Exploiting Agent-Oriented Programming for Building Advanced Web 2.0 Applications

Mattia Minotti
University of Bologna
Cesena, Italy
Email: mattia.minotti@studio.unibo.it

Andrea Santi
DEIS, University of Bologna
Cesena, Italy
Email: a.santi@unibo.it

Alessandro Ricci
DEIS, University of Bologna
Cesena, Italy
Email: a.ricci@unibo.it

***Abstract*—We believe that agent-oriented programming languages and multi-agent programming technologies provide an effective level of abstraction for tackling the design and programming of mainstream software applications, besides being techniques effective for dealing with (Distributed) Artificial Intelligence problems. In this paper we support this claim in practice by discussing the use of a platform integrating two main agent programming technologies – *Jason* agent programming language and CArtAgO environment programming framework – to the development of Web 2.0 applications. Following the cloud computing perspective, these kinds of applications will more and more replace desktop applications, exploiting the Web infrastructure as a common distributed operating system, raising however challenges that are not effectively tackled – we argue – by mainstream programming paradigms, such as the object-oriented one.**

I. Introduction

The value of Agent-Oriented Programming (AOP) [21] – including Multi-Agent Programming (MAP) – is often remarked and evaluated in the context of Artificial Intelligence (AI) and Distributed AI problems. This is evident, for instance, by considering existing agent programming languages (see [6], [7] for comprehensive surveys) – whose features are typically demonstrated by considering AI toy problems such as block worlds and alike. Besides this view, we argue that the level of abstraction introduced by AOP is effective for organizing and programming software applications in general, starting from those programs that involve aspects related to reactivity, asynchronous interactions, concurrency, up to those involving different degrees of autonomy and intelligence. In that context, an important example is given by Web 2.0 applications, which share more and more features with desktop applications, combining their better user experience with all the benefits provided by the Web, such as distribution, openness and accessibility. Applications of this kind are at the core of the cloud computing vision.

In this paper we show this idea in practice by describing a platform for developing Web 2.0 applications using agent programming technologies, in particular *Jason* for programming agents and CArtAgO for programming the environments where agents work. We refer to the integrated use of *Jason* and CArtAgO as JaCa and its application for building Web 2.0 application as JaCa-Web. Besides describing the platform, our aim here is to discuss the key points that make JaCa and – more generally – agent-oriented programming a suitable paradigm for tackling main complexities of software applications, advanced Web applications in this case, that – we argue – are not properly addressed by mainstream programming languages, such as object-oriented ones. In that, this work extends a previous one [12] where we adopted a Java-based framework called simpA [20] to this end, replaced in this paper *Jason* so as to exploit the features provided by the Belief-Desire-Intention (BDI) architecture.

The remainder of the paper is organised as follows. First, we provide a brief overview of JaCa (Section II) programming model and platform. Then, we discuss the use of JaCa for developing Web 2.0 applications (Section III), remarking the advantages compared to existing state-of-the art approaches. To evaluate the approach, we describe the design and implementation of a case study (Section IV), and we conclude the paper by discussing related and future work (Section V).

II. Agent-Oriented Programming for Mainstream Application Development – The JaCa Approach

An application in JaCa is designed and programmed as a set of agents which work and cooperate inside a common environment. Programming the application means then programming the agents on the one side, encapsulating the logic of control of the tasks that must be executed, and the environment on the other side, as first-class abstraction providing the actions and functionalities exploited by the agents to do their tasks. More specifically, in JaCa *Jason* [5] is adopted as programming language to implement and execute the agents and CArtAgO [18] as the framework to program and execute the computational environments where agents are situated.

Being a concrete implementation of an extended version of AgentSpeak(L) [16], *Jason* adopts a BDI (Belief-Desire-Intention)-based computational model and architecture to define the structure and behaviour of individual agents. In that, agents are implemented as reactive planning systems: they run continuously, reacting to events (e.g., perceived changes in the environment) by executing plans given by the programmer. Plans are courses of actions that agents commit to execute so as to achieve their goals. The pro-active behaviour of agents is possible through the notion of goals (desired states of the world) that are also part of the language in which plans are written. Besides interacting with the environment, Jason agents can communicate by means of speech acts.

