BioAgent: A Multi Agent System Dedicated to Biologists

Emanuela Merelli
Dipartimento di Matematica e Informatica
Università di Camerino
via Madonna delle Carceri
62032 Camerino, Italy
email:{emanuela.merelli}@unicam.it

Leonardo Mariani
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
via Bicocca degli Arcimboldi, 8
I-20126 Milano, Italy
email:{mariani}@disco.unimib.it

June 15, 2002

Abstract

A new agent-based declarative language BioAgent-L is proposed for the definition of applications is Biology. Any application is bound to a specific domain, it consists of an agent, properly created to elaborate several tasks. A task is view as data manipulation in a specific context; the tasks workflow is coordinated by the agent itself. The agent is defined in terms of data knowledge, data manipulation tools useful for application mission. The agent behavior is expressed by a recursive function, ({*approfondire* }) *that allows to formally describe the tasks workflow.*

The new language is fully declarative although ({da dimostrare }) it corresponds functionally to DAML+OIL+... (?) RDF+...+... (?) ...which are procedural languages ({*dire perché* })...

BioAgent-L is implemented in on ...

1 Introduction

In the complex context of heterogeneous access software applications, the nature of data sources makes difficult to design an application suitable to, in a specific domain, extract data from different places and integrate those for a specific task. Usually, a user that aims to solve a given problem by accessing different

distributed data sources, implements a dedicated program, which, step by step, turns into the specific data context. This scenario implies that the user knows, both the nature of the data sources and the tools to access and manage these, different for each context.

(*elencare gli strumenti che sono attualmente disponibili sia a supporto del web workflow che per l'integrazione*)...

The definition of a declarative language makes the user free to concentrate on his application without knowledge of low level heterogeneous access problems. We propose to define a mathematical model suitable to describe the an agent in terms of application domain. An application domain consists of a set of environments...

2 Background

3 A Bio application

A Bio application is defined in a specific domain, as sequence alignments, hybridization data analysis (microarray), proteins interrelations ... Each application consists of a set of tasks, each of one manipulates data by ad hoc tools. Usually the tasks may be processed in pipe, some other times in parallel and rarely they are independent one from the others. As an example, consider an application based on hybridization data experiments, data that in many cases, are available on heterogeneous data sources, and they can be described by MAML, GEML or BSML. Suppose that the data analysis may consists on data mining, functional classifications and clustering. In particular, if the interest is on data clustering by using Euclidean distance, we can decide to discover gene function, then the clustering can be made on experiments (ORF-Open Reading Frame "guilty-by-association, or to define a diagnostic tool, then the clustering can be made on ORFs (transcriptional fingerprints). This means that an application user that aims to study a gene function over a set of distributed experiments must either collect experiments on a data warehouse (this means that during the transformation from local to global data become homogeneous but obsolete) then apply the clustering algorithm or it leaves the data on local places but it makes the clustering algorithm distributed over the places. The latter case owns the advantage that the algorithms work on a quasi real time updated data and the network traffic is optimized for the application purpose. We are aware of the existence of general problems as data inconsistency, data incompleteness, data privacy, but we also believe and share the following opinion *if the data owner wants his data to be used from the scientific web community he must offer his data either in a standard form or equipped with proper access tools.* On this belief we propose an high level language suitable for a user who must own knowledge only for the application domain.

4 Software Architecture

The BioAgent architecture is a layered software architecture (see Fig.1). It is made up of different layers, each one providing features at increasing level of abstraction. This kind of software architecture presents several advantages [1]: complex problem could easily be partitioned into several steps to be performed incrementally, the architecture is easy to update and reuse.

