An Agent-Based Approach to Web Site Maintenance*

Wamberto W. Vasconcelos[1] and João Cavalcanti[2]

[1] Department of Computing Science, University of Aberdeen
AB24 3UE Aberdeen, United Kingdom
wvasconcelos@acm.org
[2] Department of Computer Science, University of Amazonas
69077-000 Manaus, AM, Brazil
john@dcc.fua.br

Abstract. Web sites are public representations of corporations, businesses and governmental bodies. As such they require proper maintenance to ensure that accurate and updated information is presented adequately. Dynamically created Web sites sometimes are not an option: they add a computational overhead on the server and make the automatic indexing of pages difficult. Static Web sites may grow to an extent that manually performing maintenance operations becomes unfeasible: automatic or semi-automatic means ought to be put in place. In this paper we explain one such approach, using software agents to look after large data-intensive Web sites. Our maintenance operations are performed by a team of autonomous agents that communicate with each other as well as with Web masters or other human agents. Our approach can be naturally incorporated into existing Web sites and its use can be gradually extended to encompass larger portions of the site. Because our agents are independent, their individual malfunctioning should not stop the maintenance effort as a whole.

1 Introduction

Web sites provide to the public at large representations of corporations, businesses and governmental bodies. Increasingly Web sites provide the first (and sometimes only) form of contact between the general public and the actual organisation. It is imperative that proper care and attention be placed into the *design* of the Web site: the look-and-feel of pages, the provision of maps for the site, the amount and positioning of information, and so on [1]. However, an independent and equally important (or even more important) issue concerns the actual *contents* of the Web site: how accurate and updated the information presented is and what mechanisms are in place to support their maintenance. This is a particularly sensitive issue in data-intensive Web sites, where data suffers constant or frequent updates and there may be many sources of data involved.

* Partially supported by the Brazilian Research Council (CNPq) grant no. 55.2197/02-5 (Project SiteFix – Adapting Web Sites to Perform Information Retrieval Tasks).

N. Koch, P. Fraternali, and M. Wirsing (Eds.): ICWE 2004, LNCS 3140, pp. 271–286, 2004.

We propose an automatic approach to Web site management via a team of software agents: a collection of independent and communicating pieces of software will share the responsibility and computational effort of looking after a large data-intensive Web site, managing static Web pages. We introduce an abstract and generic architecture and describe how it can be implemented using existing technologies and standards.

Dynamic-page techniques [2,3] have been used to tackle the contents maitenance problem. However, static pages still have an important place as they are simpler to manage, require less computational power and are visible to search engines. It should be clear that we do not propose a replacement for dynamic pages: information which usually resides in a database or that needs some computation before presentation are clearly better addressed via dynamic generation. Our approach targets pieces of information which do not require complex computations, changing only their values and, in many cases, not being stored in databases. Both approaches should co-exist and be used in a single Web site application.

This paper is organised as follows. In the next section we address some of the issues related with using agents to carry out Web site maintenance. In Section 3 we explain our proposed architecture and its components. In Section 4 we present an example to illustrate our approach. Section 5 contrasts our approach with existing work and in Section 6 we draw conclusions and discuss the work presented.

2 Web Site Maintenance with Agents

The idea to employ autonomous agents to perform Web site maintenance comes from the fact that maintenance tasks are usually small, well defined and recurrent. To our knowledge, no-one has attempted this before: simple and robust software agents can be designed to carry out stereotypical tasks and be reused and customised to suit particular needs.

Typical maintenance tasks our agents can handle are those involving updating a piece of information and publishing it on a Web page. In order to specify this sort of task, it is only necessary to specify the information item, its data source, the frequency of update and the page where it is published. As a result, the amount of knowledge agents need is relatively small. Agents can be implemented as simple lightweight programs, using only the necessary system resources. The main advantages of using agents are [4,5]:

- *proactiveness* – agents are proactive, *i.e.*, they take action when necessary.
- *autonomy* – each agent is autonomous, being able to perform its task on its own with little or no human intervention.
- *social ability* – agents can send and receive messages to the Webmaster, making it easier to follow maintenance activities.

It is important to note that the proposed approach is not a replacement for dynamic pages techniques, such as ASP [3] and JSP [2]. Dynamic pages comprises a widely adopted solution for maintaining the information updated. Although

any piece of information that changes over time can be regarded as dynamic, we can make a distinction between two types of dynamic information: (1) pieces of information which result from a computation, often requiring parameters given by users or coming from other source. (2) pieces of information that only have their values changed periodically. Once instantiated, this sort of information can be presented in static Web pages, which are simpler to be served to users and easier to be found by search engines.