On the environment side, CArtAgO – following the A&A meta-model [13], [19] – adopts the notion of *artifact* as first-class abstraction to define the structure and behaviour of such computational environments and the notion of *workspace* as a logical container of agents and artifacts. Artifacts explicitly represent the resources and tools that agents may dynamically instantiate, share and use, encapsulating functionalities designed by the environment programmer. In order to be used by the agents, each artifact provides of a usage interface composed by a set of *operations* and *observable properties*. Operations correspond to the actions that the artifact makes it available to agents to interact with such a piece of the environment; observable properties define the observable state of the artifact, which is represented by a set of information items whose value (and value change) can be perceived by agents as percepts. Besides observable properties, the execution of operations can generate signals perceivable by agents as percepts, too. As a principle of composability, artifacts can be assembled together by a link mechanism, which allows for an artifact to execute operations over another artifact. CArtAgO provides a Java-based API to program the types of artifacts that can be instantiated and used by agents at runtime, and then an object-oriented data-model for defining the data structures in actions, observable properties and events.

Finally, the notion of workspace is used to define the topology of complex environments, that can be organised as multiple sub-environments, possibly distributed over the network. By default, each workspace contains a predefined set of artifact created at boot time, providing basic actions to manage the set of artifacts in the workspace – for instance, to create, lookup, link together artifacts – to join multiple workspaces, to print message on the console, and so on.

JaCa integrates Jason and CArtAgO so as to make the use of artifact-based environments by Jason agents seamless. To this purpose, first, the overall set of external actions that a Jason agent can perform is determined by the overall set of artifacts that are actually available in the workspaces where the agent is working. So, the action repertoire is dynamic and can be changed by agents themselves by creating, disposing artifacts. Then, the overall set of percepts that a Jason agent can observe is given by the observable properties and observable events of the artifacts available in the workspace at runtime. Actually an agent can explicitly select which artifacts to observe, by means of a specific action called *focus*. By observing an artifact, artifacts' observable properties are directly mapped into beliefs in the belief-base, updated automatically each time the observable property changes its value. So a Jason agent can specify plans reacting to changes to beliefs that concern observable properties or can select plan according to the value of beliefs which refer to observable properties. Artifacts' signals instead are not mapped into the belief base, but processed as non persistent percepts possibly triggering plans—like in the case of message receipt events. Finally, the Jason data-model – essentially based on Prolog terms – is extended to manage also (Java) objects, so as to work with data exchanged by performing actions and processing percepts.

A full description of Jason language/platform and CArtAgO framework – and their integration – is out of the scope of this paper: the interested reader can find details in literature [18], [17] and on Jason and CArtAgO open-source web sites[1][2].

III. Programming Web 2.0 Applications with JaCa

In this section, we describe how the features of JaCa can be exploited to program complex Web 2.0 applications, providing benefits over existing approaches. First, we sketch the main complexities related to the design and programming of modern and future web applications; then we describe how these are addressed by JaCa-Web, which is a framework on top of JaCa to develop such a kind of applications.

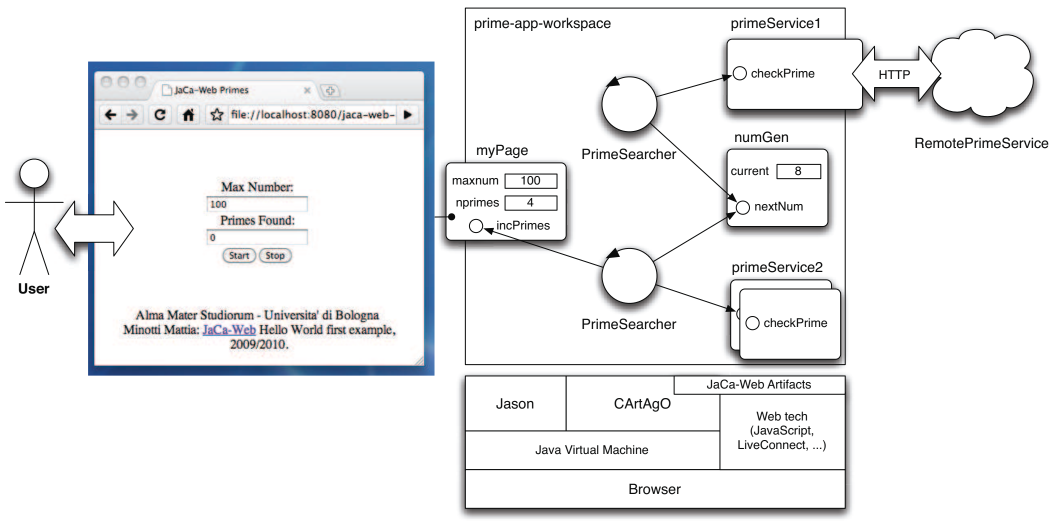
A. Programming Future Web Applications: Complexities