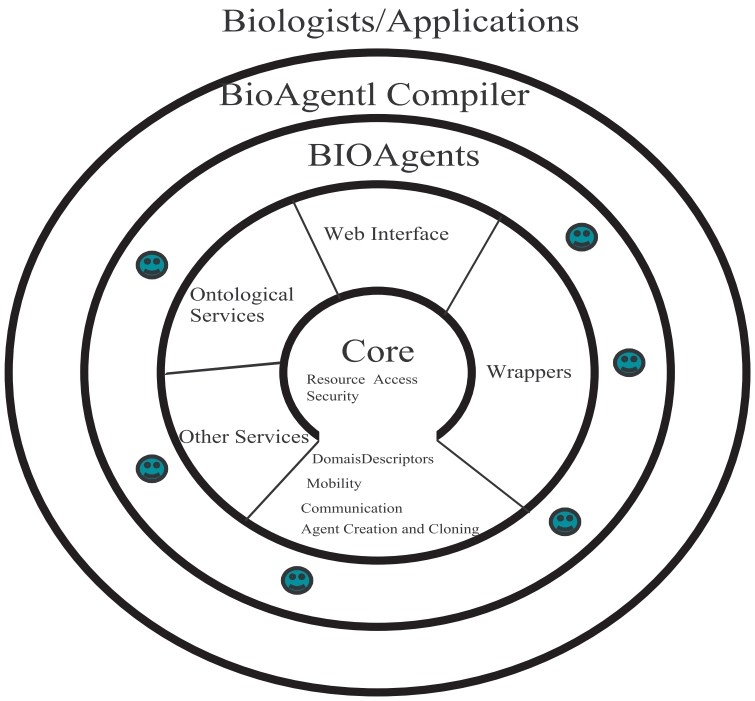

Figure 1: BioAgent Software Architecture

4.1 Core Layer

The *core* layer contains two distinct set of features. Features exposed to *services* and features exposed to *agents*. Features exposed to services are critical features, in general related to system security or safety, and it would be very dangerous to give access to these primitives to user agents. At this moment our architecture contains two critical features: security and resource access.

Security is the component that intercepts any request, invocation or instantiation, in order to verify if the caller has got the right permission to perform the operation. The security component also checks any incoming agent in order

to prevent malicious agents to improperly use the system. Agents that may be dangerous or without having the needed permission are not accepted.

We can say that the security component acts as a filter whenever a core primitive is invoked.

Another interesting feature is the management of services. The core keeps trace of each running services and provides to any new agent the list of all accessible services. Each agent can access to a subset of available services on the base of its permissions. The set of permission in a place is defined as soon as the agent migrates or is created. In the eventuality, an agent tries to use a service without having the right permission, the core layer intercepts this attempt when the agent deploys a message to the service. In this case the core destroys the message and can also takes any precaution against this agent.

The resource access is managed by a resource manager. The resource manager provides all basic primitives to access to physical stored data. All controls on the permission to access to local resources by a user agent are left to the security manager. The specific service could also implement additional features for the control or for the limitation of permissions. It is possible to observe that the agent never gains direct control or direct access to a resource, but it can collect only responses to it request. If an agent passes the security control and permission check and also any other permission policies encapsulated in the service it wants to use the resource manager could not denied any operation for two reason, the first is that this feature, in our system, is not conceptually in the resource manager, the second is that the resource manager is not aware of which agent made the request. The resource manager knows what it is doing, but not why.

Basic features exposed to agents are *Mobility*, *Communication*, *Agent Creation and Cloning* and *DomainDescriptors*.

Mobility is the support to agent migration. The type of migration supported by our system is the *weak mobility*, so there is the loss of the agent's execution after a migration is performed. The use of the weak mobility is not a problem because the agent preserve its internal state and it uses this information to continue is work after the migration.

Agent creation and cloning supports agents on creation of others agents. It is possible to create agents in two different way. An agent A can clone itself creating an agent B, in this way the new agent B is an identical copy of the creator A, excepts for the unique identifier. The other possible procedure is to create another agent without using its own internal state as initializing data. The initializing data is provided explicitly by the creator in the way it prefers.

The agent creation is invoked by an agent, but it is monitored by the core in order to avoid the presence of too much agents. The uncontrolled creation of agents could lead to the situation where the system collapse due to the presence of too much agents. Leaving the agent creation to the core, permits fine management politics, for example to limit the number of agents created by each agent or limit the type of agents simultaneously present on a place.

The domain descriptors is the part of the security manager exposed to user agent. Essentially it contains the description of the local domain in terms of

services available and exports methods to read this information.

Only using exposed core primitives any agent is sure to be able to communicate, migrate, create other agents, clone itself or obtain a description of services running, and this set of primitives are available on each running platform.

Using only the core primitives an agent is not able to access resources. This is correct because an agent directly accessing resources would violate any security policy. Services guarantees the right level of abstraction and limitations on resource access.

4.2 Services Layer

Upon the core layer there is the *Services Layer*. In this layer is possible to implement any service reusing the primitives provided by the core. Services have got full access to the layer below, so services can use security and resource access. In our framework services are implemented as a special kind of agents: *service agents*. So when we talk about services we talk about service agents.

