W-ACE: A Logic Language for Intelligent Internet Programming

Enrico Pontelli & Gopal Gupta
Lab. for Logic and Databases
New Mexico State University
Las Cruces, NM 88003
{epontell,gupta}@cs.nmsu.edu

May 28, 1997

Abstract

The development of the *World Wide Web (WWW)* has been considerably delayed due to the excessive complexity of developing advanced and intelligent applications for the Internet. An average application may require the use of different languages, an in-depth understanding of various communication protocols and low-level communication mechanisms, etc. In this paper we propose a logic programming system, called *W-ACE*, extended with various features to support natural, effective, and efficient development of Internet tools. The nature of the constructs introduced in the language makes it particularly suitable to support *intelligent* Internet applications (knowledge-based systems, agents, etc.). W-ACE covers various issues in supporting knowledge-based handling of the World Wide Web, allowing structured and constraint-based management of WWW information, passive and active view of WWW, as well as a powerful support for concurrent applications. The resulting system offers a *general* and *unified* framework for Internet programming, in which the flexibility and high abstraction of logic programming is coupled with general Internet-related mechanisms. Various examples of complex intelligent applications are presented, to underline the simplicity and the power of the proposed ideas.

Keywords: World Wide Web, Logic Programming, Constraint Programming, Modal Logic.

1 Introduction

The *World Wide Web (WWW)* and, more in general, the *Internet*, has grown at exponential rate during the last few years, becoming an indispensable part of every day's life for millions of individuals. This growth has lead the computer science community to focus a great deal of research effort within this application field. This has produced considerable advancement in various areas, with notable results in:

- development of faster and more reliable hardware for supporting network and satellite based communications;
- development of reliable and verified communication protocols;
- development of sophisticate, user-friendly, and esthetically appealing WWW browsers.

On the other hand, this trend of research shows that the focus has been mainly on the lowest and highest level issues for supporting Internet software (communication and high-level applications). It is opinion of the authors, as well as a growing concern in the community [3, 8], that not enough effort has been placed in filling the gap between these two levels.

This gap is mainly caused by the lack of an adequate programming support for the development of Internet related applications. As a result, the task of developing and maintaining this sort of applications is unnecessarily complex, error-prone, and not accessible to average programmers.

The large majority of programs operating on the Internet—e.g., browsers, servers, agents—have been developed by mixing a variety of different languages (e.g., C, Perl, Java), as there is no single language which offers all the adequate features for developing Internet tools. This leads to applications composed of a variety of modules in different languages, interacting through complex and unnatural interfaces.

Furthermore, traditional languages lack some basic features that we deem fundamental for the next generation of Internet applications, specifically:

- *concurrency*: the ability to structure programs as a collection of cooperating components; this is fundamental to allow the development of tools which break free from a single threaded nature (e.g., multiple-connections servers).
- *data structuring*: most of the data currently obtained from the Internet (e.g., Web pages) are seen as unstructured sequences of characters. Thus, there is a strong need for languages capable of structuring and manipulating these data.
- *dynamicity*: existing Web-related applications are seen as programs manipulating static collections of

information. This is unsatisfactory for a variety of situations, where the accessible information need to be sensitive to the pattern of execution.

- *statefulness*: current execution behaviour associated to Web pages is highly limited (e.g., CGI, SSI)—in particular the life-span of an execution is typically limited to a single interaction.

In this project we propose to develop a complete framework aimed at supporting the efficient development and execution of complex Web-based applications. The system, named *W-ACE*, is based on one of the most efficient parallel logic programming systems available, the *ACE* system [33, 30] and takes advantage of the search and symbolic manipulation facilities offered by logic programming [39]. The W-ACE layer is created on top of these basic facilities, and it offers support for

1. symbolic structuring of Web data;
2. complete interfaces for Internet access (HTTP, TCP/IP);
3. a unique language for the development of all the components of an Internet-based application (e.g., Web pages design, CGI scripting, etc.);
4. dynamic structuring of Web information (dynamic links as well as dynamic pages);
5. Web information as dynamic entities: modularization of programs and their use as active data;
6. support for concurrency.

W-ACE can be used as a tool to develop both stand-alone Internet applications as well as extensions of existing software (e.g., Mosaic through Common Client Interface (CCI) [25]).

The advantages of W-ACE over any other existing languages for Internet development are its generality and flexibility. W-ACE combines some original ideas (specific data domains and data structuring, modality-based operations, both server and client side active view) with other ideas proposed in various separate proposals (client side computation, HTML interfaces), to produce a complete and uniform framework. The use of a logic-based language and the choice of operators (e.g., use of modal operators for managing web-related information) makes W-ACE an ideal choice for the development of agents and knowledge-based systems. Furthermore, the parallel capabilities of the underlying ACE system allow the exploitation of the concurrency inherent in Web-applications, enhancing their expressivity and efficiency.

2 Problem Description

The *World Wide Web (WWW)* is a wide-area client-server architecture which is suitable to store and retrieve documents over the Internet. With the term *Web-based application* we indicate any application which draws (part of) its input/output from the WWW. WWW is characterized by:

1. the concept of *Universal Resource Locator (URL)*, aimed at identifying a document on the Internet—specifying its location (host and directory path) and the kind of protocol to be used for accessing it.

2. *protocols* for retrieving documents over the Internet (e.g., *FTP*, *NNTP*). *HTTP* has been specifically designed to support document transfer on WWW.

3. a *document format* (HTML) which supports text, images, and hyperlinks to connect different document's components (hyperlinks are based on URLs).

The WWW is a unique tool for sharing information. On the other hand, the current view of the WWW is that of a giant, highly disorganized database. Retrieval of information is realized in the large majority of the cases using browsers (e.g., Netscape, Mosaic) with fixed capabilities—their only purpose is that of reading and displaying fixed content documents.

Our claim is that the explosive growth of the WWW will soon require more sophisticate tools, ranging from user programmable browsers, intelligent and stateful servers, user-defined agents, etc.—applications which make use of AI and intelligent techniques in order to extract and process information more effectively. The need for this sort of extensions can be easily justified with a simple analogy: the current status of the WWW is analogous to that of a purely extensional database, which is lacking of any mechanism to perform inferences and extract intensional knowledge. Thus, there is an emerging need to give to both programmers and final users the ability to easily manipulate this giant source of knowledge. The first step in this direction can be seen in the growing success of general search engines (e.g., Yahoo, WebCrawler)—which are limited by their general purpose nature, their keyword search behaviour and, especially, their dependence on a prebuilt database of information. And, the number of tools for *Network Information Retrieval (NIR)* is constantly growing [17], to witness the great need for this sort of applications.

In the current setting, developing applications which deal with the WWW requires considerable skills. We can distinguish two classes of applications:

- *Direct Access Applications (DAA):* these are independent applications which perform a direct access to Internet via any of the basic protocols (e.g., FTP, HTTP). *Agents/Softbots* and *browsers* are typical examples of applications in this class.
- *HTML-based Applications (HA):* this class contains applications which rely on the use of HTML-based interfaces—i.e., input and output of the applications is realized through HTML pages and via standard browsers (Intelligent servers, knowledge-based systems, etc.).

2.1 Direct Access Applications

These applications directly communicate with remote servers without the mediation of a browser. Intelligent Agents [27] are a typical example of DAAs, as they independently contact various Web servers, directly fetching and processing the documents obtained. Two major components are required for the development of DAAs.

Support for HTTP-based communications: Traditional programming languages do not offer any support in this direction. Libraries and system calls are present in most implementations allowing TCP/IP based communications (typically via sockets). Nevertheless, the management of the HTTP protocol (construction of messages, interpretation of the headers, etc.) is left completely to the programmer. Some libraries have been recently realized offering a primitive interfacing to HTTP [43], but they appear rather complex and poorly documented. The main exception to the above situation is represented by Java, which embeds a controlled interface to the HTTP protocol.

Symbolic Manipulation of Data: information obtained from a direct HTTP communication can be interpreted at different levels of abstraction. At the lower level information is structured as basic HTTP messages; at the higher level, the communicated information is seen as structured objects (images, HTML documents, etc.). In this sense, none of the traditional languages offer an explicit support to allow structured manipulation of this information. Data extracted from the Internet is seen as unstructured sequences of characters and the management of the structuring is completely left to the programmer. Java itself gives little support in this direction.

Few proposals have been made to overcome part of this problem [8, 3, 38], by developing libraries which offer some abstraction for data organization—notable is the case of PiLLoW [8], which allows for the encoding of an HTML document as a Prolog list of terms (each describing a tagged HTML element). Nevertheless, in virtually all the proposals this abstraction is rather incomplete, as it provides a simple one-to-one mapping to HTML without any operational support for the manipulation of these structures. Furthermore, in all the existing proposals the structuring is limited to the representation of a single HTML document—ignoring the fact that HTML documents are themselves part of a larger structure (collection of documents connected via hyperlinks).

2.2 HTML-based Applications

Web browsers are used more and more often as interfaces towards databases or more complex applications. This allows to obtain appealing interfaces with relatively limited effort, and, on the other hand, to allow remote accesses to selected data and applications. Development of this sort of applications requires three major steps: (i) development of the HTML body; (ii) development of the actual application; (iii) development of the interface between (i) and (ii).