Due to network speed problems overcoming and machine computational power increasing, the client-side of so-called *rich web applications* is constantly evolving in terms of complexity. Web 2.0 applications share more and more features with desktop applications in order to combine their better user experience with all Web benefits, such as distribution, openness and accessibility. One of the most important features of Web 2.0 is a new interaction model between the client user interface of a Web browser and the server-side of the application. Such rich Web applications allow the client to send multiple concurrent requests in an asynchronous way, avoiding complete page reload and keeping the user interface live and responding. Periodic activities within the client-side of the applications can be performed in the same fashion, with clear advantages in terms of perceived performance, efficiency and interactivity.

So the more complex web apps are considered, the more the application logic put on the client side becomes richer, eventually including asynchronous interactions – with the user, with remote services – and possibly also concurrency – due to the concurrent interaction with multiple remote services. This situation is exemplified by cloud computing applications, such as Google doc[3].

The direction of decentralizing responsibilities to the client is evident also by considering the new HTML standard 5.0, which enriches the set of API and features that can be used by the web application on the client side[4]. Among the others, some can have a strong impact on the way an application is designed: it is the case of the Web Worker mechanism[5], which makes it possible to spawn background workers running scripts in parallel to their main page, allowing for thread-like operation with message-passing as coordination mechanism.

[1]http://jason.sourceforge.net
[2]http://cartago.sourceforge.net
[3]http://docs.google.com
[4]http://dev.w3.org/html5/spec/
[5]http://www.whatwg.org/specs/web-workers/current-work/



Alma Mater Studiorum - Universita' di Bologna
Minotti Mattia: JaCa-Web Hello World first example, 2009/2010.

Fig. 1. An abstract overview of a JaCa-Web application, referring in particular to the toy example described in the paper. In evidence: *(Top)* the workspace with the agents (circles) and artifacts (rounded square); among the artifacts, `myPage` and `primeService1` enable and rule the interaction with the external environment sources, namely the human user and the remote HTTP service; *(Bottom)* the layers composing the JaCa-Web platform, which includes – on top of the Java Virtual Machine and browser/web infrastructure – *Jason* and CArtAgO sub-system and then a pre-defined library of artifacts (JaCa-Web artifacts) specifically designed for the Web context.

Another one is cross-document messaging[6], which defines mechanisms for communicating between browsing contexts in HTML documents.

Besides devising enabling mechanisms, a main issue is then how to design and program applications of this kind [11].

A basic and standard way to realise the client side of web app is to embed in the page scripts written in some scripting language – such as JavaScript. Originally such scripts were meant just to perform check on the inputs and to create visual effects. The problem is that scripting languages – like JavaScript – have not been designed for programming in the large, so using them to organize, design, implements complex programs is hard and error-prone.

To address the problems related to scripting languages, higher-level approaches have been proposed, based on frameworks that exploit mainstream object-oriented programming languages. A main example is Google Web Toolkit (GWT)[7], which allows for developing client-side apps with Java. This choice makes it possible to reuse and exploit all the strength of mainstream programming-in-the-large languages that are typically not provided by scripting languages—an example is strong typing. However it does not provide significant improvement for those aspects that are still an issue for OO programming languages, such as concurrency, asynchronous events and interactions, and so on.

We argue then that these aspects can be effectively captured by adopting an agent-oriented level of abstraction and programmed by exploiting agent-oriented technologies such as JaCa: in next section we discuss this point in detail.