Our approach can co-exist with dynamic pages in a Web site, since the maintenance agents are designed only to maintain static Web pages. It is an alternative for maintaining static pages which contains dynamic pieces of information of type (2) as explained earlier. Note that we have not yet addressed other dynamic aspects that can appear in the navigation structure and presentation of a Web site application. Another important feature of our approach is the ability to update content and visualisation separately. That allows, for instance, changing completely the look-and-feel of a Web page without affecting the content specification or its data source. As benefits, our approach keeps Web pages automatically up-to-date, speeding up the maintenance process. Since it requires fewer technical personnel (Webmaster), it also helps to reduce the overall costs of maintaining a Web site.

3 An Agent-Based Architecture for Web Site Maintenance

In this section we describe the components of our architecture, their details, how they relate to each other and how we implemented them. It is worth pointing out that the architecture herewith described could have been implemented rather differently, using distinct communication infrastructures, different programming languages and even different notations to specify our agents with. In Fig. 1 we show a diagrammatic account of our architecture. The Web Master is shown on the right-hand side interacting with the Web pages (white arrow): he/she annotates the HTML files with specifications of agents to be started up by the Scanner Agent, shown as a grey circle. The Scanner Agent is responsible for going through a given directory where the HTML files are stored and scans these for the annotations specifying agents. For each annotation found in the HTML file, a corresponding agent (black circles) is started up – the complete set of agents obtained at the end of the scanning process is called the Team of Agents looking after the Web site, updating the same Web pages that gave rise to them (vertical arrows in the diagram). The Web Master and the Team of Agents communicate via message-passing (black two-way arrow in the diagram). We explain below each component of the architecture.

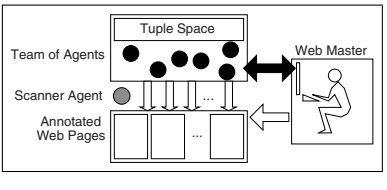

Fig. 1. Agent-Based Architecture

3.1 Annotated Web Pages

The Webmaster adds annotations to individual Web pages, that is, to particular HTML [6] files. Since these annotations become part of an HTML file which Web browsers need to display, they must be *innocuous*, that is, they should not alter the rendering of the HTML by any browser. We achieve this by employing the HTML comment tag "`<!-- ... -->`" to wrap our annotations.

The actual annotations that go within the HTML comment tags are a special-purpose XML [7] construct of the form

`<agent info="`*`InfoId`*`" type="`*`AgType`*`" param="`*`AgParams`*`"></agent>`

where *`InfoId`* is a label to uniquely identify the piece of information within the Web site, *`AgType`* is the type of agent to be started and *`AgParams`* is an optional attribute with any parameters which ought to be passed on to the agent being started up.

The information identification *`InfoId`* allows *content agents* (explained below) to refer to the particular portion of the page they should look after. A single agent, the *publisher agent* explained below, is responsible for updating the annotations of a Web page. The scanner agent assigns to each page with at least one annotation a publisher agent; the annotations themselves cause a number of content agents to be started up – these should look after particular pieces of information on the page. Whenever there are changes to be carried out on the HTML file, they are done via the publisher agent. This arrangement solves any issues arising from concurrent updates of a file and preserves the separation of contents and presentation matters. We exploit a fully distributed coordination mechanism, explained below, by means of which content agents independently offer their updates which are published by the publisher agent. The information identification *`InfoId`* labels a piece of information within a Web site, allowing us to take advantage of one agent's effort for more than one page. As an agent is started up for an annotation to look after a piece of information, we might need the same information elsewhere within the Web site. If the same information appears elsewhere in the site then the Webmaster should annotate it with the same label – the agent looking after the information will interact with the publisher agent of the pages where the information appears and that have been annotated. If, however, the Webmaster accidentally uses a new label for a piece of information already associated with an agent, a new agent will be started up and will look after the information. This will not affect the Web site as the agents will independently look after the same information, but there will duplication of effort.

```
<html><head><title>Weather Forecast...</title></head>
<body>
...
Current Temperature
<!-- <agent info="temp" type="weatherAg"
          param="[every,15,min]"> -->
30 <!-- </agent> --> &deg;C
</body></html>
```

Fig. 2. Sample Annotation for Web Pages

Our annotations also work as delimiters for the HTML portion that the agents should be looking after, that is, the part of the page they are responsible for monitoring and/or updating. In order to tell agents where the portion

they should be looking after ends, we split the "`<agent ...>`" and "`</agent>`" tags, (using HTML comments around them) and enclosing the portion of HTML within the two. We show in Fig. 2 a complete example of an annotation. In it, a specific item of information of an HTML file is enclosed within tags `<agent...>` and `</agent>`. We have associated the agent `weatherAg` with this portion of the HTML file which will be responsible for updating the information every 15 minutes. The actual kinds of agents and their parameters are extensible. Webmasters may create annotations and associate them to special purpose agents which they also develop. We have explored a class of such agents which we explain below.