Web Pages Development: Development of Web documents accessible via HTTP protocol is based on a special *markup* language, called *HTML* [34]. HTML supplies a simple language which allows to describe text, its layout, as well as supporting inclusion of images. An HTML program describes a single document page. A document is composed by a collection of pages. Document pages, as well as pages belonging to different documents, are connected through *(hyper)links*. HTML is a relatively simple language. It mainly contains tags which are used to format the content of the document. The special features are:

- *Links*: these are other URLs which may point to either another HTML document, an arbitrary file (FTP protocol), or an executable program.
- *Forms*: HTML forms allow communication with external applications. A form in a document allows the user to assign values to named fields, and to transmit these *name-value* pairs to an associated executable.

Interfaces: At present the principal way offered to a non-expert user to design applications capable of interacting with the web is through the use of the *Common Gateway Interface (CGI)* [18]. CGI is a simple protocol that can be used to communicate between HTML forms and external applications. A plain HTML document has a static structure, which is retrieved as-is by the browser; on the other hand a CGI script is a dynamic entity, which is executed in real-time during the browsing activity. CGI scripts are invoked by a client (browser) and executed on the *server machine*.

The input to the script is the sequence of name-value pairs which have been collected from an HTML form. The output of the computation is going to be consumed by the browser which required the execution of the CGI script—and it is typically coded in HTML.

Observe that the lifetime of a CGI script is limited to the interval of time between its invocation and the moment in which the output is returned to the browser. Once the (unique) output is transmitted, the CGI script is terminated. This makes CGI scripts, in principle, history insensitive and stateless. Scripting languages (Perl, Applescript, TCL, etc.) are commonly used, as they provide parsing and string manipulation capabilities, as well as they are relatively easy to debug and maintain (for small programs).

Connected Applications: While Perl or Javascript can be suitably used for CGI scripting, they are far from being suitable to develop large size applications.

In today's more sophisticated applications which deal with input/output from HTML documents, it is common to have a simple Perl program to interact with the browser, while the main application is written in a traditional programming language (C, C++). The large application will receive its input from the CGI script and return the result to the script itself. Typically, the connected application has to provide the history sensitivity that is lacking in the CGI scripts—i.e., provide a state that is maintained across different connections to the WWW. If this feature is needed, then the connected applications will have a "life of their own", acting as "servers" which process the requests coming from the WWW. If the state is not needed, then the connected application can be directly started by the CGI script itself, and completed before returning the output to the script.

Observe that developing a stateful connected application requires non-straightforward operations; in fact this application is required to communicate with the various CGI scripts that are invoked by the client; since a new execution is created for each CGI call, the connected application should be capable of creating dynamic connections. Traditional languages require the use of either files or sockets to perform communication, leaving to the programmer the task of guaranteeing all the necessary properties (integrity of the communication, sharing of the communication channels between different clients, etc.).

2.3 Discussion

From the previous discussion emerges the following situation: a typical web-based application requires the use of three different programming languages: HTML for the web pages, Perl for CGIs, and a traditional language for the "real" computing application. In the rest of this document we will refer to this issue as the *three-paradigms problem.* Even if the interfaces between the different components are well-defined and standardized, the issue of communication between different components is delicate, and the design of the interfaces represents a considerable part of the whole developing process. This whole system places a considerable burden on the developer, and calls for a *unified* framework which could allow to *effectively,* and *efficiently* tackle all the issues described above.

Limited support has been offered by the current tools in trying to hide the complex issues of communication and data structures, with few exceptions like Java (support for networking and HTTP) and libraries like PiLLoW [8] (HTML structuring) and LibWWW [43] (HTTP interface). Furthermore, traditional languages lack some of the features that are seen as fundamental for the next generation of Internet applications. Internet data (e.g., HTML documents) are required to show a dynamic behaviour, either in their content or in their structure; adequate mechanisms to structure and manipulate them are needed. This data structuring organization should allow a closer interaction between clients and servers, allowing the management of multiple interacting components. This is the direction of Java's *Applets* (server-initiated executions on client's side), Common Gateway Interface (CGI) scripts [18] (client-initiated executions on server's side), and Server Side Includes (SSI) [42] (server-initiated executions on server's side). These schemes are very limited and do not offer sufficient control to the programmer.

Last but not least, very little consideration has been given to the issue of efficiency of Internet related applications. This is justified by the fact that current connected applications are elementary, and their execution time is largely overcome by the communication time. This trend is likely to change, as larger and more complex applications will be hooked to the Internet. Furthermore, servers are ill-equipped to properly handle large numbers of concurrent requests. To tackle these issues concurrency and parallelism are required. Traditional languages are highly inadequate for this purpose [19, 16], as they lack good support for expressing concurrency and communications; logic programming [19, 33] has been proved to be a highly viable programming paradigm to handle this sort of requirements, and the ACE [33, 30] system is one of the most effective logic programming systems developed that incorporates concurrency and parallelism.

3 W-ACE

3.1 (Constraint) Logic Programming

A *logic programming language* [22] is based on a proper subset of first-order logic. The *"data"* manipulated by logic programs is expressed as terms, where a term is either a *variable*, a *constant*, or a *composite structure* $f(t_1, \ldots, t_n)$, where f is a function while $t_1, \ldots, t_n$ are themselves terms.

A logic program consists of a collection of *clauses.* A clause is an entity of the form *head* :-$B_1 \wedge \ldots \wedge B_k$, ($k \geq 0$) where *head*, $B_1, \ldots, B_k$ are *atoms*, i.e., applications of predicate symbols to terms. A clause describes a logical implication (i.e., whenever $B_1, \ldots, B_k$ are true, then *head* is also going to be true).

Selected symbols can be assigned a special meaning, to represent operations over specific domains. For example, $\geq$ and $+$ can be used to represent operations over integer numbers; the other symbols are defined by the programmer using clauses. Atoms built using predefined symbols are called *constraints.* Constraints are interpreted over a predefined interpretation (constraint domains) and they are handled using specialized *constraint solvers.*

Computation in a generic logic programming language is based on the well-known concept of *resolution* [22]. The computation terminates when a query composed only by constraints is produced; these represent the "conditions" that allow the query to be proved valid. As multiple clauses satisfy the requirements to reduce a subgoal, different resolution sequences (also knows as *derivations*) can be created, each potentially leading to a different solution.

The previous description left open the choice of the set of constraint symbols. The first domain we consider, aimed at allowing a higher degree of abstraction, is the *Set domain*. *Sets* are unordered collections of distinct elements. In W-ACE we propose the integration of a very general set domain (as in [12, 6]) as well as the notion of *multiset* [31]. This domain introduces both extensional (e.g, $\{a, b, 1, f(a, X)\}$) and intensional (e.g., $\{x : even(X) \wedge 0 \leq X \leq M\}$) definitions of sets.

More specialized domains will be described in the next sections.

3.2 Overview of W-ACE

The general idea is to offer 3 layers in the computational model.

Object Layer: this is the layer at which many of the schemes available can be placed (large part of PiLLoW, Amzi, etc.). The idea is to extend a traditional language with a set of constructs which are capable of

- interacting with the web (e.g., socket-based communication with browsers/servers, CGI scripting);
- manipulating the HTML trees as object of the computation.

In this level we see the HTML world as the *object* of the computation. HTML documents represent the *data* on which computation takes place.

HTML document representations have been very superficially studied, thus forcing programmers to deal with them in awkward ways (e.g., complex string handling procedures). What we propose is a more precise structural representation, together with an associated set of operations which makes manipulation of pages very intuitive. Clients become more active computing entities, as well as servers can more easily interact with the users. Furthermore, the concept of parallelism and concurrency allows for a more expressive and efficient handling of computations.

Active Layer: The second layer represents the possibility of handling the web pages as programs. Thus they are not anymore passive objects of computation, but they become dynamic and active computations themselves. This follows the lines of proposals like lightweight databases [11] and LogicWeb [23]. This can be structured in two alternative ways, depending on whether the execution of the active pages takes place on the server or client side:

- the *active server* scheme, where the client is a simple browser and the server is a "smart" program capable of following the browsing on the execution of a program (tight connection).
- *active client*: where servers are just supplying pages and the pages are interpreted at the client site as programs to be executed.

Important features:

- dynamic: this scheme offers the possibility to dynamically adapt the computation. This can be realized by parameterizing the computations (i.e., passing arguments) or using self-modifying code;
- concurrent: the notion of concurrency will free the browsing process from its current sequential nature. Concurrency is extremely important to realize interactive computations.

Bridging the Layers: the two layers above are not separate, but they have to be seen as alternative points of view of the same reality. Thus additional mechanisms must be supplied to allow switching from one level to the other. This is equivalent to the concept of *meta-programming*, for which logic programming has been recognized as one of the best paradigms. We will not deal further with this issue, which is still object of research.

3.3 The Object Layer

The object layer is related to developing applications which manipulate information obtained from Internet as *data*. Thus, to stress the point, Internet information is, in this layer, a *passive* entity. Most current proposals in the literature regarding tools for WWW/Internet programming are within this layer of operation. The power of using logic programming to handle object layer applications has already been pointed out in some recent proposals [8, 3, 38].