[6]http://dev.w3.org/html5/postmsg/
[7]http://code.google.com/webtoolkit/

B. *An Agent-Oriented Programming Approach based on* JaCa

By exploiting JaCa, we directly program the Web 2.0 application as a normal JaCa agent program, composed by a workspace with one or multiple agents working within an artifact-based environment including a set of pre-defined type of artifacts specifically designed for the Web context domain (see Fig. 1). Generally speaking, agents are used to encapsulate the logic of control and execution of the tasks that characterise the Web 2.0 app, while artifacts are used to implement the environment needed for executing the tasks, including those coordination artifacts that can ease the coordination of the agents' work. As soon as the page is downloaded by the browser, the application is launched – creating the workspace, the initial set of agents and artifacts.

Among the pre-defined types of artifact available in the workspace, two main ones are the *Page* artifact and the *HTTPService* artifact. *Page* provides a twofold functionality to agents: (i) to access and change the web page, internally exploiting specific low-level technology to work with the DOM (Document Object Model) object, allowing for dynamically updating its content, structure, and visualisation style; (ii) to make events related to user's actions on the page observable to agents as percepts. An application may either exploit directly *Page* or define its own extension with specific operations and observable properties linked to the specific content of the page. *HTTPService* provides basic functionalities to interact with a remote HTTP service, exploiting and hiding the use of sockets and low-level mechanisms. Analogously to *Page*, this kind of artifact can be used as it is – providing actions to do HTTP requests – or can be extended providing an higher-level application specific usage interface hiding the HTTP level.

To exemplify the description of these elements and of JaCa-

Web programming in the overall, in the following we consider a toy example of Web 2.0 app, in which two agents are used to search for prime numbers up to a maximum value which can specified and dynamically changed by the user through the web page. As soon as an agent finds a new prime number, a field on the the web page reporting the total number of values is updated.

The environment (shown in Fig. 1) includes – besides the artifact representing the page, called here `myPage` – an artifact called `numGen`, functioning as a number generator, shared and used by agents to get the numbers to verify, and two artifacts, `primeService1` and `primeService2`, providing the (same) functionality that is verifying if a number is prime.

`myPage` is an instance of `MyPage` extending the basic *Page* artifact so as to be application specific, by: *(i)* defining an observable property `maxnum` whose value is directly linked to the related input field on the web page; *(ii)* generating `start` and `stop` signals as soon as the page button controls start and stop are pressed; *(ii)* defining an operation `incPrimes` that updates the output field of the page reporting the actual number of prime numbers found.

`numGen` is an instance of the `NumGen` artifact (see Fig. 3), which provides an action `getNextNum` to generate a new number – retrieved as output (i.e. action feedback) parameter.

The two prime number service artifacts provide the same usage interface, composed by a `checkPrime(num: integer)` action, which generates an observable event `is_prime(num: integer)` if the number is found to be prime. One artifact does the computation locally (`LocalPrimeService`); the other one, instead – which is an instance of `RemotePrimeService`, extending the pre-defined *HTTPService* artifact – provides the functionality by interacting with a remote HTTP service.

Fig. 2 shows the source code of one of the two agents. After having set up the tools needed to work, the agent waits to perceive a `start` event generated by the page. Then, it starts working, repeatedly getting a new number to check – by executing a `getNextNum` – until the maximum number is achieved. The agent knows such a maximum value by means of the `maxnum` page observable property—which is mapped onto the related agent belief. The agent checks the number by performing the action `checkPrime` and then reacting to `is_prime(Num: integer)` event, updating the page by performing `incPrimes`. If a `stop` event is perceived – which means that the user pressed the stop button on the Web page – the agent promptly reacts and stops working, dropping its main intention.

A final note about implementation: Java applet technology is used to run the full application stack (including *Jason* and CArtAgO) in the browser, using signed applets so to avoid limitations imposed by the sandbox model. LiveConnect technology[8] is exploited to enable a bi-direction interaction between the applet and the web page resources (DOM, scripts).

[8]https://jdk6.dev.java.net/plugin2/liveconnect/

```
!setup.

+!setup
  <-  focusByName("MyPage");
      makeArtifact("primeService1",
                        "RemotePrimeService");
      makeArtifact("numGen","NumGen").

+start
  <-  focusByName("primeService1");
      focusByName("numGen");
      !!checkPrimes.

+!checkPrimes
  <- nextNum(Num);
     !checkNum(Num).

+!checkNum(Num): maxnum(Max) & Num <= Max
  <- checkPrime(Num);
     !checkPrimes.

+!checkNum(Num) <- maxnum(Max) & Num > Max.

+is_prime(Num) <- incPrimes.

+stop <- .drop_intention(checkPrimes).
```