This point of view gives a lot of flexibility to the system, in fact user agents access to services communicating with service agents. Any services can be added at any moment simply adding an agent and all interactions take place by exchanging messages using standard communication. So, in order to provide a new service, it is necessary to implement the service agent and a communication protocol. Any user agent will use this service exchanging messages responding to the defined protocol.

We begin developing three services of general applicability and very often used: *Web Interface*, *Wrapper* and *Ontological Services*.

The main task of the Web Interface service is to manage interactions among user and platform. Using this services the user is able to send its own agents, send commands to agents, receive responses and collect any partial or final results. This service agent represents the user alter ego in the agent system, so when the user agent, while performing a task, want to comunicate with the user, it simply send a message to the Web Interface agent. Then the Web Interface agent will manage all interactions with the user and will send back, if necessary, a message as answer.

The use of a service agent as a user alter ego is not only a paradigm choice. In fact, the user agent has not to have all components to manage interactions with user, but it uses the standard communication primitives. These primitives are always provided by the place, so the user agent is a very small execution unit that uses features exposed by the core to comunicate with other agents, users and services.

The wrapper is the service that gives access to local resources, abstracting from their nature, and providing a uniform representation, for instances using XML. This structural or syntactic abstraction is powered by the ontology services that permits to understand the semantic of data. If the ontology service is not available any interpretation of extracted data is left to the agent.

The ontology service manages local ontologies used to give semantic on data locally stored. Agents can use these ontologies to understand extracted data, otherwise it can choice to use its own ontology. Generally the latter situation happens when the ontology service is not present.

This set of services could be extended at any time as explained before.

4.3 Agent Layer

The next layer is the *Agent Layer*. This layer is populated by agents accessing local services and local primitives. The behaviour of an agent is defined by its creator or by its owner. There are no constraints on the heterogeneity of agents present, so each user could develop and implement its agent and then send it in the BioAgent system. This will work if the user has extended our UserAgent Java class.

4.4 Compiler

The creation of an agent is not a simply task. It involves knowledge in any programming language (usually Java, as for our platform) and a general user is not so skilled. So the last layer provides a declarative language, BioAgent*l* and its compiler, to create and manage agents. This language permits to specify the type of agent the user intend to use and a tasks to be performed. This is possible without knowing any programming language. After have written a few statement in this declaring language a compiling operation will generate the agent ready to perform all tasks we desire. The user has not to verify if a service is available or if a platform is available, all this problem are encapsulated in the semantic of the language and in the agent generation and execution.

This layer is directly usable by a user, but can also be used by an application.

5 Data Abstractions

We can reason about our platform in term of data abstraction, so we can work with different type of data in different type of situations. Generally a user will work using the highest level of abstraction (Ontology), but it is also possible to develop agents able to work directly at other level of data abstractions. These level are:

- *Domain*: Ontology and Domain Based Operation
- *Context*: Data Structure and Algorithms
- *Environment*: Data and access methods

It is possible to understand correlations between each layer both following a top-down or a bottom-up approach.

In a *top-down* approach, we begin considering ontology. An Ontology describes the domain of the conversation and it addresses to represents *concepts*. At this level there is the need to perform operations related to a specific domain, so domain based operations or other expressive languages (logic based languages) are necessary to manage and extract information. Ontologies represent concepts, domain based operations represent the way to extract *Knowledge* from concepts. In our view, the domain level is well described in term of concepts and knowledge we want to extract and manipulate. Putting together ontology and domain based operations is how to put together concepts and knowledge.

Each ontology is concretized into one or more data structures and the same term in an ontology could be mapped to more than one possible data structure. Data structures is not as expressive as ontologies are, but they are used to define *how data is stored*. Ontology try to describe the semantic of data, meanwhile *context* describes the syntactic aspects of data concretization. Knowing data structure it is possible to define algorithms working on data. These algorithms define natural *computation* on a certain kind of data. So, algorithms are the counterpart of domain based operation at the context level. Obviously, as ontologies are concretized into one or more data structure, in the same way domain based operations make use of algorithms.

At the lower level we find concrete physical *data*. Each data instance type has got a well defined structure, but the same kind of ontological information could be splitted into several data instances of different structure. At this level the equivalent of domain based operations and algorithms are *access methods*. Access methods are used by algorithm to access and manage data.

