoals are executed successfully, and aborts otherwise. Of course, only successful reactions produce effects. When a reaction succeeds, all the side-effects associated to it are carried out simultaneously, as a single transition of the tuple space state. Due to this transaction semantics, reaction nesting is not allowed, and reactions triggered by another (successful) reaction are executed only *after* the first one has been successfully completed.

However, the model ensures that all such reactions are executed *before* serving any other agent-triggered communication event, so that agents can perceive *only the final result* of the execution of both the communication event and all its related reactions. In this way, agents perceive the response of a tuple space to a communication event as a *single* computational step (i.e., a single transition of the tuple space state), yet it is no longer forced to be simple and fixed by the model like in standard Linda, but can be as complex as required by the coordination policy.

Finally, since the reaction specification language is founded on the same communication pattern exploited for agent interaction (logic tuples and basic operations over tuple spaces), agents can in principle dynamically manipulate the communication abstraction behaviour, possibly as a result of deductions on the current coordination state and according to the overall system goals.

4. Merging Logic Programming into Internet-based applications

4.1. Programmable logic tuple spaces as software components

Basically, our proposal is based on the idea of using a programmable, logic tuple space as a software component within a Java-based Web architecture, exploiting ACLT both as a reference model and as the first kernel for the software layer itself.

By adding a tuple space component, all the coordination features of the Linda model become available to Web-based applications. So, a higher abstraction level for object interaction, with generative communication, can be exploited, as well as Linda's well known properties such as time uncoupling and space uncoupling between the communicating agents. In this way, the low-level, point-to-point interaction model of the Web can be overcome, helping to provide for the coordination weaknesses of the Web platform.

Moreover, the choice of adopting a *logic* tuple space (instead of a conventional one) makes it possible to interpret the communication channel both as a high-level communication device and as a knowledge base, thus allowing heterogeneous (logic and non-logic) components to coexist and fruitfully interoperate. This chance opens new perspectives to Web-based applications, as intelligent components can be added with no impact on the overall system architecture.

Since there can be no reasonings without a proper representation of the relevant information (i.e., communication events) as logic tuples in the tuple space, at least some application components will have to be made aware of the existence of the logic tuple space component, and exploit it by inserting there any relevant knowledge. This is the (obvious) unavoidable drawback of the architectural restructuring implied by the insertion of a coordination medium in the existing architecture.

For instance, in order to insert the logic tuple space component on a Web server, so that logic inferences can be done on its activity (and subsequent actions be taken), each communication event of the HTTP protocol should be made observable by generating a tuple in the tuple space. This requires that the Web server is slightly extended, so as to output such a tuple in correspondence to each relevant HTTP protocol event: however, the (HTTP) interaction protocol remains untouched, and, therefore, any application based on such a server will continue to work normally. The next step will be to define the number and the role of logic agents, which, depending on the desired purposes - network traffic monitoring, page download statistics, automatic mirroring, etc. - will exploit the available coordination state to perform the required reasonings (some of these case studies are presented below).

Although a logic tuple space already provides many interesting features, the availability of a programmable (and dynamically extensible) tuple space makes it possible to implement coordination policies which exploit this global coordination medium. This feature is particularly relevant in a component-based architecture such as the Web platform, as it implies that the global system behaviour may be changed without intervening on the single components.

In [19,20], for instance, several agents behave as Dining Philosophers [34], coordinating themselves by means of an ACLT programmable, logic tuple space. Each philosopher agent interacts with the communication abstraction according to a very simple protocol (acquire/release forks), while all the related low-level handling is put onto the programmable tuple space. In this way, subsequent changes to the basic problem structure (e.g., introducing a concept of meal time - breakfast, lunch, dinner - and corresponding specific forks) can be dealt with by simply re-programming the extensible logic tuple space, with no change at all to the basic agent interaction protocol. In a Web-based platform, philosophers could easily be Java components, possibly distributed over the Internet, coordinating themselves through the programmable software component discussed above.

More generally, components spread over the Internet may become aware of the existence of our logic-based programmable component, and decide to exploit its services simply by downloading the "bridge" software component, which may be an applet or a Java Bean, from a conventional Web or FTP server. In this way each browser, software component, operating system or application could become *ipso facto* able to interoperate with any other component across the net, using the programmable, logic-based component as a coordination support.