Fig. 2. *Jason* source code of a prime searcher agent.

```
public class MyPage extends PageArtifact {
   protected void setup() {
    defineObsProperty("maxnum",getMaxValue());
    //Operation for event propagation
    linkEventToOp("start","click","startClicked");
    linkEventToOp("stop","click","stopClicked");
    linkEventToOp("maxnum","change","maxnumChange");
  }
  @OPERATION void incPrimes(){
    Elem el = getElementById("primes_found");
    el.setValue(el.intValue()+1);
  }
  @INTERNAL_OPERATION private void startClicked(){
    signal("start");
  }
  @INTERNAL_OPERATION private void stopClicked(){
    signal("stop");
  }
  @INTERNAL_OPERATION private void maxnumChange(){
    updateObsProperty("maxnum",getMaxValue());
  }
  private int getMaxValue(){
    return getElementById("maxnum").intValue();
  }
}

public class RemotePrimeService extends HTTPService {

 @OPERATION void checkPrime(double n){
    HTTPResponse res =
          doHTTPRequest(serverAddr,"isPrime",n);
    if (res.getElem("is_prime").equals("true")){
      signal("is_prime",n);
    }
  }
}

public class NumGen extends Artifact {
  void init(){ defineObsProperty("current",0); }

  @OPERATION void nextNum(OpFeedbackParam<Integer> res){
    int v = getObsProperty("current").intValue();
    updateObsProperty("current",++v);
    res.set(v);
  }
}
```

Fig. 3. Artifacts' definition in CArtAgO: `MyPage` and `RemotePrimeService` extending respectively `PageArtifact` and `HTTPService` artifact types which are available by default in JaCa-Web workspaces, and `NumGen` to coordinate number generation and sharing.

C. Key points

We have identified three key points that, in our opinion, represent main benefits of adopting agent-oriented programming and, in particular, the JaCa-Web programming model, for developing applications of this kind.

First, agents are first-class abstractions for mapping possibly concurrent tasks identified at the design level, so reducing the gap from design to implementation. The approach allows for choosing the more appropriate concurrent architecture, allocating more tasks to the same kind of agent or defining multiple kind of agents working concurrently. This allows for easily programming Web 2.0 concurrent applications, that are able to exploit parallel hardware on the client side (such as multi-core architectures). In the example, two agents are used to fairly divide the overall job and work concurrently, exploiting the number generator artifact as a coordination tool to share the sequence of numbers. Actually, changing the solution by using a single agent or more than two agents would not require any substantial change in the code.

A second key point provided by the agent control architecture is the capability of defining task-oriented computational behaviours that straightforwardly integrate the management of asynchronous events generated by the environment – such as the input of the user or the responses retrieved from remote services – and the management of workflows of possibly articulated activities, which can be organized and structured in plans and sub-plans. This makes it possible to avoid the typical problems produced by the use of callbacks – that can be referred as *asynchronous spaghetti code* – to manage events within programs that need – at the same time – to have one or multiple threads of control.

In the prime searcher agent shown in the example, for instance, on the one hand we use a plan handling the `checkPrimes` goal to pro-actively search for prime numbers. The plan is structured then into a subgoal `checkNum` to process the number retrieved by interacting with the number generator. Then, the plan executed to handle this subgoal depends on the dynamic condition of the system: if the number to process is greater than the current value of the `maxnum` page observable property (i.e. of its related agent belief), then no checks are done and the goal is achieved; otherwise, the number is checked by exploiting a prime service available in the environment and the a new `checkPrimes` goal is issued to go on exploring the rest of the numbers. The user can dynamically change the value of the maximum number to explore, and this is promptly perceived by the agents which can change then their course of actions accordingly. On the other hand, reactive plans are used to process asynchronous events from the environment, in particular to process incoming results from prime services (line 22) and user input to stop the research (line 24).

Finally, the third aspect concerns the strong separation of concerns which is obtained by exploiting the environment as first class abstraction. *Jason* agents, on the one side, encapsulates solely the logic and control of tasks execution; on the other side, basic low-level mechanisms to interact and exploit the Web infrastructure are wrapped inside artifacts, whose functionalities are seamlessly exploited by agents in terms of actions (operations) and percepts (observable properties and events). Also, application specific artifacts – such as `NumGen` – can be designed to both encapsulate shared data structures useful for agents' work and regulate their access by agents, functioning as a coordination mechanism.