The Scanner Agent – The scanning process is itself carried out by an agent. This agent is constantly running, checking for new annotations in the files or changes to existing annotations. When a new annotation is found the scanner agent starts up a corresponding agent which will be responsible for that annotation in the HTML file. If there is already an agent responsible for that piece of information, the scanner agent will skip the annotation. Changes to existing annotations (type of the agent or parameters) will cause the previously started agent to be killed and a new agent to be started up instead. The scanner agent parses a hierarchy of HTML files, starting up the team of agents that were specified by the Webmaster to look after the Web site. The Webmaster will include an annotation everywhere in the Web pages where there is a need for monitoring and updating of information and/or formatting. A special agent, the *publisher agent* is started up for each page with at least one annotation. This agent is responsible for collecting the updates on the pieces of information within its page and update them in the HTML file. The scanner agent does not abort when it finds ill-constructed annotations, but skips over them and tells the Webmaster about them. The scanning process does not check for the correctness of the HTML contents, simply concentrating on the annotations and their associated effects (*i.e.*, start up of agents).

Publisher Agents – The scanner agent starts up a *publisher agent* for each HTML file with at least one annotation. This publisher agent is responsible for collecting the information from other agents looking after particular portions of the file and updating it. An alternative to our approach would be to have one agent looking after all annotated pieces of information of a page as well as updating the HTML file. However, if the annotated pieces of information have different frequencies for updates, then this one single agent would need to assess the time elapsed for each previous update in order to perform the next update. Although this is not an impossible task, such an agent will be unnecessarily more complex.

Content Agents – For each piece of information annotated, a corresponding *content agent* is created. This agent's task is to access periodically the data source as defined by the task frequency checking for any update in the information content. Having a specific agent for each piece of information isolates the details of the access to each data source, supporting multiple data sources and facilitating changes in the data source of any information.

Hence, content agents manage and access data only and publisher agents manage the Web pages visualisation (HTML files).

3.2 A Team of (Logic-Based) Agents

Each annotation may give rise to one or more agents, if the piece of information has not already got an agent started up. All our agents are self-contained processes with which Webmasters can communicate via message passing. The type of agent specified in the annotation informs the scanner agent which contents (functionalities) the agent ought to have. Each type is associated with a file containing the source code the agent will use once started up. The parameters in the annotation are passed to the agent which will use its source code with them.

In principle, any programming language can be used to represent the code. We have employed a simple executable temporal logic called TeLA (Temporal Logic of Assertions) [8] that confers a clean design on our agents and makes it possible to formally verify them with temporal theorem provers [9] as well as with model-checking tools such as SPIN [10] and LTSA [11]. The temporal dimension is also required to cope with the issues of frequency. We have used SICStus Prolog [12] to implement all our agents and infrastructure.

Our simple executable temporal logic has only one operator, the next time operator $\bigcirc$, and our formulae have a very restricted syntax, of the form *Present Conditions* $\Rightarrow \bigcirc$*Assertions*, the meaning of which is, if it can prove *PresentConditions*, a conjunction of non-temporal (ordinary) literals, at the current state of affairs (current state of the system), then *Assertions*, a conjunction/disjunction of (possibly temporal) literals will be made to hold in the next state. This simple temporal logic is a practical compromise between the expressiveness of full first-order temporal logic [13] and an efficient means to compute models for its formulae. The creation of such models guides the execution of temporal logic programs. We show in Fig. 3 a simple agent in TeLA. The execution changes between two states, *wait* and *process*(*Msg*), where *Msg* is a variable (we adopt Prolog [14] conventions) to be instantiated to a message. Our simple agent switches between these two states: it stays in the *wait* state until a message *Msg* arrives (predicate *check* succeeds if a message is available for the agent). When a message arrives it moves to state *process*(*Msg*) (first axiom) – a state in which the agent will handle the message it just got. The second axiom allows the agent to reply to the message it has received and the agent goes back to the *wait* state.

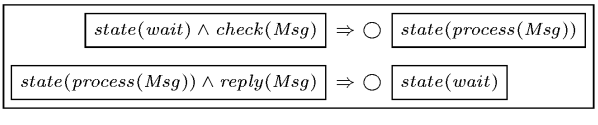

Fig. 3. Sample TeLA Agent

We have enclosed the left and hand sides of the formulae in boxes to improve visualisation. The formulae show an agent that keeps an account of its current state, one of the set $\{wait, process(Msg)\}$, and the conditions that ought to hold in order for the states to change. Actions can be conveniently encoded as

conditions: when a predicate like *check*/1 is attempted to be proved, it causes the execution of a routine to check if a message is available for the agent. A set of temporal formulae such as those above is put to use by means of a proof procedure that tries to build a model that satisfies all the formulae. The formulae depict the program and the proof procedure its interpreter. When agents are started up their corresponding program is the union of temporal formulae and the proof procedure.