W-ACE aims at providing the user with a language which allows a more natural management of the structured information deriving from web processing. We will discuss in the next section the additional built-ins that are going to be provided to interact with Internet. In this context we focus our attention on the basic data structure design.

Messages: At the lowest level, information is exchanged in the form of messages which are communicated using TCP/IP protocol. W-ACE supports direct access to this level, encoding messages as appropriate terms (e.g., udp_msg(SockIn,SockOut,Data)). TCP connections are used to support also HTTP communications.

HTML: At the higher level, information acquired from HTTP communications is typically encoded as documents (typically HTML documents). Various existing proposals do not supply any representation for HTML documents. E.g., Eclipse server [3] and Amzi WebServer [38] require the user to deal with explicit HTML, typically coded as a string of characters.

PiLLoW [8] proposed (to our knowledge) for the first time the idea of handling HTML documents as Prolog structures. W-ACE represents (part of an) HTML document as a tree structure (or, equivalently, as a logic term). Each node is labeled with a special identifier indicating the nature of the entities at the next immediate level. The root of the tree for a page is labeled html. Each level can be named either using the corresponding HTML tag (e.g., '⟨li⟩') or using the corresponding LaTeXname (e.g., item). Figure 1 illustrates the tree for a fragment of a document, describing a simple enumeration; on the side the corresponding HTML code is reported. The translation

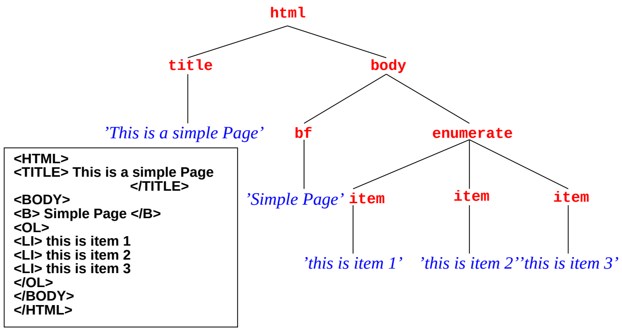

Figure 1: Fragment of HTML Tree

of the the tree into a W-ACE terms follows the traditional syntax of logic (each logical term *is* a tree).

Hyperlinks can be either *external*, i.e. they contain an actual URL, or *internal*, identifying another tree. Each HTML tree is *named*, and the name is automatically associated to the instance of the html functor.

Additionally, the W-ACE language provides very powerful mechanisms for symbolic management of HTML documents—a feature whose need has been largely underestimated in virtually all existing proposal. The final aim is that of developing W-ACE as a *CLP(HTML)*, i.e., a constraint programming language (CLP) capable of dealing with equations (e.g., $=$) and inequations (e.g., $\preceq$) over W-ACE HTML trees. This allows us to perform in a very natural way very sophisticated operations on trees. Equality ($=$) can be used to easily extract components from a page (in the same way in which unification is used in standard logic programming to decompose terms); the constraint nature of the schema allows also to view equality as a constraint on the *possible* structure of the pages created/accessed. $\preceq$ is used to represent the *subtree* relation. This allows, for example to easily identify the presence of a certain structure in a page (without having to "dismantle" the whole page—as required in any other proposed schemes); a W-ACE goal like:

$$\mathsf{Links} = \{\ \mathsf{X} : \mathsf{ref(X)} \preceq \mathsf{Page}\ \}$$

allows to extract in the set Links all the the HTML links (denoted by the tag ref) which appear in the document Page.

Collection of Pages: HTML trees are themselves organized into *HTML graphs*. These originate from the intuition that HTML pages on the web are not separate and independent entities, but are more commonly tightly coupled collections of interdependent information. To our knowledge the concept has not been extensively explored within the framework of languages for Internet/WWW-development. The concept has been successfully used in some domain-specific applications [1, 36].

In W-ACE, pages (i.e., W-ACE HTML trees) are organized into active graphs. Each node of the graph is an HTML document. An edge between two nodes denotes the accessibility relations between the two documents (existence of a link from one page to the other). Edges are directed. The graph is maintained as a multiset of edges and a set of nodes (using the set representation introduced in the previous section). This allows for immediate verification of various properties (connectivity between two nodes, degree of nodes, etc.). HTML graphs themselves can be treated as nodes which are part of a higher level graph, and so on. This allows to impose a hierarchical structure on the HTML documents, controlling at the same time the accessibility of documents (e.g., security).

Each level in the HTML hierarchy (from the level of the tags within a document to the level of classes of classes of documents) is named and various user-defined attributes can be associated to such name. This allows to encode with relative ease complex information structuring schemes, like those described in WebArchitect and PilotBoat [41].

3.3.1 Semantic Features

Part of the operational support required in object layer has already been introduced in the previous section (manipulation of the various data structures). In the same spirit as Java, W-ACE also provides an extensive collection of built-ins for performing a variety of Internet-related operations.

Low-level Communication: W-ACE provides support for low level interprocess communication based on sockets. The built-ins supplied are a generalization of the C-library predicates used to perform socket-based communication. Additionally, W-ACE provides a simple implementation of the HTTP protocol [15]. This is composed of a simple message interface together with various built-ins to build and decode HTTP message, and control the protocol's options (e.g., cache, time-out).

Web-Pages Development: The structure representation given for HTML documents and their organization as HTML graphs can be directly used to develop web pages using W-ACE. Two predicates are available to perform conversion between internal (structured) and external (byte stream/string): html_page and html_graph. The two predicates are bidirectional.

Furthermore, a predicate html_io is available to store an HTML structure into external files (HTML structure to a single file and HTML graph to a hierarchy of directories) and vice versa.

This approach also offers to the programmer an easy mechanism to design programs which automatically generate HTML documents (e.g., an HTML editor) or which perform symbolic manipulation of HTML documents (e.g., filters from/to HTML).

Generic Access: An additional layer is built on top of the HTTP access, supplying a high level programmable interface to the Internet (through HTTP). The generic predicate is called get_url, taking as argument the URL to access. All the parameters for the access are programmable with separate optional predicates. For example, the simple clause in figure 2 expands the current HTML graph by adding all the HTML documents linked by a certain page.

CGI Scripting: Scripting languages, like Perl and TCL, are commonly employed for CGI scripts, as they are very simple to debug and allow to write simple CGI scripts in few lines. Nevertheless, we can argue about why it is worth to pursue different approaches. Scripting languages are notoriously slow, inherently sequential, and unsuitable for activities more complex than simple file and data filtering. For these and other reasons [31], we decided to continue the proposals made by various researchers [8, 5] aiming at providing language extensions capable of integrating CGI scripting with main applications written in high-level languages. W-ACE provides a simple interface for CGI-scripts. Any W-ACE program can be used as a CGI script. The entry point of a CGI script can be an arbitrary predicate with a single argument. Whenever the script is executed (called by a client session), the indicated entry point will be started, and the argument supplied will be initially instantiated to a structure:

cgi_info(*remote_host, remote_adr, server_port, request_method,* Pairs)

where Pairs is a list of name-value pairs passed as input to the CGI script. For example, the input from a form having a single field named client will be coded as

cgi_info('joe.nmsu.edu','128.123.3.4',80,'GET',[client(1)])

The decoding of the input string is performed automatically and transparently. The output of the CGI script is produced using the predicate cgi_output(*type*, *data*).

HTML documents developed using the HTML tree structure can encode calls to CGI scripts by a simple reference to the entry point of the script (assuming, of course, that the CGI script in part of the same program used to develop the HTML tree). For example, the encoding of an HTML tree fragment aimed at obtaining an input from the user is shown below:

```
...
Tree = html(title(X),body(
   input(text,25,name,cgiscript))),
...
cgiscript(X) :-
...
```

where cgiscript is the entry point of the script (a unary predicate in the program).

3.3.2 Applications

In the following sections we will sketch the major ideas behind some applications that can be developed using the object layer support offered by W-ACE.

Simple Forms Handling: An HTML *form* offers to the user the possibility of selecting values to be assigned to a certain number of (internally) named fields. Each form is connected (through an *action* specification) to a CGI script. Once the fields have been filled, the CGI script is executed (and the values of the fields are passed to the CGI).

CGI scripting is extremely simple in W-ACE, as shown in the previous section. Description of forms is relatively simple, thanks to the HTML description features of W-ACE. Single input fields are described using the tag input_field (indicating the type, size, name of the field, as well as a reference to the corresponding CGI script). More complex forms (multiple fields) are described within the form environment. E.g., the following is a description of a simple fragment of a form which selects a field named action (selected as a radio button) and associates it to the CGI script starting at predicate cgiscript:

```
input_action(cgiscript,Reference),
input_radio(Reference,action,
   [out,maci,malk,raci,ralk],
   ['Exit','Metabolic Acidosis',
   'Metabolic Alkalosis',
   'Respiratory Acidosis',
   'Respiratory Alkalosis']),
```

The Web Tourist Advisor: The HTML graph can be used to generalize the notion of browsing history offered by traditional Web browsers (e.g., the *Window History* menu of NCSA Mosaic). In traditional browsers, the window history is a simple linear structure storing the references to the most recently accessed pages, stored in order of access.

During browsing, it is quite common that the user accessing a new page has the desire to explore all the links emanating from such page in a systematic way. Using traditional browsers this becomes a quite cumbersome activity, involving repeated backtracking steps to get the user back to the root page.