IV. A Case Study

To stress the features of agent-oriented programming and test-drive the capabilities of the JaCa-Web framework, we developed a real-world Web application – with features that go beyond the ones that are typically found in current Web 2.0 app. The application is about searching products and comparing prices from multiple services, a "classic" problem ton the Web.

We imagine the existence of N services that offer product lists with features and prices, codified in some standard machine-readable format. The client-side in the Web application needs to search all services for a product that satisfies a set of user-defined parameters and has a price inferior to a certain user-defined threshold. The client also needs to periodically monitor services so as to search for new offerings of the same product. A new offering satisfying the constraints should be visualised only when its price is more convenient than the currently best price. The client may finish its search and monitoring activities when some user-defined conditions are met—a certain amount of time is elapsed, a product with a price less than a specified threshold is find, or the user interrupts the search with a click on a proper button in the page displayed by the browser. Finally, if such an interruption took place, by pressing another button it must be possible to let the search continue from the point where it was blocked.

Typically applications of this kind are realised by implementing all the features on the server side, without – however – any support for long-term searching and monitoring capabilities. In the following, we describe a solution based on JaCa-Web, in which responsibilities related to the long-term search and comparison are decentralised into the client side of the application, improving then the scalability and quality of service for the users.

A. Application Design

The solution includes two kinds of agents (see Fig. 4): a *UserAssistant* agent – which is responsible of setting up the application environment and manage interaction with the user – and multiple *ProductFinder* agents, which are responsible to periodically interact with remote product services to find the products satisfying the user-defined parameters. To aggregate data retrieved from services and coordinate the activities of the *UserAssistant* and *ProductFinder* we introduce a *ProductDirectory* artifact, while a *MyPage* page artifact and multiple instances of *ProductService* artifacts are used respectively by the *UserAssistant* and *ProductFinder* to interact with the user and with remote product services.

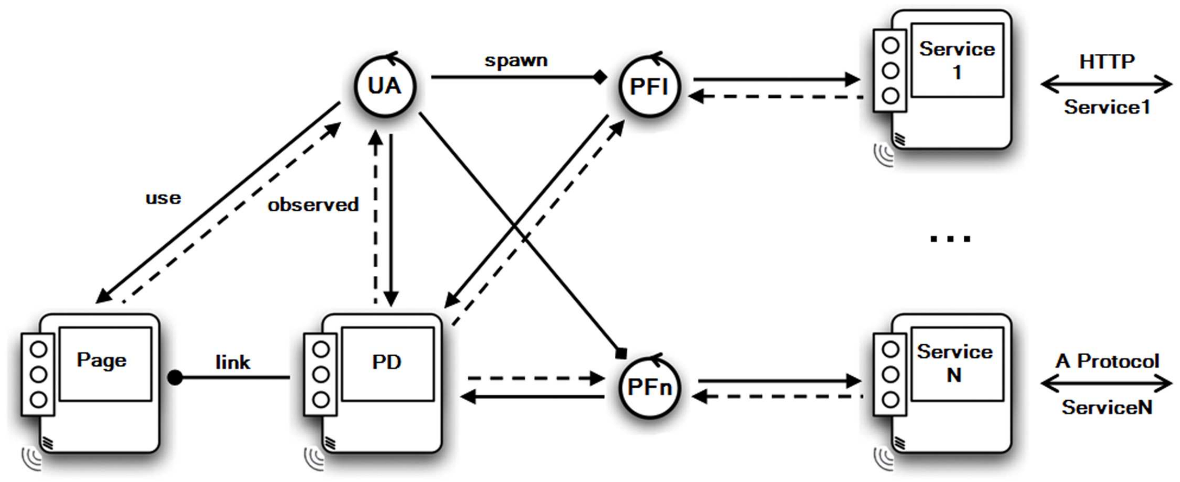


Fig. 4. The architecture of the client-side Web application sample in terms of agent, artifacts, and their interactions. `UA` is the *UserAgent*, `PF`s are the *ProductFinder* agents, `PD` is the *ProductDirectory* artifact and finally `Service`s are the *ProductService* artifacts

More in detail, the *UserAssistant* agent is the first agent booted on the client side, and it setups the application environment by creating the *ProductDirectory* artifact and spawning a number of *ProductFinder* agents, one for each service to monitor. Then, by observing the *MyPage* artifact, the agent monitors user's actions and inputs. In particular, the web page provides controls to start, stop the searching process and to specify and change dynamically the keywords related to the product to search, along with the conditions to possibly terminate the process. Page events are mapped onto `start` and `stop` observable events generated by *MyPage*, while specific observable properties – `keywords` and termination conditions – are used to make it observable the input information specified by the user.