The proof procedure works by checking if the atomic formulae on the left-hand side hold true. If they do, then the right-hand side is used to build a model for the next state of the computation. The proof procedure builds a model for the next state in which *all* formulae hold. In order to prove the left-hand side, the proof procedure checks the model for the current state for all those atomic formula that hold. However, the proof procedure also keeps a conventional (atemporal) logic program in which auxiliary predicates are defined – in our example above, the *check* and *reply* predicates are proved by means of one such program. The separation between temporal and atemporal aspects provides neater programs.

3.3 Communication Among/with Agents

Our agents are started up and run as independent Prolog processes which exchange messages by means of the Linda process communication mechanism available in SICStus Prolog [12]. Linda [15] is a concept for process communication, consisting of an area, the *tuple space* (shown in Fig. 1) where data from a number of different processes can be stored and a few basic operations (write, read and delete) can be performed on this area. The Linda concept has proven very successful due to its simplicity and flexibility, being incorporated into a number of programming languages, including Java [16]. SICStus Prolog incorporates a Linda library and offers a repertoire of built-in predicates with which one can write programs that run independently and exchange data. The messages our agents exchange via the tuple space are variants of the FIPA-ACL [17] standard adapted to Prolog's syntax. The information within messages are XML constructs [7] transferred as strings. This standardisation allows for changes in our infrastructure: for the sake of simplicity, we have used Prolog to implement all components of our architecture. However, we could move to a more standard and popular multi-agent platform like JADE [18] without much reimplementation.

Webmasters can communicate with agents, to find out about their status and follow their activities. A simple interface allows communication between the Webmaster and the team of agents. When agents encounter problems they can send messages to the Webmaster alerting to the difficulties they meet and whether they require intervention. Ideally, such interactions should be kept to a minimum, conferring as much autonomy to the agents as possible when they are being designed.

4 Working Example

In this Section we describe three maintenance agents as an example of the use of our approach for automated Web site maintenance. In this example there is one publisher agent and two content agents. For simplicity, it is assumed that all Web site content is kept in a database and all agents have access to this database. Let us consider a Web site for weather forecast having the following information for each location: day of the week, minimum temperature, maximum temperature and current temperature. This database can be represented by the Prolog [14] facts `weather(`*`Location`*`,`*`Day`*`,`*`MinTemp`*`,`*`MaxTemp`*`)` and `current_temp(`*`Location`*`,`*`CurrentTemp`*`)`. The information content in this database is constantly updated from multiple sources. The agents' task is to keep this information updated on a page for a chosen location. Figure 4 illustrates a Web page of this sort. Given that this sort of information is volatile, the content agents need to check the database periodically – for example, every hour the database is checked for a new current temperature. The forecast for maximum and minimum temperatures for the next 5 days does not change in the same frequency, for that reason there is another content agent responsible for keeping track of that particular information.

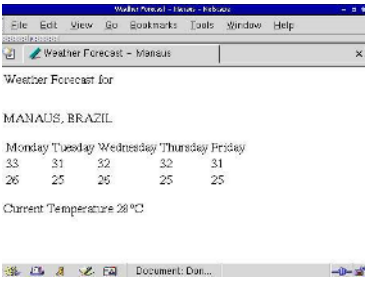

Fig. 4. Weather Forecast Page

```
<html><head><title>Weather Forecast...</title></head>
<body>
...
<!-- <agent info="days_temp" type="weather_ag1"
          param="[every,12,hour]"> -->
<table><tr><td>Monday</td><td>Tuesday</td>...
<tr><td>33</td><td>31</td>... </table> <!-- </agent> -->
<br> Current Temperature
<!-- <agent info="curr_temp" type="weather_ag2"
          param="[every,1,hour]"> -->
30 <!-- </agent> --> &deg;C ...
```

Fig. 5. Annotation for Web Page Maintenance

In order to specify the content agents, the portion in the Web page code where the information about current temperature and the 5-day forecast appears must be annotated as presented in Fig. 5. These are identified, respectively, as agents `weather_ag1` and `weather_ ag2`. Note that a publisher agent is automatically created for every page with an annotation. In this example we will identify this agent as `pub_ag`.

A typical content update agent behaviour can be illustrated by a state transition diagram, depicted in Fig. 6. From the initial state (`start`), the agent immediately moves to the `check` state, where it checks the data source for an

information update. At this point there are two possibilities: there is an update to perform and the agent moves to state `update`, or there is no update to be performed and the agent goes to state `sleep`. Yet another possibility is a problem state, caused by unavailability of the data source, for example. In this case the agent should notify the Webmaster and finish its execution accordingly. We omitted this state for simplicity.

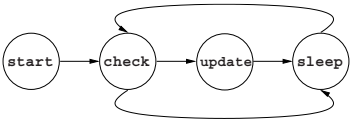

Fig. 6. Content Agent Behaviour

When an agent is in state `update` it puts the new information on the tuple space and notify all publisher agents that are expecting that message – we explain below how this notification is done. After sending the information, the content agent goes to sleep. The details of this coordination among agents is explained in Section 4.1.