W-ACE has been developed as a *Common Client Interface (CCI)* [25] compliant application. This allows W-ACE to exchange information with NCSA Mosaic (another CCI compliant system). This feature can be used to create a simple interface which allows to maintain a graphical representation of the current HTML graph, which changes in sync with the browsing actions performed during a Mosaic session. The

```
update(Page,GraphIn,GraphOut) :-
          Links = { X : ref(X) ≼ Page},
          verify_link(Links,Page,GraphIn,GraphOut).
verify_link(∅,_,GraphIn,GraphOut) :-
          GraphIn = GraphOut.
verify_link(X less Rest,Page,GraphIn,GraphOut) :-
          get_url(X,Tree),!,
          GraphTemp = {(Page,Tree)} ∪ GraphIn,
          verify_link(Rest,Page,GraphTemp,GraphOut).
verify_link(X less Rest,Page,GraphIn,GraphOut) :-
          GraphTemp = {(Page,dead)} ∪ GraphIn,
          verify_link(Rest,Page,GraphTemp,GraphOut).
```

Figure 2: Updating a HTML graph

graphical representation will represent pages as nodes and connect them with edges whenever there is a link between them. The user will be able to detect its current position and move directly to the desired pages.

Stateful Servers CGI scripts have a lifetime that spans a single interaction; as such they do not allow to retain (at least in a simple and *efficient* way) a state across different interactions with the user. The problem can be tackled by storing information in the exchanged documents (using hidden fields)—but this may dramatically increase the amount of information exchanged at every interaction, with consequent impact on the system efficiency. Furthermore, since CGI scripts are restarted at every interaction, this implies having to pay at every interaction the price for starting up the whole system. The use of CGI scripts only as as communication scripts to allow interaction between a browsing session and an external server (different from the server that is supplying the Web pages) is becoming more widespread. The external server has a lifetime which spans multiple interactions, thus maintaining a state across them. Figure 3(i) describes intuitively the structure of the system, while figure 3(ii) illustrates the lifetimes of the different components.

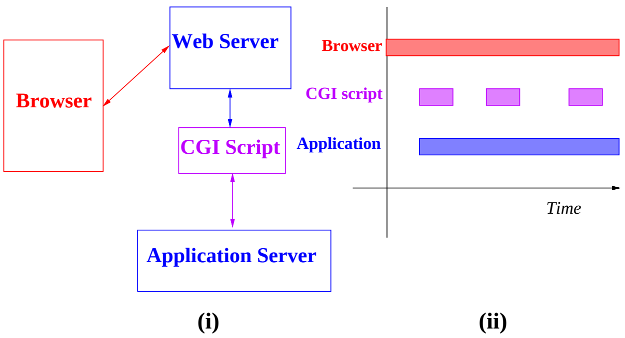

Figure 3: Stateful Servers

Interactions between the application server and the CGI scripts in W-ACE are simple. W-ACE is an inherently concurrent system (based on the ACE parallel system [30, 33]). This allows to spawn server computations which support the state of one interaction with a client. Communication between client and server is transparent and (syntactically) realized via standard read operations (the underlying implementation takes care of the CGI-based communication).

3.4 The Active Layer

The second layer offers the possibility of handling Web pages as *programs*. Thus they are not anymore passive objects of computation, but they become dynamic and active computations themselves.

The use of forms and CGI scripts to process forms' inputs represent one first step towards a more dynamic notion of browsing. Applets in Java represent a further step in this direction, as these are small programs that are automatically executed once the page is visualized by the browser.

In the active layer we make one step further, viewing HTML information as *programs*, which can be both executed and manipulated. This itself can be structured into two possible ways:

1. the *active server* scheme, where client is mainly a simple browsing scheme and the server is a "smart" program capable of following the browsing on the execution of a program (tight connection).
2. *active client*: where servers are just supplying pages and the pages are interpreted at the client site as programs to be executed.

3.4.1 Important Features

The encoding of WWW documents as active entities (*Web programs*) enables implementation of some unique facilities that cannot be offered at the level of object layer.

Dynamicity: logic programming offers the feature of allowing self-modifying programs—i.e., programs can change during their execution (e.g., through built-ins like assert and retract and various meta-programming facilities). Encoding HTML documents as part of programs, and allowing self-modification translates to the possibility of having HTML documents that can adapt themselves depending on the course of action. This is of primary importance to effectively tackle issues like adaptive navigation [7], user modeling, automatic learning.

The principle of dynamic programs can be applied as well to arbitrary computing components of the Internet applications. Thus we can envision servers that can modify their structure during execution (e.g., by learning user's characteristics and appropriately customizing their services), and clients that use the information received to change their own behaviour. The dynamicity of the Internet component can be either:

- *compositional:* the component is parameterized on some modules, which are meant to be generated during the computation. The best example is that of lightweight databases [11], in which HTML documents contains small databases which are *asserted* in the client domain.
- *interactive:* the component is instantiated during a session (e.g., during one interactive session with a browsing client) and self-modifies itself during such session. The modifications disappear once the session is completed. The typical example is that of a server which learns information from the client and consequently modifies its output.
- *permanent:* the component get *permanently* modified during an execution. The best example of this is given by automatic code patching (applications which automatically connect to servers to download upgrades and patches).

Stateful Computations: A state is [10] "Information that a server maintains about the status of ongoing interactions with clients". WWW servers maintain a minimal amount of information about a session (the current page, eventually some information about security access); even their dynamically generated executions (CGI scripts) have a bounded lifespan and as such cannot maintain a state. Analogously, the state kept by the client is limited to the sequence of accessed pages. HTTP is based on a stateless protocol. This problem is evident, for example, in game playing applications. In an interactive game of chess on the Web, the description of the board has to be *explicitly* passed back and forth, and the board have to be explicitly reconstructed every time the server is contacted. This slows-down the execution, makes playing more sensitive to broken connections, and dramatically increases the traffic on the network.

The ability of W-ACE to describe the backbone of an HTML tree as a W-ACE program allows a simple way to maintain a stateful server. Each browser connection can be mapped to a new instance of execution of the corresponding program. Arguments can be passed around during the execution maintaining a tight interaction between server and browser.

In addition, W-ACE allows the creation of *Intelligent Browsers (IB)*. This a simple meta-interpreter layer, written in W-ACE, which filters all the interactions between the browser and the user. The intelligent layer's main task is to replace the dumb caching of traditional browsers with the intelligent local creation of the HTML backbone. The IB generates a W-ACE program during the browsing session, which represents the current session, as well as maintaining information relative to the connection with the correct instances of the server. This is equivalent to treating the browsing session as a client-server execution based on datagrams—thus, the only information that need to be exchanged are those need to keep the synchronization between client and server. This interaction can be made more efficient using a *code-shipping* mechanism (see figure 4). This allows the server to ship to the client procedures that will be run during the browsing session.

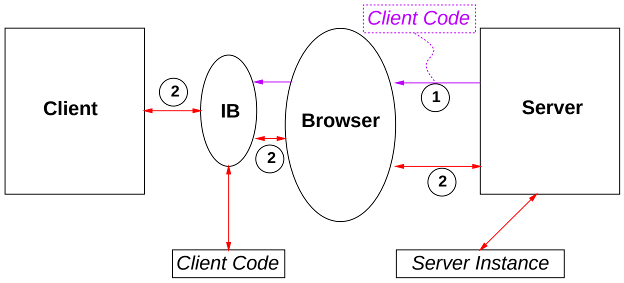

Figure 4: Intelligent Browsing

3.4.2 Active Servers

In the active servers scheme the code associated to dynamic Web pages is executed on the server side. This is consistent with the trend of research shown in various works [18, 42, 26].

Mechanism: W-ACE support for active servers originates from a modified notion of HTML tree and HTML graph. As explained in the description of Object layer, an HTML tree is the term representation of HTML documents and HTML graph is the representation of a cluster of connected HTML documents.

In active layer, the nodes of the HTML graph are goals to be executed in a W-ACE program; intuitively, each "page" is now represented by a fragment of code in the program. The activity of producing a Web page to the client is replaced by the action of executing the corresponding fragment of code.

Observe that this approach is deeply different from both CGI scripting and Applets. In active servers we introduce the concept of *connection* between client and server. A connection is realized when the browser (client) performs the first request to the server. This corresponds, from a "logical" point of view, to generating a server instance dedicated to serve the specific client. Thus client and server instance will proceed in a close lock-step; the server instance will maintain the control, occasionally sending information (e.g., in the form of traditional HTML documents) back to the client for further interaction. Each action of the client corresponds to a specific input supplied to the server instance. Control actions (e.g., selecting an hyperlink, pressing the *Back* button in the browser) correspond closely to actions in the server instance (call to a new predicate, failure and activation of backtracking). Data actions by the client (input of data) are translated into arguments for predicates call in the server instance (e.g., the input of a form becomes the argument for the subgoal associated to the form). The client structure is unchanged, i.e. the activity of the server is transparent to the client.

At the implementation level, there is a need of concurrency in the language to realize the multiple instances of the server, as well as an efficient parallel implementation (consider that most of the time the server is probably waiting for further information from the client, creating the need for suspension/release mechanisms to avoid waste of resources).