The *UserAssistant* reacts to these observable events and to changes to observable properties, and interacts with *ProductFinder* agents to coordinate the searching process. The interaction is mediated by the *ProductDirectory* artifact, which is used and observed by both the *UserAssistant* and *ProductFinder*s. In particular, this artifact provides a usage interface with operations to: *(i)* dynamically update the state and modality of the searching process – in particular `startSearch` and `stopSearch` to change the value of a `searchState` observable property – useful to coordinate agents' work – and `changeBasePrice`, `changeKeywords` to change the value of the base price and the keywords describing the product, which are stored in a `keyword` observable property; *(ii)* aggregate product information found by *ProductFinder*s – in particular `addProducts`, `removeProducts`, `clearAllProducts` to respectively add and remove a product, and remove all products found so far. Besides `searchState` and `keywords`, the artifact has further observable properties, `bestProduct`, to store and make it observable the best product found so far.

Finally, each *ProductFinder*s periodically interact with a remote product service by means of a private *ProductService* artifact, which extends a *HTTPService* artifact providing an operation (`requestProducts`) to directly perform high-level product-oriented requests, hiding the HTTP level.

```
// ProductFinder agent

...

+searchState("start")
  <-  focus("service1");
      !!search.

+!search: keywords(Keywords)
  <-  requestProducts(Keywords,ProductList);
       !processProducts(ProductList,
                        ProductsToAdd,
                        ProductsToRemove);
       addProducts(ProductsToAdd);
       removeProducts(ProductsToRemove);
       .wait({+keywords(_)},5000,_);
       !search.

+searchState("stop")
  <-.drop_intention(search).
```

Fig. 5. A snippet of *ProductFinder* agent's plans.

B. Implementation

The source code of the application can be consulted on the JaCa-Web web site[9], where the interested reader can find also the address of a running instance that can be used for tests. Here we just report a snippet of the *ProductFinder* agents' source code (Fig. 5), with in evidence *(i)* the plans used by the agent to react to changes to the search state property perceived from the *ProductDirectory* artifact - adding and removing a new `search` goal, and *(ii)* the plan used to achieve that goal, first getting the product list by means of the `requestProducts` operation and then updating the

[9]http://jacaweb.sourceforge.net

ProductDirectory accordingly by adding new products and removing products no more available. It is worth noting the use of the `keywords` belief – related to the `keywords` observable property of the *ProductDirectory* artifact – in the context condition of the plan to automatically retrieve and exploit updated information about the product to search.

V. Related Works and Conclusion

To conclude, we believe that agent-oriented programming – including multi-agent programming – would provide a suitable level of abstraction for tackling the development of complex software applications, extending traditional programming paradigms such as the Object-Oriented to deal with aspects such as concurrency, reactiveness, asynchronous interaction managements, dynamism and so on. In this paper, in particular, we discussed the advantages of applying such an approach to the development of Web 2.0 advanced applications, exploiting the JaCa integrated platform.

Concerning the specific application domain, several frameworks and bridges have been developed to exploit agent technologies for the development of Web applications. Main examples are the Jadex Webbridge [14], JACK WebBot [2] and the JADE Gateway Agent [1]. The Webbridge Framework enables a seamless integration of the Jadex BDI agent framework [15] with JSP technology, combining the strength of agent-based computing with Web interactions. In particular, the framework extends the the Model 2 architecture – which brings the Model-View-Controller (MVC) pattern in the context of Web application development – to include also agents, replacing the controller with a bridge to an agent application, where agents react to user requests. JACK WebBot is a framework on top of the JACK BDI agent platform which supports the mapping of HTTP requests to JACK event handlers, and the generation of responses in the form of HTML pages. Using WebBot, you can implement a web application which makes use of JACK agents to dynamically generate web pages in response to user input. Finally, the JADE Gateway Agent is a simple interface to connect any Java non-agent application – including Web Applications based on Servlets and JSP – to an agent application running on the JADE platform [3].