4.2. Some Web-based case studies

As a first case study, consider the case of the management of a large Web server (such as the one used in many large companies to deliver information and software to their customers), physically made of several computers, all mapped to the same URL. Since traffic has peaks at some hours, but is much lower at other times, a different number of servers could be running at different times of the day. In addition, extra communication channels may be reserved for this company (by the telecommunication company) in some range of hours, and paid correspondingly. Moreover, some servers may be disconnected from time to time for maintenance or failures. In all these cases, a dynamic, adaptive behaviour of the overall system would be welcome. In particular, the company may not like to hire extra communication channels unless in presence of a very high traffic volume, nor may it like to keep many servers uselessly working at off times. On the other hand, should a server be down for maintenance or failure, it would be desirable that all its requests could be diverted to another server or to a backup computer.

Another interesting case study could be a dynamic mirror service, once again aimed to reduce the overall network traffic. A highly requested page (or file) could be detected and backed up to another (possibly geographically closer) server, rerouting subsequent requests to the new server. Or, conversely, a local copy of a remote highly requested page (or file) could be made if certain conditions hold, thus achieving a proxy-like, but more intelligent, behaviour.

The first service could be delegated to a "resource manager" logic agent, while the second one could be assigned to a "mirror manager" logic agent. Since the resource manager is asked to keep the traffic under control, it may be triggered by each new connection request.

For this purpose, the coordination component should be programmed so as to translate the occurrence of such an HTTP event into an observable tuple of the form expected by the "resource manager" logic component. This should reason over the current connections (and possibly on their origin), applying its rules to decide whether to activate (and pay for) new lines, and/or new servers - or, conversely, whether it is the case to disable some of them if traffic is getting lower. During this monitoring activity, it may notice that some server is out of service, in which case an interesting behaviour would be to patch the situation by re-programming "on the fly" the coordination component, so that all requests to that server are dynamically redirected to another backup computer.

This further degree of flexibility, however, requires a re-design of the Web server architecture, so that its actions can be influenced by the knowledge currently available in the tuple space, instead of being a-priori defined. This means that the existing interaction protocol - and the corresponding slight extension of the Web server code discussed above, consisting of "reflecting onto the tuple space" all relevant HTTP events - is no longer sufficient: instead, all the communication must occur through the tuple space, so that its coordination power can be fully exploited. To this end, the two main components of the Web server - the one receiving client requests, and the one replying to those requests -, currently embedded into a single component and interacting with their own protocol, should be uncoupled and coordinated through the coordination medium, too, thus achieving an effective flexibility in changing interaction policies, while leaving things unmodified at the external perception level.

Obviously, the "mirror manager" could be structured in a similar way, too: in this case, however, the relevant HTTP event would be the single page (or file) request, rather than the opening of a new connection. Correspondingly, the tuple space should be programmed differently, so as to translate each page request into an observable tuple of a proper form, which would activate another proper intelligent component. This agent should check the situation, and once again, based on its rules, decide whether to back up the page to another server, or (in a proxy-like behaviour) evaluate if the number of requests is worth a local mirror, etc.

5. Related works and conclusions

The World Wide Web and Internet are not only offering a standard support for the development of middleware services; they are also demanding an effective integration of several concepts, methodologies and mechanisms, by focusing the attention upon architectural issues rather than on the algorithmic aspects of computation. Unsurprisingly, the Object-Oriented Programming model is adopted as the reference model and no attention seems to be given to Logic Programming. However, the intrinsic complexity of middleware software and the ever-growing functionality requested to the underlying systems demand higher abstraction levels. In fact, as it usually happens, the current WWW architectural model is the result of an abstraction process performed upon low-level mechanisms. From the programmer's viewpoint, this means dealing with concepts such as messages, events, event handlers, and so on, while the need for knowledge-based programming concepts, reflective capabilities and declarative computational models are gradually emerging.

So, the problem is how to exploit Logic Programming without having to radically change systems which are not only already available, but also rapidly growing according to the incremental methodology provided by the Object-Oriented Programming paradigm.

In this work, we propose the use of the software component model itself as a way to introduce logic-based agents in the context of Web-based systems in the short time. The key-point consists of providing a new kind of communication abstraction. First, the twofold interpretation of a logic tuple space makes it possible to support the interaction of heterogeneous agents, by providing a bridge between logic (Prolog) and conventional (Java, C) agents. Other systems, such as Linda Interactor and LogiMOO [35,38,39], successfully deal with the same issues, providing access to a common logic-based tuple space for both BinProlog [39] and Java agents, even as Netscape applets. However, ACLT further exploits the logic nature of the tuple space, through its *demo* primitives [16] which safely integrate synchronisation issues and logic proofs.