Applications: In the active server, the server at every moment in time knows what the client is doing.
Monitoring: the active server can keep track of every action that the client is taking. It knows which pages are accessed (i.e., which procedures of the HTML program have been executed), and which data have been communicated. This allows to create statistics about accesses, as well as implement security policies. For example, the user can communicate to its server instance a password and obtain a validation that allows him to access certain documents that would be otherwise protected.
Fault Tolerance: the server can monitor the communication with the client. If the Internet connection fails during a session, the server instance may save a trace of the execution, and eventually generate an e-mail message to the client, communicating that the session was interrupted. The message may also contain a password used to restore the session once the connection is re-established. This can be realized in the server with a code like:

```
    ...
    html_send(Page,Port),
    wait_server_event(Port,Event),
    (Event = request(_) -> continue(Event) ;
        save_state(Port,ReqId),
        mail_client(Port,ReqId))
        ...
wait_server_event(Port,Event) :-
    get_client_ip(Port,IP),
    ( ping(IP,5) -> Ping=ok; Ping=fail) &
    ( html_get_reply(Port,Event,5) ),
    (Ping = fail -> Event = fail;
        Event = none ->
            wait_server_event(Port,Event);
            true).
```

Intuitively, the wait for a reply from the client is intermixed with the parallel (&—see section on parallelism) test of whether the client host is still reachable (ping is the W-ACE interface to the ping function of Unix). If the host becomes unreachable, then the current state is saved, and a mail message is generated to the corresponding client.

3.4.3 Active Clients

Active clients represent the dual approach to WWW handling w.r.t. the active servers scheme presented before. Active clients involves allowing the client to treat information coming from the Internet as programs to be executed, instead of data to be displayed. In this framework, we are assuming that

- the client (e.g., a browser) has been developed using W-ACE—this doesn't mean that the whole client need to be a W-ACE application, as GUI can be "borrowed" from existing browsers, as long as the they allow to plug in the necessary reasoning component. This is possible, for example, by using NCSA Mosaic as browser.
- The client is a parametric application which is capable of accepting additional modules and perform execution against them;
- a mechanism is used to encode and decode program information (clauses, goals) into HTML documents transmitted by the server.

Traditional programming paradigms (e.g., imperative paradigms) do not allow modification of the structure of a program during the execution. Logic programming languages implementations (e.g., Prolog and its successors) embed operators that allow to add/remove clauses during the execution. This represents the simplest solution to the above mentioned problem. Nevertheless, these program modifying operators (e.g., assert and retract in Prolog), are not the ideal solution, as they are typically rather inefficient and they can operate only one clause at the time.

What we would really like to achieve is extending the logic language with a set of operators that allow various *operations on programs* [4]. Logic programs are treated as sets of clauses, and operators to perform union, intersection, etc. of programs are defined. Thus, for example, the task described above of "extending" the client with a new module can be coded as executing the query with respect to the *union* of the client with the module obtained from the server. Additionally, some of the operators that are required in the context of Internet modules manipulation are not present in the works proposed in the literature [4]; thus, their structure, semantics, and implementation need to be carefully studied. More precisely, we would like to have support to create logical theories, manipulate them, and *reason* on them.

W-ACE Logical Theories: At the active layer, in order to support active clients, we need to design a scheme to encode W-ACE programs within HTML documents. Each HTML document is viewed as the definition of a logical theory (collection of clauses). Each HTML document is uniquely identified by its URL; thus, the URL is used as identifier for the theory described by the page.

The HTML page is parsed to construct the usual HTML tree. The main difference is that the occurrences of the tag script are used to detect the parts of the document that have been used to encode the W-ACE active module. The parsing process, that builds the HTML tree of the page, will take care of generating the correct internal representation of the logic theory. A set of basic facts is also collected in the theory corresponding to each HTML document, containing the description of the page, its location, and other bookkeeping information. The notation *theory*($\langle$URL$\rangle$) allows to access the theory which has been obtained from accessing the URL *URL*.

From the syntactic point of view, logical theories can be thought of as *modules*. Each module is named, and the access to a module through its name is realized as indicated above. The syntactic structure of a module includes an *interface definition* (indicating the imported and exported predicates) and a *module body*.

W-ACE Composition of Theories: Logic theories obtained from the various HTML documents obtained from the Internet can be manipulated by W-ACE programs, using powerful meta-level operations. For the moment we will focus only on the following operators for manipulating logic theories:

(i) the meta-predicate *theory* is available to access the definition of a theory from its name.

(ii) logic theories can be combined either by taking their simple union ($\cup$), or by using an *overriding union* ($\triangleleft$) [4], where, in presence of definitions of the same predicate in different theories, one of them is kept as preferred definition, while the others are discarded.

(iii) the operator ($*$) is used to protect a theory's predicates from being affected by the definitions coming from other theories (i.e., we want to create a closed local scope).

Each theory is, by definition, a *set of closed clauses*. The same predicate names can be present in different theories, and their handling will depend on the combination approach chosen (e.g., $\cup$, $\triangleleft$). Observe that these operators for combining logic theories are sufficient to cover most of the foreseeable cases. For example, $\triangleleft$ can be used to implement both dynamic and static inheritance [35].

W-ACE Dynamic Use of Theories: The next step is to provide the language with adequate mechanisms to allow the use of these logical theories. In this section we explore how to embed the use of theories in W-ACE programs and the specification of executions which are *local* to a given theory.

Execution of a goal using a different theory is expressed using the notation $\mathcal{G}@\mathcal{T}$, indicating that the execution of the goal $\mathcal{G}$ has to be performed taking into account the content of the theory $\mathcal{T}$. This is equivalent for various aspects to the notion of *embedded implication* which has been studied in the context of modular programming [4]. The interpretation given to $\mathcal{G}@\mathcal{T}$ is simply that of attempting a proof of $\mathcal{G}$ in a the current *logical context* (defined later) enriched with the content of the theory $\mathcal{T}$. The way in which the content of $\mathcal{G}$ has to be used is expressed through the use of a prefix operator: if the current context is $\mathcal{P}$, then $\mathcal{G}@\oplus\mathcal{T}$ implies proving $\mathcal{P}\oplus\mathcal{T}\models\mathcal{G}$ (where $\oplus$ is either $\cup$, $\triangleleft$ or $\triangleright$). Observe that, in order to guarantee flexibility in the language, we need to allow nesting of the @ operator—i.e., composition of multiple theories. W-ACE supports [4] a *dynamic* scoping of predicates—thus, the set of clauses available to solve a certain goal can only be determined by following the trace of the execution. *Static* scoping constructs for modular logic programming have been studied [4], but it is our opinion that they are not sufficiently flexible to tackle the issues of handling W-ACE theories.

W-ACE Meta-Engine: The approach of the previous section requires the client program to *constructively* use the logical theories, combining them and executing goals in the resulting set of clauses—e.g., if an HTML document contains the extension of a database, then the best way of handling it is to add it to an intensional component and perform the desired queries. The constructive approach is sufficiently powerful and expressive for virtually any application. Nevertheless, in W-ACE we would like to support more expressive manipulation mechanisms, simplifying the task of moving across different logical theories and avoiding the burden of explicitly combining theories.

Thus, the HTML graph becomes the description of the *accessibility* relation between *possible worlds*. This well matches the semantic structure offered by some traditional forms of modal logic [28]. The advantage is that we can use the HTML graph as a representation of the accessibility relation used to infer truth in a modal structure.

Modal Structure: a typical modal logic programming language extends the traditional logic language with two new components:

(1) modal interpretations [9] are composed by a set

of standard models, called *worlds*, connected via an *accessibility relation*. Truth of a goal is not absolute, as it can change across different worlds.

(2) some high-order operators are added to the language, in order to relate goals validity with the underlying worlds structure—e.g., $\Box G$ expresses the fact that G is valid in all the worlds which are accessible from the present one.

In the context of W-ACE the accessibility relation is common to all the modal operators used in the language. The intuition is that the logical theories obtained from the HTML documents represent different worlds and the HTML graph is used to supply the accessibility relation.

Modality in W-ACE: The accessibility relation, originating from the HTML graph, is assumed to be *reflexive* (self-accessible), *direct* (i.e., in general non-symmetrical), and *transitive*. Each possible world is obtained from one of the logical theories $\mathcal{T}$ combined with the main program ($\mathcal{T} \oplus \mathcal{P}$). The operator for the combination is by default $\cup$, but can be modified ($\lhd$ or $\rhd$) with appropriate declarations.

The HTML graph present in the W-ACE execution is typically only a finite window over the potentially infinite branching of links over the Internet. For this reason, W-ACE supports two classes of operators:

- *local operators:* in which the logical theories that belong to the current HTML graph represent the possible worlds. Thus, the modal operators do not follow any further link from such theories toward other ones.
- *global operators:* in this case the set of possible worlds are represented by all the pages that are reachable from a given entry point. The operator will take care to automatically access any additional pages as required.

The programmer will be responsible to guarantee a correct usage of global operators. The following modal operators are supported in W-ACE:

- $\Box$ representing the validity of a goal in all the accessible worlds;
- $\Diamond$ representing the existence of an accessible world in which the goal is valid;
- $\bigcirc$ representing the validity of the goal in all the worlds accessible in one step from the present one.