All these approaches explore the use of agent technologies on the *server* side of Web Applications, while in our work we focus on the *client* side, which is what characterises Web 2.0 applications. So – roughly speaking – our agents are running not on a Web server, but inside the Web browser, so in a fully decentralized fashion. Indeed, these two views can be combined together so as to frame an agent-based way to conceive next generation Web applications, with agents running on both the client and server side.

Finally, the use of agents to represent concurrent and interoperable computational entities already sets the stage for a possible evolution of Web 2.0 applications into *Semantic Web* applications [4]. From the very beginning [9], research activity on the Semantic Web has always dealt with *intelligent agents* capable of reasoning on machine-readable descriptions of Web resources, adapting their plans to the open Internet environment in order to reach a user-defined goal, and negotiating, collaborating, and interacting with each other during their activities. So, a main future work accounts for extending the JaCa-Web platform with Semantic Web technologies: to this purpose, existing works such as JASDL [10] and the NUIN project [8] will be main references.

References

[1] JADE gateway agent (JADE 4.0 api) – http://jade.tilab.com/doc/api/jade/wrapper/gateway/jadegateway.html.

[2] Agent Oriented Software Pty. JACK intelligent agents webbot manual – http://www.aosgrp.com/documentation/jack/webbot_manual_web/index.html#thejackwebbotarchitecture.

[3] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[4] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 2001.

[5] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, Ltd, 2007.

[6] R. e. a. Bordini, editor. *Multi-Agent Programming: Languages, Platforms and Applications (vol. 1)*. Springer, 2005.

[7] R. e. a. Bordini, editor. *Multi-Agent Programming: Languages, Platforms and Applications (vol. 2)*. Springer Berlin / Heidelberg, 2009.

[8] I. Dickinson. *BDI Agents and the Semantic Web: Developing User-Facing Autonomous Applications*. PhD thesis, University of Liverpool, September 2006.

[9] J. Hendler. Agents and the Semantic Web. *IEEE Intelligent Systems*, 16(2):30–37, 2001.

[10] T. Klapiscak and R. H. Bordini. JASDL: A practical programming approach combining agent and semantic web technologies. In *Declarative Agent Languages and Technologies VI*, volume 5397/2009 of *LNCS*, pages 91–110, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] T. Mikkonen and A. Taivalsaari. Web applications: spaghetti code for the 21st century. Technical report, Mountain View, CA, USA, 2007.

[12] M. Minotti, G. Piancastelli, and A. Ricci. An agent-based programming model for developing client-side concurrent web 2.0 applications. In J. Filipe and J. Cordeiro, editors, *Web Information Systems and Technologies*, volume 45 of *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, 2010.

[13] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17 (3), Dec. 2008.

[14] A. Pokahr and L. Braubach. The webbridge framework for building web-based agent applications. pages 173–190, 2008.

[15] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In R. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors, *Multi-Agent Programming*. Kluwer, 2005.

[16] A. S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world*, pages 42–55. Springer-Verlag New York, Inc., 1996.

[17] A. Ricci, M. Piunti, L. D. Acay, R. Bordini, J. Hübner, and M. Dastani. Integrating artifact-based environments with heterogeneous agent-programming platforms. In *Proceedings of 7th International Conference on Agents and Multi Agents Systems (AAMAS08)*, 2008.

[18] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming in CArtAgO. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications, Vol. 2*, pages 259–288. Springer, 2009.

[19] A. Ricci, M. Viroli, and A. Omicini. The A&A programming model & technology for developing agent environments in MAS. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Programming Multi-Agent Systems*, volume 4908 of *LNAI*, pages 91–109. Springer Berlin / Heidelberg, 2007.

[20] A. Ricci, M. Virolil, and G. Piancastelli. simpA: A simple agent-oriented Java extension for developing concurrent applications. In M. Dastani, A. E. F. Seghrouchni, J. Leite, and P. Torroni, editors, *Languages, Methodologies and Development Tools for Multi-Agent Systems*, volume 5118 of *LNAI*, pages 176–191, Durham, UK, 2007. Springer-Verlag.

[21] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.