Following the *bottom-up* approach we first find data and access methods to data. Unstructured heterogeneous data sources are difficult to manage so we need to add the structure level. Knowing data structure we defining algorithm that takes advantages from hypothesis on the structure.

Structured data lacks of semantic so we add ontologies that try to abstract structured data to a common conceptualization view. Finally we use domain based operations or logic to strongly relate algorithms to the applicative domain.

These level of data abstraction are mapped in the software architecture. The environment hosting physical data is encapsulated in the core layer. Access methods are encapsulated and can be used only by services. The wrapper implements the context abstraction, so it knows the data structure and provides algorithms to manipulate data. Finally the ontology level is implemented by the ontology service. The ontology service provides ontology and domain related operations. Agents usually works at the ontology abstraction because the ontology service in general is available. In other situation, for example if the ontology service is not available or if the agent explicitly wants to operate at the context level, the agent can use the wrapper service to work at the context data abstraction. If nobody of these two services are available the agent cannot access data because the environment level is restricted to service agents.

The ontological level is defined in a standardized way and is shared by the community, or probably there is a reasonably good ontology shared in the agent society. The physical data layer is a standard de-facto, simply because it exists before the system is built. The structure level also exists, but generally it is wrapped to another different one easily manageable .

6 Software Agent Architecture

When an agent is created, it has automatically defined a set of attributes. These attributes define the *agent's context*: agent's domain, agent's mission, agent's tools and agent's knowledge.

6.1 Agent's Context

6.1.1 The Agent's Domain

The agent's domain is made up of public and privates ontologies. *Public ontologies* are ontologies inherited from the domain layer and that the agent can use to reach its own purpose. Exactly these are the ontologies provided by each place.

The *private ontologies* are ontologies defined internally an agent. The agent keeps with it these ontologies and uses them as support to its mission. At the agent creation phase private ontologies have to be specified. During its life the agent carries on private ontologies and use local public ontologies. So public ontologies can migrate as consequence of a migration, but private ontologies are entirely managed by the agent who owns them.

6.1.2 Agent's Tools

Agent's tools are the set of domain related operation the agent can perform. As for ontologies agent's tools could be divided into public and private tools.

6.1.3 Agent's Mission

An *agent's mission* is formalized as a task that the agent has to perform. Tasks are described by workflows, see 6.3.

The more useful and recurrent tasks are available to be directly reused without to define them from scratch each time.

6.1.4 Agent's Knowledge

The *agent's knowledge* is a set of facts stored in the agent's memory in a consistent way respect to ontologies (private or public).

6.1.5 Agent's Action

Each agent has got a primitive defined set of *actions*. These actions are used in the workflow to act in the domain.

6.2 Operational Part

6.3 Task and Mission Management

The agent, in order to accomplish its mission, has to execute a set of actions (tasks), both primitive or service provided. To specify a *task*, that will be used as plan for a mission, we must define a workflow of actions. In a workflow, actions are combined together in order to achieve a specific goal. Combining different tasks we obtain more complex tasks.

We could summarize the task definitions problem in this way. Initially we could use primitive actions and service provided actions in order to define tasks. At this point, we could define complex task by using all kind of actions and previous defined tasks. By combining together all this stuff we obtain very complex tasks that could be used as mission for an agent.

6.4 Primitive Action and Services Management

7 How BioAgent*l* Will Be

Agents acting in our framework have a *task* to accomplish. In order to accomplish their task it is necessary to define a workflow which completion means a successful mission. Tasks and actions could be briefly defined as follow:

- task: a workflow of task and/or action
- action: a primitive action or a service provided action

A *primitive action* is an action defined on the agent, so it can be performed on any place at any time. This is not exactly true, because a complex action could contain dependences on the local environment, for example its success could depend on the presence of a particular kind of service. In a primitive action any dependence is managed internally.

A *service provided action* is an action supported by a services. This action is accessed sending a message to the service agents. Response is collected in the same way. Obviously, service provided actions change in every place.

Combining actions of both type we obtain *simple tasks*. If we combine simple task with actions or other simple task we obtain *complex tasks*. If we do not need to distingue between simple and complex task, we will refer to them simply by *task*.

8 Application on Biology

9 Conclusions

References

[1] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Descipline*, Prentice-Hall, 1996.