State `sleep` corresponds to a time (defined by the frequency of the task) in which the agent remains inactive. When the agent gets active again it moves back to state `check`. Fig. 7 shows the specification of a content maintenance agent using TeLA. Predicate `check_info(I)` encapsulates all necessary steps to access the data source and retrieve the latest data. If there is no new information available, the predicate returns `nil`. Predicate `notify_publishers(I)` makes the updated information available to the publisher agent. Predicate `sleep` makes the agent inactive for a specific duration of time, which is also specified in the Web page annotation.

```
start ⇒ ◯ state(check)
(state(check) ∧ check_info(nil)) ⇒ ◯ state(sleep)
(state(check) ∧ check_info(I) ∧ I ≠ nil) ⇒
                                  ◯ state(update,I)
(state(update,I) ∧ notify_publishers(I)) ⇒
                                    ◯ state(sleep)
(state(sleep) ∧ sleep(1,hour)) ⇒ ◯ state(check)
```

Fig. 7. Content Agent for Current Temperature

The specification of the five-day forecast agent `weather_ag1` is basically the same. The only differences are the piece of information the agent is interested in and the frequency of its update. In this case the agent looks after the information with label `days_temp`, defined in the page annotation.

The publisher agent is, however, rather different as it needs to know about the Web page visualisation style, the actual page layout and the specific place where the information managed by the content agents will be published. It works by placing an information request on the tuple space and it waits for a signal from the content agent responsible for that piece of information. Once the content agent has placed an updated information,

```
start ⇒ ◯ state(check)
(state(check) ∧ in(inform(_,pub_ag,wake_up))) ⇒
                                  ◯ state(publish)
(state(publish) ∧ get_all(Request) ∧
 extract_info(Request,Info) ∧ publish(Info)) ⇒
                                  ◯ state(check)
```

Fig. 8. Publisher Agent

the publisher receives a "wake up" signal and updates the Web page. Fig. 8 shows the publisher agent specification. In state `check` the publisher agent waits for a "wake up" message from a content agent, denoted by predicate `in(inform(_,ag_pub,wake_up))`. In state `publish` the publisher gets all updated information it needs via predicate `get_all(Request)`, then it extracts the value of the pieces of information from the messages using predicate `extract_info/2`, and finally publishes the Web page via `publish(Info)`. Details of the implementation of these predicates and those predicates used in the content agents are given in Section 4.1 below. Note that the page layout is associated with the publisher agent. It is encapsulated by predicate `publish(I)` which inserts the updated piece of information `I` in its right place as defined in the annotation.

Changing the style is also possible, via the publisher agent. This is possible because of an important feature of our approach: keeping content and visualisation management separate. We can define individual styles for pieces of information as a set of predicates where, given a piece of information, it produces a corresponding publishable piece of code in a target mark-up language, such as HTML. For example, `itemize(L)` where `L` is a list of items with the form [$\texttt{item}_1$,$\texttt{item}_2$, ..., $\texttt{item}_n$] results in the HTML code $\texttt{item}_1$
$\texttt{item}_2$... $\texttt{item}_n$. Similarly, other style predicates are defined as `table(LL)`, `paragraph(P)`, `enumerate(L)`, `bigText(P)`, and so on, each resulting in a piece of code which is placed in a page template. As a result the visualisation style of a particular data is changed. This allows the publisher agent to change a piece of information presentation style without affecting its content and completely independent of content maintenance agents and data sources. This also reinforces the idea of separation between information content and visualisation details.

Another use of the publisher agent is to re-publish a corrupted or deleted page, as the agent includes the layout and all static information of a Web page. The content maintenance agents provide the dynamic information of a page and once with all this information a new page can be produced.

The agents presented in this example keep the Web page presented in Fig. 4 automatically updated. Whenever the current temperature or the 5-day weather forecast changes in the database the content agents capture the new values and send them to the publisher agent, which in turn updates the HTML file.

4.1 Agent Coordination

The maintenance agents are coordinated in a particular way, in order to provide independence between content and publisher agents, allowing the same piece of information to be used by more than one publisher agent and to minimise overhead in message exchanging.

A major concept used in our agent architecture is the tuple space. It works as a notice board where any agent can post and retrieve messages. Our coordination mechanism is defined by this order of events:

1. the publisher agents put a request for updated information on the tuple space;
2. the content agents, when they have new updates, look up the tuple space for requests for their pieces of information – each piece of information is identified by the label provided by the Webmaster;
3. the content agents add their new information to the requests and put a wake-up notice on the tuple space to alert the publisher agents there is something for them.