Moreover, the programmability feature of ACLT tuple spaces [20] makes it easier to design complex component-based architectures [19], by naturally shifting design focus from agents to the coordination media. Being specified in terms of first-order logic tuples, ACLT reactions [18] reuse the same communication language, based on logic tuples, as ACLT agents: so, intelligent agents are given the chance to dynamically alter the coordination policies, according to reasonings performed upon the overall system state (as represented in the coordination medium).

Our intent is then to provide a tool allowing Prolog agents to run together and to interact with Java agents, in the conviction that this can get the benefits of logic programming to be concretely perceived and spread into a large and interested community. Of course, there is a price to be paid, which is to adapt the Logic Programming model to the notion of a modifiable theory: but perhaps this is one of the most critical points to be faced in order to merge Logic Programming within the current Internet technology.

6. References

[1] P.Ciancarini, R.Tolksdorf, F.Vitali. *Weaving the Web Using Coordination*. In *Coordination Languages and Models (First International Conference Coordination '96 - Cesena, Italy, April 1996)*, P. Ciancarini and C. Hankin (Eds), Lecture Notes in Computer Science No. 1061, Springer Verlag, Berlin-Heidelberg, Germany, 1996, pp. 411-415. ISBN 3-540-61052-9.

[2] P.Ciancarini. *Coordination Models as Software Integrators*. ACM Computing Surveys, 28(2), June 1996.

[3] D.Gelertner, N.Carriero. *Coordination Languages and Their Significance*. Communications of the ACM, 35(2), February 1992.

[4] *The Common Gateway Interface.* Internet reference. http://www.pricecostco.com/exchange/irf/cgi-spec.html

[5] Object Management Group. *The Common Object Request Broker Architecture: specification.* Technical Report, OMG, July 1995. Revision 2.0.

[6] Sun Microsystems. *The JavaScript Language*. Internet reference http://java.sun.com/pr/1995/pr951204-03.html

[7] D.Gelertner. *Generative Communication in Linda*. ACM Transactions on Programming Languages and Systems, 7(1), January 1985.

[8] M.Campione, K.Walrath. *The Java Tutorial - Object-oriented programming for the Internet.* The Java Series, Addison-Wesley, 1996. ISBN 0-201-63454-6

[9] K. Arnold, J. Gosling. *The Java Programming Language*. The Java Series, Addison-Wesley, 1996. ISBN 0-201-63455-4.

[10] Sun Microsystems. *The JavaBeans technology.* Internet Reference http://splash.javasoft.com/beans/

[11] Sun Microsystems. *The JavaBeans bridge for ActiveX.* Internet Reference http://splash.javasoft.com/beans/bridge/

[12] *The Perl Language v. 5.001 documentation*. Internet reference. http://www.cheque.uq.edu.au/misc/perl-5.001m/perl.html

[13] Jinjer L. Simon.*VBScript SuperBible.* Waite Publishers.

[14] Sun Microsystems. *Remote Method Invocation specification*. Internet reference http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-intro.doc.html

[15] Swedish Institute of Computer Science. *Sicstus Prolog User's Manual.* Kista, Sweden. Internet Reference http://www.sics.se/isl/sicstus.html

[16] A.Omicini, E.Denti, A.Natali. *Agent Communication and Control through Logic Theories.* In *Topics in Artificial Intelligence (Proceedings of the Fourth Congress of the Italian Association for Artificial Intelligence AI*IA '95 - Florence, Italy, October 1995)*, M. Gori e G. Soda (Eds), Lecture Notes in Artificial Intelligence No. 992, Springer Verlag, Heidelberg, Germany, 1995, pp. 439-450. ISBN 3-540-60437-5

[17] E.Denti, A.Natali, A.Omicini,M.Venuti. *Logic Tuple Spaces for the Coordination of Heterogeneous Agents.* In *Frontiers of Combining Systems (Proceedings of the First International Workshop on Combining Systems FroCoS '96 - München, Germany, 26-29 March 1996)*, F. Baader e K. U. Schulz (Eds.), Applied Logic Series 3, Kluwer Academic Publishers, Boston, USA, 1996, pp. 147-160. ISBN 0-7923-4271-2

[18] E.Denti, A.Natali, A.Omicini, M.Venuti. *An Extensible Framework for the Development of Coordinated Applications.* In *Coordination Languages and Models (First International Conference Coordination '96 - Cesena, Italy, April 1996)*, P. Ciancarini and C. Hankin (Eds), Lecture Notes in Computer Science No. 1061, Springer Verlag, Berlin-Heidelberg, Germany, 1996, pp. 305-320. ISBN 3-540-61052-9

[19] E.Denti, A.Omicini. *Designing Multi-Agent Systems around an Extensible Communication Abstraction.* Proceedings of the 4th ModelAge workshop on Formal Models of Agents - Certosa di Pontignano, Italy, January 15-17, 1997. A. Cesta and P.Y. Schoebbens (Eds), Istituto di Psicologia del CNR, Rome, Italy, pp. 87-98, ISBN 88-85059-07-4.
To appear also in the Lecture Notes in Artificial Intelligence, Springer Verlag (2nd quarter 1997).