Each operator can be parameterized as follows:

- $\oplus^{\uparrow}$ and $\oplus^{\downarrow}$ representing respectively the global and local versions of the operator $\oplus$;
- $\oplus_i$ representing the fact that the operator is meant to be restricted on all the worlds that are reachable within i steps from the current one.

Additionally, the computation can be placed in a specific world by using the above mentioned @ notation.

Using these operators various applications can be expressed in a few lines of code. For example, in a diagnosis program (as the one mentioned in a later section), each HTML page represents different assumptions about the values of certain laboratory results. Thus, starting from a page, we can enquire whether the current symptoms can lead to the diagnosis of a certain illness with a simple application of $\Diamond$.

Applications: Intelligent clients which can self adapt to the Web-pages accessed (e.g., interpreting documents having a special format).

Lightweight Databases: The intuition behind lightweight data-bases is to view HTML documents as small databases, and allow clients to act as intelligent (e.g., deductive) query solver over these databases. The concept, introduced by Dobson [11] and generalized by others [23, 8], can be implemented with extreme ease in the active client model offered by W-ACE. The database information are stored in HTML documents, as a sequence of facts (for example). Once a page is loaded, it is turned into a W-ACE theory, against which the client can pose queries. The same can be generalized to include complex knowledge-bases within HTML documents (e.g., part of rule-bases for an expert system).

Active Programs e.g. normal programs that at the beginning of their execution accesses some URL to update themselves. This would allow, for example, to solve the problem of distributing patches for software—the program itself will connect to the appropriate Web server before starting execution and eventually download and apply automatically the patches.

Adaptive Browsing The idea is to allow the structure of the document to change due to the interaction with the user. This can range from changing the structure of the HTML tree during the browsing process (e.g., collapse the HTML tree), to performing more extensive modifications, like in the *Adaptive Hypermedia Systems* [13]. We can distinguish two levels of adaptivity:

- adaptive presentation: the *content* of the HTML tree is changed depending on the user model. This is easy to model in W-ACE, thanks to the possibility of parameterizing (with a simple selector argument) the execution of the HTML program that has been downloaded, producing "alternative" pages.
- adaptive navigation: the *structure* of the HTML tree is modified depending on the user model.

The adaptivity of the scheme can be varied also by accessing separate *databases of links*. Adaptation can be performed by keeping a *user model* [29]. Limited applications of these ideas have been realized in some AI projects [14], capable of performing automatic generation of Web pages.

3.5 Concurrency and Parallelism

Parallelism and *concurrency* are two separate concepts. *Parallelism* indicates ability of performing parallel execution of different components of the computation. Thus, its main aim is *efficiency*. *Concurrency* indicates the ability to express concurrent execution of different components of the program. This is a *semantic* issue. Both the issues are of primary importance in the context of Internet-based applications. Management of multiple agents takes big advantage from parallelism, while interacting sessions or managing multiple connections represent concurrency issues.

Web-related applications are going to become, in our opinion, one of the major areas of application of techniques for parallel execution. In the rest of this section we will analyze the two different forms of parallelism that we envision as most promising for W-ACE. We distinguish between *Intra-Parallelism*—exploitation of parallelism within a single browsing/serving activity—and *Inter-Parallelism*—exploitation of *"parallelism"* between separate Internet sessions.

3.5.1 Intra-Parallelism

The concept of intra-parallelism originates from the exploitation of parallelism within a single browsing/serving activity. In both cases, this form of parallelism can be obtained by using the basic support for parallel execution which is supplied by the ACE parallel system. The ACE system supports both and-parallelism and or-parallelism in logic programs.

And-parallelism attempts parallel execution of different subgoals belonging to the current query. Thus, in a computation like

$?- extract\_links(L, Page), extract\_pics(P, Page)$

the two subgoals can be solved in parallel. It is common [33] to distinguish two flavors of and-parallelism: *independent parallelism* which originates when the parallel subgoals are not allowed to affect each other's executions [30], and *dependent and-parallelism* where [32, 20] subgoals executed in parallel are allowed to share variables and compete for their bindings.

Or-Parallelism allows parallel execution of the alternatives associated to a resolution step—i.e., allows to explore in parallel the different clauses that can be used to reduce the selected subgoal. Thus or-parallelism is a mechanism to speedup the exploration of the search space generated by a logic program.

In the previous work on ACE we have studied mainly the issue of parallelism. Nevertheless, the results we have achieved have the nice side-effect of offering a very flexible language to express concurrency, as well as very efficient mechanisms to implement it. In addition, as soon as extra processors become available, concurrency in ACE becomes real parallelism. In ACE, concurrency emerges from the fact that an intermediate language is supplied which allows to express points where parallelism can be exploited—i.e., notation that allows to define program components that can be logically exploited in parallel. The concurrent language supported by ACE consists of the following components:

- *and-concurrency:* the notation $p(X)\&q(Y)$ is used to denote conjunctions of subgoals in which the two subgoals can be run in parallel. The notation becomes slightly more complex if we allow dependences between the concurrent subgoals (we need to indicate the source of dependences—either manually or through a static analyzer). Dependent execution in ACE is structured in such a way to respect the sequential semantics of the language [32]. Nevertheless, W-ACE allows us to specify for each argument position in a concurrent subgoal whether such position should be treated as a producer or consumer position. This can be supplied simply as mode declarations (in the same fashion as in Parlog), or by explicitly describing the dependence graph to be used during execution. This feature is extremely powerful, as it allows to create separate computations and to use variables as unidirectional communication channels.
- *or-concurrency:* ACE admits the declaration parallel to indicate which procedures of the program have to be considered for or-parallel execution.

An and-concurrent execution has the following semantics: given a query like $(p(X)\&q(Y)), r(X)$, then $p(X)$ and $q(Y)$ are concurrent, while $r(X)$ is constrained to start its execution only after both p and q have been completed. Thus, an implicit barrier is present at the end of the parallel call. There are situations where we do not want such barrier to be present. The operator $\bar{\&}$ can be used to this purpose—in $p(X)\bar{\&}q(Y)$, the first subgoal to complete will proceed with the continuation of the parallel call, while the other will complete and disappear.

Regarding or-concurrency, W-ACE admits an extension that is currently present also in ACE: the possibility of declaring a procedure to be reactive. This is aimed at capturing the behaviour typical of guarded commands. A reactive procedure implies that, at the moment of clause selection, the selection process will not consider one clause at the time, in their textual order, but it will loop through the different clauses in the procedure until a successful head unification will be completed. This follows the same idea behind Hoare's non-deterministic guarded commands. This, combined with the use of producer/consumer and-parallelism (as described above), allows the development of servers which can react to different kinds of stimuli coming from the clients.

And-Parallelism in Web Applications: Executions over recursive data structures, data-parallel computations, producer/consumer pipelined computations are all examples of executions which are very likely to offer large amounts of and-parallelism [30].

More specific to the case of W-ACE, the explicit use of notations to indicate concurrency can be directly translated to parallel executions. This is typical, for example, of multiple agents which are exploring different set of HTML documents searching for certain patterns/keywords. Another example is represented by efficient servers which are implementing various forms of "distributed" management of data. Queries arriving from client applications can be

- partitioned between different servers that are maintaining different databases—this does not involve anything more than an independent form of concurrency;
- processed through a pipelined organization of multiple servers—this requires communication between different active entities, i.e., a form of dependent and-concurrency.
- assigned to independent instances of servers, to allow overlapping of requests handling.

Or-Parallelism in Web Applications: In general or-parallelism is associated to the concept of *search*. Thus most of the applications which involve a sufficiently complex search are suitable to this form of parallelization. Typical examples are:

- agents which are performing a search over the nodes of the HTML graph; alternative paths emanating from a given node can be explored in parallel;
- analysis of data often requires non-deterministic steps—e.g., parsing data to identify their structure with respect to a given structural description (e.g., a grammar, a finite state automata);
- searching for an element from a given set which satisfies a certain property (e.g., looking for dead links between the set of all links in an HTML document).

Modal Parallelism: An additional form of parallelism emerges from the introduction of modality in the language. The operations to be performed across different worlds can be easily cast into parallel executions. *Necessity* modal operators ($\Box$) require the verification of a goal across all the worlds belonging to a certain modal interpretation (e.g., all the logical theories currently accessible through the HTML graph); this verification can be done in parallel (exploring concurrently different worlds). This is a generalization of and-parallelism, and the W-ACE system can be easily extended to accommodate for it. *Possibility* modal operators ($\Diamond$) are required to verify whether a given goal holds in at least one of the accessible worlds; this can also be attempted in parallel, by considering alternative accessible worlds in parallel. This can be seen as a generalization of or-parallelism.

3.5.2 Inter-Parallelism

The possibility of expressing parallel computations and communication assumes extreme importance when dealing with different executions which should be capable of interacting. The mechanisms required for this sort of computations are the same described in the previous section. W-ACE goals represents the different concurrent computations while *shared* variables are used as communication channels between them.