The request message format is `request(Sender,Receiver,Info):Flag`. In this notation, `Sender` and `Receiver` are agent identifications. `Info` has the form `info(Label,Data)`, where `Label` is a unique identifier of a piece of information, as defined in the page annotation and `Data` is its corresponding value. The information label (originated in the Web page annotation) identifies a piece of information and is used by publisher agents to find the right place of the data on its Web page. `Flag` is used as an indication whether the information value `Data` has been updated by a content agent. Our convention is to set `Flag` to value 0 indicating that `Data` has not been updated yet and zero otherwise.

The second type of message is only a signal used to wake up publisher agents with the form `inform(Sender, Receiver,Signal)`. This message is necessary to avoid unnecessary repeated verifications of the tuple space by publisher agents, checking for updated information. Fig. 9 illustrates the given example showing how agents communicate via the tuple space. The top half of the diagram illustrates the `pub_ag` writing on the tuple space the requests for the two pieces of information it needs in its Web page – the entries have "_" (anonymous variables) in the fields whose value is to be supplied by a content agent.

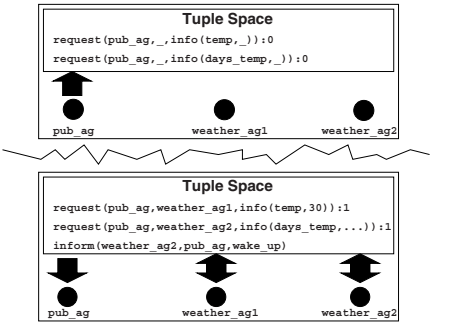

Fig. 9. Coordination via Tuple Space

The bottom half of the diagram of Fig. 9 depicts the content agents `weather_ag1` and `weather_ag2` independently accessing the tuple space and altering the requests with their information. The first agent to supply information to the publisher agent `pub_ag` also inserts the wake-up notice. In our example, `weather_ag2` does this.

The agent implementation in TeLA presented in Figs. 7 and 8 makes use of auxiliary (atemporal) predicates which actually perform the maintenance tasks, such as accessing a database, checking messages on the tuple space and creating the HTML files. It is important to point out that these predicates also have the important role of hiding implementation details, keeping the agent specification clean and easier to read. It allows, for instance, changing the access to data sources, without changing the main specification in TeLA. We describe the im-

plementation details of these predicates below. The content agents also include three such predicates:
1. `check_info`/1: succeeds if the data item has changed in the data source, returning its value, or succeeds if the data item has not changed, returning `nil`. The actual implementation might involve details for connecting and accessing a database, via SQL queries, for example.
2. `sleep`/1: succeeds if the execution of an agent is suspended for the duration specified by the arguments (number and time unit).
3. `notify_publishers`/1: succeeds if there are requests from publisher agents for the piece of information in the tuple space and sends the updated value of the information and a signal to wake up the publisher agents who have requested that information. The code in Fig. 10 illustrates this predicate implementation for the agent that updates the current temperature. Predicate `bagof_rd_noblock(A,B,C)` builds list `C` with all terms `A` that match term `B` in the tuple space. In the program in Fig. 10, `AllRequest` is a list of publisher agent ids, who have asked for information `curr_temp`. Publisher agents also require special predicates described in [19].

```
notify_publishers(I) :-
   bagof_rd_noblock(P,
       request_info(P,_,info(curr_temp,_)):0,
       AllRequest),
   my_id(Me),
   send_info(Me,AllRequest,I).

send_info(_,[],_).
send_info(Me,[P|RestP],I) :-
   out(request_info(P,Me,info(curr_temp,I)):1),
   out(inform(Me,P,wake_up),
   send_info(Me,RestP,I).
```

Fig. 10. Content Agent Implementation

5 Related Work

We can speculate that batch files and daemons have been employed in an ad-hoc fashion by Webmasters to automate repetitive tasks, but these have not led to a systematic approach to Web site maintenance. One major drawback with batch files and daemons is their lack of responsiveness: in order to find out about them (*i.e.*, whether they are alive and running), the Webmaster must look up their process identifications and their status or log files – scaling up clearly becomes a problem.

Similar annotations, the associated scanning process and the bootstrapping of agents were originally described in [20], using Java [21] as a programming language and JADE [18] as a communication platform. Each agent incorporated a particular functionality: a particular piece of information from *another* Web site was specified for the agent to monitor and periodically fetch, using it to update one's Web site [22]. These two references inspired our work herewith described and, apart from them, we have not been able to find any work on using multi-agent systems to look after Web sites.

A number of methods and techniques for Web site development have been proposed such as OOHDM [23], Araneus [24], Strudel [25], WebML [26], UWE [27], and the logic-based approaches of [28] and OMSwe [29], among others. The use of a formal approach for Web site modelling and development facilitates

maintenance tasks as they usually provide a high-level view of the application and separation between content, navigation structure and visualisation. This allows, for instance, updating page templates or colour schemes without affecting the page content or links. Although these works have addressed the problem of Web site application development in the large, where maintenance is one of the issues involved, our work proposes an approach to static Web page maintenance which can also be adapted in order to be employed as part of a complete Web site application development method, as those mentioned above.