[20] E.Denti, A.Natali, A.Omicini. *Programmable Coordination Media.* Proceedings of Coordination '97 (Berlin, Germany, 1-3 September 1997). Lecture Notes in Computer Science, Springer Verlag, Berlin-Heidelberg, Germany, 1997. To appear.

[21] D.Gelertner. *Multiple tuple spaces in Linda.* Proceedings of PARLE, Lecture Notes in Computer Science No. 365, Springer Verlag,1989.

[22] P.Ciancarini. *Distributed Programming with Logic Tuple Spaces*. New Generation Computing, 12, 1994.

[23] A. Brogi, P.Ciancarini. *The concurrent language Shared Prolog*. ACM Transactions on Programming Languages and Systems, 13(1), January 1991.

[24] P.Wegner. *Interactive foundations of computing*. Technical Report, Brown University, Providence (RI), August 1996.

[25] Kraig Brockschmidt. *Inside OLE.* Microsoft Press, 1995, 2nd Edition

[26] Sun Microsystems. *Introduction to the Javaspace Model and Terms.* Internet Reference
http://chatsubo.javasoft.com/javaspaces/js-spec/js-intro.doc.html

[27] Sun Microsystems. *The Javaspace Specifications.* Internet Reference
http://chatsubo.javasoft.com/javaspaces/js-spec/js-ops.doc.html

[28] Sun Microsystems. *Javaspace Transactions.* Internet Reference
http://chatsubo.javasoft.com/javaspaces/js-spec/js-txns.doc.html

[29] Sun Microsystems. *Javaspace Utilities.* Internet Reference
http://chatsubo.javasoft.com/javaspaces/js-spec/js-util.doc.html

[30] Sun Microsystems. *Administering Javaspaces.* Internet Reference
http://chatsubo.javasoft.com/javaspaces/js-spec/js-admin.doc.html

[31] T.J. LeBlanc, J.M. Mellor-Crummey, R.J. Fowler. *Analyzing parallel program executions using multiple views.* Journal of Parallel and Distributed Computing, No 9, pp. 203-217, 1990.

[32] D.C. Marinescu, J.E. Lumpp, T.L. Casavant, H.J. Siegel. *Models for monitoring and debugging tools for parallel and distributed software.* Journal of Parallel and Distributed Computing, No 9, pp. 171-184, 1990.

[33] *The Tcl/Tk interface.* Internet Reference. http://www.tcl-tk.com

[34] E.W.Dijkstra. *Co-operating sequential processes.* Academic press, London, 1965.

[35] K. De Bosschere, P. Tarau. *Blackboard-based Extensions in Prolog.* Software Practice and Experience, vol. 26(1), January 1996, pp. 49-69, John Wiley & Sons, Ltd.

[36] E.Denti, A.Natali, A.Omicini, F.Zanichelli. *Robot Control Systems as Contextual Logic Programs.* In "Logic Programming: Formal Methods and Practical Applications", C. Beierke and L. Plumer (Eds.), SCSAI 11, Elsevier Science Publisher B.V., 1994, pp. 343-379. ISBN 0-444-82092-2

[37] F.Zanichelli, S.Caselli, A.Natali, A.Omicini. *A Multi-Agent Framework and Programming Environment for Autonomous Robotics.* Proceedings of the IEEE International Conference on Robotics and Automation, ICRA'94, San Diego (CA), USA, May 11-13, 1994. IEEE Computer Society Press, pp. 3501-3506. ISBN 0-8186-5330-2

[38] K. De Bosschere, P. Tarau. *Virtual World Brokerage with BinProlog and Netscape.* http://clement.info.umoncton.ca/~lpnet/lp-internet/lpnet10/art.html

[39] P. Tarau, V.Dahl, K. De Bosschere. *High Level Logic Programming Tools for Remote Execution, Mobile Code and Agents.* http://clement.info.umoncton.ca/html/remote/art.html