These mechanisms are actually sufficient by themselves to cover most of the interesting cases emerging from Web-related applications. Nevertheless, there are some new issues that have been recently emerging within the Internet community which require more sophisticate mechanisms:

- the notion of a single server supplying HTML documents to the various clients is progressively overcome by the notion of servers providing more comprehensive services (e.g., access to knowledge bases). The need to perform complex computations as answers to client requests may require the need to allow servers to distribute computations across different connected hosts. Analogously, client themselves can use HTTP protocol to distribute a computation between different computing resources. This may require assigning execution of certain computations to specific resource (e.g., a specific host).
- in various situations (e.g., independent agents that decide to combine their results) we can envision the need to combine independent computations. W-ACE provides this feature, through the predicate `meet`. A careful memory organization allows to combine with minimal effort the memory images of two W-ACE workers.

3.5.3 Serving HTML Clients

The problem of having a server handling multiple clients through Web-connections has been largely underestimated. The issue, as pointed out by Szeredi [40], is of extreme importance: modern "intelligent" servers are AI-like applications, which are large and expensive; maintaining a separate running copy of the server for each client connection may not be a suitable solution, since

- the impact of the multiple running computations on the server's side may be unbearable;
- keeping separate server processes makes interactions between them (if necessary, e.g., accessing and modifying a common database) extremely challenging to program and expensive to maintain.

On the other hand, the ideal situation of handling the different connections within the scope of a single server can be as difficult to realize within a sequential programming framework—and can easily lead to inefficient systems (which may force an unnatural sequentiality between clients' requests).

The potential of using parallelism to serve different browsing sessions has already been pointed out in the literature. One good example, within the area of Prolog, can be found in Szeredi's work [40]. Szeredi suggested the use of the Aurora Or-Parallel system to handle this situation, preserving the declarative nature of the program and with a good level of efficiency.

W-ACE for Remote Diagnosis (Example): The application we considered is an intelligent system for the diagnosis of Acid-base disorders [2]. The system was originally developed in standard Prolog and was capable of supporting a single interactive user session. Purpose of the program is to acquire successive details of data from the user in order to produce one or more medical diagnosis.

The program was transformed into a Web-based application using the preliminary W-ACE prototype (the whole process took hardly a day of work). The original organization of the resulting system is shown in figure 5. The diagnosis server is an independent running application, which listens to a socket for queries and data coming from the HTML pages (through a small CGI script, hidden in the W-ACE interface).

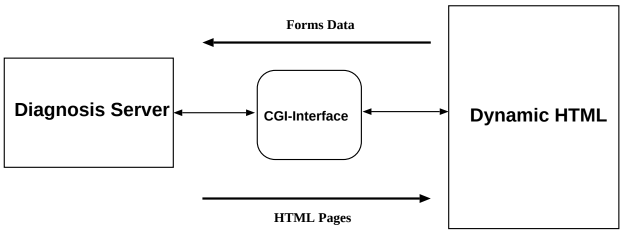

Figure 5: General Structure of Diagnosis System

The server in this scheme is capable of serving a single client session at the time, suspending every time a client input is required. A tight interaction between HTML browsing and server execution is kept (using backtracking to "follow" the user if he decides to use the `back` button of the browser).

The serious problem in this context is serving multiple clients. Various solutions to this problem have been envisioned in [40], ranging from developing a server capable of directly handling multiple sessions (very complex and leading to a loss of the logical structure of the server), to using complex logic features (`freeze`, `continuations`). Neither of these solutions is particularly attractive.

The different sessions are *separate threads* of computation. Furthermore, in order to make possible an efficient usage of resources, the different threads should *share* the same resources as much as possible (thus, spawning a new process for each client connection is not a viable solution).

In [40] an solution to a similar problem is proposed using an *or-parallel* Prolog system (Aurora). The solution suffers of various drawbacks:

1. it is not particularly elegant, since different client connections have to be encoded as separate "alternatives" of a choice point;
2. alternatives of a choice point are *necessarily* independent, and cannot directly communicate. This means that only independent client sessions can be directly modeled in this scheme. Communications can be realized in a very unnatural (and *expensive*) fashion through the use of unsynchronized side-effect predicates (`assert` and `retract`).
3. last but not least, or-parallelism does not gracefully offer ways of suspending branches (e.g., while waiting for some input from the client). Aurora's developer have tried to solve the problem by introducing new explicit built-ins to suspend and resume branches.

The solution in W-ACE is very natural. Multiple client sessions can be easily handled as separate subgoals, which are executed in and-parallel—which better capture the "coexisting" nature of client sessions, instead of the unnatural "alternative" nature attributed in the Aurora scheme. A loop like:

```
main(Sockets,Args) :-
  wait_connection(Sockets,Inst),
  ( client_execution(Sockets,Inst,Args) &
     main(Sockets,Args)
  ).
```

is used. Suspensions are automatically obtained thanks to the suspension mechanism used by dependent and-parallelism [32]. In the specific case of the diagnosis application, the different client sessions are independent—thus the `Args` argument is actually empty. Communication with the browser is through separate sockets (one per worker) and is completely transparent to the programmer. The preliminary version of the Web-based diagnosis system has been completed (figure 6 shows a snapshot of an execution).

Interacting Clients (Example): In the previous example client sessions are non-interacting and independent (same assumption as in [40]).

There are many cases in which it is desirable to allow clients to interact with each others. This may range from the simple *indirect* interaction caused by accessing and modifying a common database, to the more complex case of a *direct* client interaction, as in a multi-players game. Schemes like the one used in Aurora do not support this pattern of execution—it can be realized only using additional expensive mechanisms (communication through common files). Traditional languages (e.g., Java) require complex and explicit communication mechanisms (sockets, IPC messages).

On the other hand, the scheme adopted in W-ACE allows to avoid any additional complexity. Since each separate client connection is represented by a separate recursive subgoal, they can directly interact via shared variables [32]. We have tested this feature in a simple server application which allows two client sessions to play tic-tac-toe one against the other.

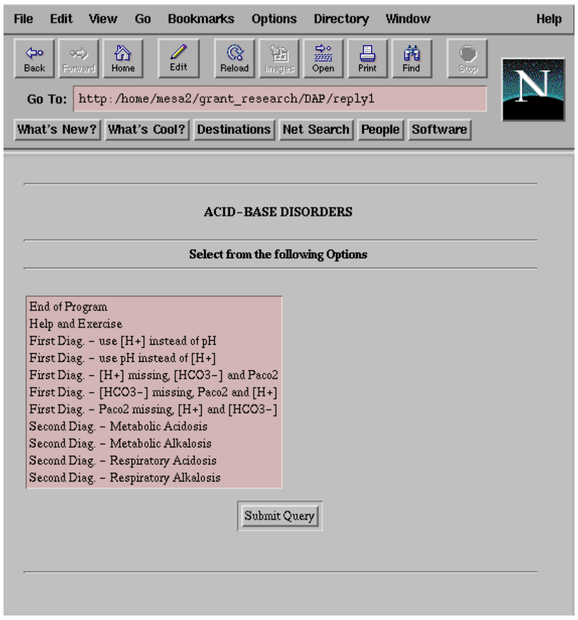

Figure 6: Snapshot of Execution of Diagnosis Program

3.5.4 Applications

Communicating Agents: Interacting agents has been for long a relevant topic in the AI community; within the framework of the WWW, this concept assumes a great importance; the search space for agents traveling on the Web is extremely large, and concurrent, interacting agents may lead to drastic reductions in search time. Using traditional programming languages, the practical development of communicating agents is a very challenging task (e.g., dealing with threads, communication, etc.).

W-ACE offers a framework where managing communicating agents is extremely simple. Each agent is represented by a goal (i.e., a call to a predicate defining the agent); parallel execution is readily available through the and-concurrent support; communication is directly available through shared variables. For example, a W-ACE top-level call to generate two communicating agents will be as follows:

```
... mark([Ch1,Ch2]),
    In1=Ch1,In2=Ch2,
    Out1=Ch2,Out2=Ch1,
     (agent(1,In1,Out1) &
      agent(2,In2,Out2)),...
```

The two unidirectional communication channels (`Ch1,Ch2`) are set between the two agents. For example, agent 1 will send a message to agent 2 by instantiating its `Out` channel in the following way: `Out = [Message | NewOut]`, where `Message` is the message to be sent and `NewOut` is a new variable (representing the "new" access to the output channel).

Generalized Browsing: The browsing activity is strictly a single user-single step execution. The user moves from one page to the other, but at any time he is focused on only one document. Availability of concurrency allows for various extensions:

1. *Concurrent browsing:* allows a single user to concurrently explore different paths of the HTML graph. This is easily realized by having a client spawning separate processes to carry on the navigation of the documents. The separate sessions may be independent (i.e., they separately report results in different parts of the user interface), or the may interact during the navigation—for example, one of the navigation sessions may attempt to explore a document which has been explored previously by another session—this can be avoided by having a shared representation of the HTML graph between the different navigations.

2. *Cooperative browsing:* it is also important to allow separate navigation sessions, eventually initiated by different users, to interact. A good example of this is represented by the principle of *annotating* HTML documents [36, 31]. Members of a working team may be separately operating via different browsing sessions (e.g., they are browsing the Web in search of information on a given topic). The members of the team, even if they are carrying on independent activities, need to interact, in order to avoid re-visiting twice the same document and to allow a member of the team to gather information regarding documents visited by other members. This can be realized using communicating browsers (e.g., shared database of annotations, shared representation of the visited HTML graph, etc.)