Specific automated maintenance is not addressed by most methods, although some claim support for it or offer some degree of automation. Given that there are certain maintenance tasks which are well defined and repetitive, it is possible to automate them in order to avoid human intervention, saving maintenance time and keeping the Web site continuously updated. With this respect, two work are closer to our proposed approach.

Sindoni [30] proposes an approach to enforce consistency between page content and database state. A specific algebra is proposed for defining views and views updates, which uses the Araneus Data Model as the reference model. Views updates are used by an algorithm that automatically updates the Web site from a set of changes on the application database. Maintenance is performed by removing or generating the appropriate sets of pages.

OntoWebber [31] is an ontology-based approach designed for the generation of data-intensive Web sites, in particular Web portals. It addresses the problem of multiple and heterogeneous sources of data by defining integration and articulation layers responsible for resolving syntactic and semantic differences between the different data sources. The composition layer includes a specification of a Web site view based on the Web site modelling ontologies. Ontologies are described using DAML+OIL [32], a semantic mark-up language for Web resources. A generation layer process queries the site specification to produce the Web pages. This approach offers some degree of automation by means of rules defined as triggers. These rules update the source data, meta-data, and site view specifications according to the fired triggers. However it is restricted to content maintenance.

6 Conclusions and Discussion

In this paper we introduce an agent-based approach to the maintenance of Web sites. Our approach caters for static Web pages whose contents require monitoring and updating. We offer a notation that Webmasters can use to annotate particular points of a Web page (an HTML file) in order to specify that an agent should monitor and update that portion of the page. The Webmaster can add as many annotations as required in one Web page. Our architecture associates, via a scanning process (carried out by an agent itself) an agent to each annotation. This agent is started up and autonomously looks after its prescribed piece of information. Any modifications to a Web page are centrally carried out by a

publisher agent with whom all agents communicate – each annotated Web page has a publisher agent, started up by the scanning process.

It is important to decouple our proposed architecture and its implementation, as explained here. We do not suggest that ours is the only or the best means to implement the proposed architecture: our implementation should be considered as a proof-of-concept prototype used to exploit and refine our ideas. However, our implementation captures all basic functionalities of our architecture and is evidence that our architecture is feasible. Some features of our proposal worth pointing out are:

- *Scalability* – as many agents as required can be started up, provided that there are available computational resources. In our experiments, we used up to 250 agents in one single Pentium III PC (1GHz) running under Linux. However, the scanner agent can be programmed to start up agents in different machines. The SICStus Prolog tuple space employs a unique *host:port* addressing which allows agent anywhere in the Internet to access it.
- *Ease of use* – agents are now responsible for tasks that relied on humans or off-line daemons and batch files. Rather than keeping a record on the status of daemons and batch files or the actions of humans, the manager can now communicate with thousands of agents collectively or individually.
- *Robustness* – because the task of monitoring and updating the pages of the Web site is divided among independent agents, if individual components fail it is still possible to achieve some degree of overall functionality. Additionally, we can enable agents to perform failsafe operations when they encounter exceptional circumstances. For instance, if the data source an agent is using suddenly becomes unavailable, the agent could provide a "`Not Available`" default value to update the information on the Web page. The agent could also send a message to the Webmaster and wait for further instructions.
- *Backwards compatibility* – any existing Web site can incorporate our architecture, as the annotations are naturally accommodated within HTML files.
- *Extensibility* – the class of available agents and their functionalities can be gradually extended, and new annotations specifying them can be added at will. Web pages can be gradually annotated as the Webmaster becomes used to the new managerial style of administering a team of software agents.

Hyperlinks within Web sites may need constant monitoring as the objects they point at may move or disappear – we envisage employing software agents for constantly scanning the whole collection of HTML files, checking for broken references. These agents notify the Webmaster and isolate the offending reference, wrapping it as a comment. We are currently working on how these agents can be incorporated into our architecture.

Work is under way to integrate our proposal for agent-based maintenance with a high-level specification of Web sites, as described in [28,33]. This approach also uses logic to specify a Web application thus facilitating the desired integration. We notice that in a high-level specification of a Web site application the annotations for associating pieces of information to agents do not need to be made in the HTML files directly; rather they should become part of the site

specification. This opens new possibilities to improve both approaches to Web site synthesis and maintenance.

References

1. Wang, P.S., Katila, S.S.: An Introduction to Web Design and Programming. Brooks/Cole-Thomson, U.S.A. (2004)
2. Hall, M.: Core Servlets & JavaServer Pages. Addison-Wesley (2000)
3. Weissinger, A.: ASP in a Nutshell, 2nd Edition. O'Reilly (2000)
4. Franklin, A. and Graesser, A.: Is it an Agent, or just a Program? In: LNAI. Volume 1193. Springer, Berlin (1997)
5. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley & Sons Ltd., England, U.K. (2002)
6. Musciano, C., Kennedy, B.: HTML & XHTML: The Definitive Guide. 4th edn. O'Reilly, USA (2000)
7. Harold, E.R.: XML: Extensible Markup Language. IDG Books, U.S.A (1998)
8. Cavalcanti, J., Vasconcelos, W.: A Logic-Based Approach for Automatic Synthesis and Maintenance of Web Sites. In: Procs. of the 14th Int'l Conf. on Soft. Eng. & Knowl. Eng.(SEKE'02), ACM Press (2002)
9. Manna, Z., Pnuelli, A.: How to Cook a Temporal Proof System for your Pet Language. In: Proc. 10th POPL-ACM. (1983) 141–154
10. Holzmann, G.J.: The SPIN Model Checker. IEEE Trans. on Soft. Eng. **23** (1997)
11. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley & Sons, England, UK (1999)
12. SICS: SICStus Prolog User's Manual. Swedish Institute of Computer Science, available at `http://www.sics.se/ sicstus` (2000)
13. Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R.: MetateM: an Imperative Approach to Temporal Logic Programming. Formal Aspects of Computing **7** (1995) 111–154
14. Apt, K.R.: From Logic Programming to Prolog. Prentice-Hall, U.K. (1997)
15. Carriero, N., Gelernter, D.: Linda in Context. Comm. of the ACM **32** (1989)
16. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces: Principles, Patterns and Practice. Addison-Wesley, U.S.A. (1999)
17. FIPA: The Foundation for Physical Agents. `http://www.fipa.org` (2002)
18. Bellifemine, F., Poggi, A., Rimassa, G.: JADE: A FIPA-compliant Agent Framework. Technical report, CSELT S.p.A (1999) `http://sharon.cselt.it/projects/jade/`.
19. Vasconcelos, W.W., Cavalcanti, J.: Agent-Based Web Site Maintenance. Technical Report 0401, Dept. of Comp. Science, Univ. of Aberdeen, U.K. (2004) Available at `http://www.csd.abdn.ac.uk/~wvasconc/pubs/techreportAUCS0401.pdf`.
20. Clarkson, D.: Agents for Web Management: An Architecture and its Implementation. MSc Report, MTP Programme in E-Commerce, Dept. of Computing Sci., Univ. of Aberdeen, U.K. (2003)
21. Spell, B.: Professional Java Programming. Wrox Press Inc (2000)
22. Shand, A.: Minion: An Approach to Automated Website Information Updating. MSc Report, MTP Programme in E-Commerce, Dept. of Computing Sci., Univ. of Aberdeen, U.K. (2003)
23. Schwabe, D., Rossi, G.: The Object-oriented Hypermedia Design Model. Comm. of the ACM **38** (1995) 45–46

24. Atzeni, P., Mecca, G., Merialdo, P.: Design and Maintenance of Data-Intensive Web Sites. In: Procs. of the Int'l Conf. on Extending Database Technology (EDBT), Valencia, Spain (1998)
25. Fernández, M., Florescu, D., Kang, J., Levy, A., Suciu, D.: Catching the Boat with Strudel: Experience with a A Web-site Management System. SIGMOD Record **27** (1998)
26. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a Modeling Language for Designing Web Sites. In: Proceedings of the WWW9 conference, Amsterdam, the Netherlands (2000)
27. Hennicker, R., Koch, N.: A UML-based Methodology for Hypermedia Design. In: Procs. Unified Modeling Language Conference (UML'2000). Volume 1939 of LNCS. Springer-Verlag (2000) 410–424
28. Cavalcanti, J., Robertson, D.: Synthesis of Web Sites from High Level Descriptions. Volume 2016 of LNCS. Springer, Germany (2001)
29. Norrie, M., Palinginis, A.: From State to Structure: an XML Web Publishing Framework. In: 15th Conf. on Advanced Information Systems Engineering (CAiSE'03), Klagenfurt/Velden, Austria (2003)
30. Sindoni, G.: Incremental Maintenance of Hypertext Views. In: Procs. of the Int'l Workshop on the Web and Databases, Valencia, Spain (1998) 98–117
31. Jin, Y., Decker, S., Wiederhold, G.: OntoWebber: Model-driven ontology-based Web site management. In: Procs. of the 1st Int'l Semantic Web working symposium (SWWS'01), Stanford, CA, USA (2001)
32. W3C: DAML+OIL Reference Description (2001) `http://www.w3.org/TR/daml+oil-reference`.
33. Cavalcanti, J., Robertson, D.: Web Site Synthesis based on Computational Logic. Knowledge and Information Systems Journal (KAIS) **5** (2003) 263–287