4 Related Work

Very few proposals have been made in the development of a unified programming framework for Internet programming. Various libraries have been developed to access the WWW, communicate via HTTP, simplify CGI scripting, but these represent only partial answers to the problem—furthermore, the supplied APIs are rather complex and require the programmer to have a thorough understanding of the underline communication protocols.

The goal of actually supplying frameworks capable of hiding most of the communication and syntactic details has been achieved in very few cases. Java is the most notable example of a language attempting to bridge over some of the complex issues related to Internet programming. Nevertheless, Java suffers the limitations of traditional imperative programming languages (e.g., very poor symbolic manipulation capabilities, high complexity in dealing with concurrency issues) as well as a rather inefficient implementation.

Various proposals have emerged recently in extending logic programming to support Internet programming, recognizing the unique power of this programming paradigm for this sort of applications [23, 8, 40, 38, 3]; similar attempts are also under exploration in the area of functional programming [5, 24]. Nevertheless, most of these limit themselves to interfaces to some of the required communication protocols, without tackling the more general issue of developing an actual programming paradigm for the WWW. The only exceptions are LogicWeb [23], which shares various similarities with the W-ACE client-side active layer, and the Fox project at CMU, aimed at extending ML to support Internet development.

Nevertheless, W-ACE is to our knowledge one of the first proposals to attempt handling all the issues related to intelligent programming for Internet within a single framework. The system is currently under development, as an extension of the already mentioned ACE parallel logic programming system. Although still in its very preliminary stages of implementation, parts of W-ACE have already been used (as mentioned in the previous sections) to successfully develop WWW applications, with very encouraging results.

5 Conclusions

This project aims at investigating and developing a unified programming framework for Internet programming. The key is to support an environment in which the programmer does not need virtually any knowledge of the complex details of the communication protocols involved, as well as he is provided with a set of domains and computational tools allowing communication and processing of WWW data at a very high level of abstraction. This system is based on logic programming, and maintains a strong similarity to Prolog in terms of program syntax. Thus, a user needs to know only Prolog to develop very powerful Internet applications, as opposed to a variety of languages and tools (e.g., Java, HTML, HTTP, CGI, etc.).

We strongly believe that our approach will produce a relevant advance in the state-of-the-art for Internet-based programming, yielding an efficient programming environment, usable by average programmers.

We also aim at showing how various techniques proposed in different frameworks can have a substantial impact in helping developing Internet related applications (e.g., modal logic, modularization, constraint management). We also provide what we believe is a novel view of the WWW—no more simple linked collections of static data, but active and interactive computations. Various researchers approximated this notion, without accomplishing it in its full generality. It is our intention to show that this is the best direction in which WWW research should move to tackle the new computational requirements.

The large availability of small multiprocessor machines (also in the class of Personal Computers) and the ability of W-ACE to take full advantage of this computing power (converting the inherent concurrency present in Internet applications into actual parallelism) represent an ideal solution to what is going to become a serious issue in Internet computing, the bottleneck of servers unable to adequately serve a large number of requests.

The target of this system is the non-specialized programming community, as it allows to develop Internet capable applications without the need of an in-depth knowledge of the various HTML, HTTP, CGI, etc. It will be of fundamental importance for people willing to develop ad-hoc applications, as well as a tool to make existing applications remotely accessible via Internet.

The research we are proposing fits very naturally within the scope of logic programming; nevertheless, we strongly believe various of the proposed ideas can be ported also to other programming frameworks, opening the doors for a more general view of WWW and Internet.

References

[1] K. Andrews et al. *Serving Information to the Web with Hyper-G.* Tech. Rep., Graz Univ. of Technology, 1996.

[2] J.W. Bang and G. Gupta. A Logic Programming System for Diagnosing Acid-base Disorders. *Conf. on Practical Applications of Prolog*, 1997.

[3] P. Bonnet et al. Towards ECLiPSe Agents on the Internet. *Workshop on Logic Progr. Tools for Internet*, 1996.

[4] M. Bugliesi et al. Modularity in Logic Programming. *Journal of Logic Programming*, Vol. 19/20, 1994.

[5] J. Buhler. A Language-structured Approach to Networking Software. Tech. Rep., Carnegie Mellon University, 1996.

[6] P. Bruscoli, A. Dovier, E. Pontelli, G. Rossi. Negation and Intensional Sets in CLP. *Int. Conference on Logic Programming*, MIT Press, 1994.

[7] P. Brusilovsky. Methods and Techniques of Adaptive Hypermedia. In *Proc. of UMUAI*, 1996.

[8] D. Cabeza et al. The PiLLoW/CIAO Library for Intenet/WWW Programming. *Workshop on Logic Progr. Tools for Internet*, 1996.

[9] B.F. Chellas. *Modal Logic: an Introduction.* Cambridge University Press, 1980.

[10] D.E. Comer et al. *Internetworking with TCP/IP Volume III.*. Prentice-Hall, 1994.

[11] S. Dobson et al. *Towards Improving Automation in the World Wide Web.* Tech. Rep., Rutherford Appleton Lab., 1995.

[12] A. Dovier, E. Omodeo, E. Pontelli, G. Rossi. {log}: a logic programming language with finite sets. In *Journal of Logic Programming*, Vol. 28, No. 1, 1996.

[13] P. Duchastel. Integrating Hypermedia into Intelligent Tutoring. *Hypermedia Courseware*, Springer Verlag, 1991.

[14] F. Espinoza and K. Hook. An Interactive WWW Interface to an Adaptive Information System. *Proc. of User Modeling*, 1996.

[15] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1.* Request for Comment: 2068, January 1997.

[16] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1994.

[17] J. Foster. *A Status Report on Networked Information Retrieval: Tools and Groups.* RFC-1689, 1994.

[18] M. Gleeson et al. Beyond Hypertext: using the WWW for Interactive Applications. *AusWeb Conference*, 1997.

[19] G. Gupta. *Multiprocessor Execution of Logic Programs.* Kluwer Academic Press, 1994.

[20] G. Gupta and E. Pontelli. Extended Dynamic Dependent And-Parallelism in ACE. *Int. Symp. on Parallel Symbolic Computing*, ACM Press, 1997.

[21] J. Jaffar and M. Maher. Constraint Logic Programming. *Journal of Logic Programming*, Vol. 19/20, 1994.

[22] J.W. Lloyd. *Foundations of Logic Programming* Springer Verlag, 1987.

[23] S. Loke et al. Lightweight Deductive Databases on the WWW. *Workshop on Logic Progr. Tools for Internet*, 1996.

[24] J.C. Mallery. A Common Lisp Hypermedia Server. *First International WWW Conference*, 1994.

[25] NCSA. *Application Programmer's Interface for the NCSA Mosaic CCI.* Technical Report, 1995.

[26] NeoSoft. *Inside NeoWebScript.* User Manual, 1997.

[27] H.S. Nwana. Software Agents: An Overview. In *Knowledge Engineering Review*, Vol. 11, No. 3, 1996.

[28] M.A. Orgun and W. Ma. An Overview of Temporal and Modal Logic Programming. *Int. Conference on Temporal Logic*, Springer Verlag, 1994.

[29] A. Paiva, J. Self, R. Hartley. Externalising Learner Models. In *Proc. of AIED95*, AACE Publications, 1995.

[30] E. Pontelli et al. &ACE: A High-performance Parallel Prolog System. *Int. Parallel Processing Symp.*, IEEE Computer Society, 1995.

[31] E. Pontelli and G. Gupta. *Using ACE as a Parallel Engine for WWW Applications.* Internal Report, NMSU, 1996.

[32] E. Pontelli and G. Gupta. Implementation Mechanisms for Dependent And-Parallelism. *Int. Conf. on Logic Programming*, MIT Press, 1997.

[33] E. Pontelli and G. Gupta. Parallel Symbolic Computing in ACE. *Annals of Mathematics and Artificial Intelligence*, (to appear).

[34] D. Raggett. *HTML 3.2 Reference Specification.* W3C Recommendation, January 1997.

[35] U. Reddy. Objects as Closures: Abstract Semantics of Object Oriented Languages. *Proc. Lisp and Functional Programming*, ACM, 1989.

[36] M. Röscheisen et al. Beyond Browsing: Shared Comments, SOAPs, Trails, and On-line Communities. Tech. Rep., Stanford University, 1996.

[37] S. Sato. Dynamic Rewriting of HTML Documents. *First International WWW Conference*, 1994.

[38] A. Sehmi, M. Kroening. WebLS: A Custom Prolog Rule Engine for Providing Web-Based Tech Support. *Workshop on Logic Progr. Tools for Internet*, 1996.

[39] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1996.

[40] P. Szeredi et al. Serving Multiple HTML Clients from a Prolog Application. *Workshop on Logic Progr. Tools for Internet*, 1996.

[41] K. Takahashi, E. Liang. Analysis and Design of Web-based Information Systems. In *Proc. of the Sixth International World Wide Web Conference*, 1996.

[42] M. West *Server Side Includes.* Technical Report, Questar Microsystems Inc., 1997.

[43] World Wide Web Consortium. *LibWWW User Documentation.* Technical Report, 1997.