Journal of Intelligent Information Systems, X, 275–298 (X)

Software Agent-Oriented Frameworks for Global Query Processing

ZAKARIA MAAMAR[†,¶,§], BERNARD MOULIN[¶,§], GILBERT BABIN[¶] AND YVAN BÉDARD[§]

zakaria.maamar@drev.dnd.ca,{moulin,babin}@ift.ulaval.ca,yvan.bedard@scg.ulaval.ca

[†]*Interoperability Group, Information System Technology Section, Defence Research Establishment Valcartier, Val-Bélair Canada.*
[¶]*Computer Science Department and* [§]*Research Center on Geomatics, Laval University, Ste-Foy Canada.*

Received November 25, 1995; Revised March 30, 1996

Editor: Larry Kerschberg

Abstract. The paper introduces the concept of software agent-oriented frameworks for global query processing in an interoperable environment. Such an environment is developed for the purpose of making cooperative interactions between several systems. These latter are distributed on networks and may present several incompatibilities. The global query processing is applied to the SIGAL project which aims at developing an interoperable environment for georeferenced digital libraries. This environment provides users with services that will free them from worrying about information distribution and disparities.

Keywords: software agent-oriented framework, global query processing, georeferenced digital library.

1. Introduction

With the increasing development of information technologies, different types of systems exit. These systems are *distributed* across the communication networks, present several *incompatibilities* at different levels (material, software, knowledge), and are usually used *independently* from each other. Combining the services provided by such systems to answer user's needs is a much more complex operation. It becomes important to assist users to perform their operations [7]. A possible solution is to set up an environment for *systems interoperability*. Interoperability is a process that allows *cooperative interactions* between distributed and heterogeneous systems. In an interoperable environment, answering user's needs usually means *global query processing*.

An interoperability process has to keep the local systems *autonomous* and *independent* [7]. To reach this objective, specialized components, called *software agents* [8, 10], may be used. These agents have the capabilities to act on the behalf of the systems and to execute operations needed during global query processing. Given the complexity of such processing, however, we propose to gather these agents into teams evolving in what we call *software agent-oriented frameworks* [7]. Such frameworks can interact in order to define a global behavior which is an outcome of such interactions. Furthermore, these frameworks can be adapted in terms of components (types of teams of agents and agents to integrate) and functionalities (types of services to offer). Moreover, these frameworks can play several roles according to the application to be developed. Finally, these frameworks can constitute a multiframework environment [7].

To design and specify software agent-oriented frameworks, we rely on different studies conducted in the fields of *distributed artificial intelligence* [9] and *distributed objects* [11, 12]. Each software agent-oriented framework is composed of several teams of agents that are able to achieve services offered by the framework and invoked by users [7]. In this paper, we show how software agent-oriented frameworks are used in designing interoperable environments and processing users' global queries.

Our research on software agent-oriented framework is applied to the SIGAL[1] project in which we aim at developing an *interoperable environment for Georeferenced Digital Libraries* (GDL). A GDL is an information base describing geodocumentary resources available in an organization. Currently, each GDL is characterized by its own informational content, its own presentation formats, and its own processing functions. All these differentiating elements urge the user to adapt his behavior to each GDL requirements, which is quite an impossible task. In order to help users to process their global queries, we are working on developing an interoperable environment of GDLs. With such an interoperable environment users will have direct access to services without being obliged to have a background knowledge on the individual characteristics of the interconnected GDLs.

The paper is organized as follows. Section 1 proposes an overview of interoperability issues. Section 2 presents basic notions of software agent-oriented frameworks. Section 3 describes the SIGAL environment. Section 4 introduces relevant aspects of the global query processing as applied to the SIGAL environment. Section 5 describes an implementation of the SIGAL environment. Finally, Section 6 summarizes the main points brought out by our research study.

2. A Software Agent-Oriented Framework

Designing software agent-oriented frameworks is a recursive activity that relies in turn on frameworks, teams of agents, and software agents. In this section, we present the main characteristics of a software agent-oriented framework and its operation mode [7].

A *software agent-oriented framework* offers a set of *services* that can be requested either by users or by other frameworks. A framework is an environment composed of a *framework supervisor* and one or several *teams of software agents*. The services provided by a framework are performed by different teams set up by the framework. These teams are composed of agents selected from a bank of software agents. This bank contains several agents having different functionalities which are specific to the application to be developed and to the characteristics of the information systems to be interconnected.

Teams of agents are made up of several software agents and are structured differently according to their responsibilities in the framework. A team is characterized by a *team supervisor* and a set of roles that agents must fulfill according to their capabilities. A role is identified by a list of parameters: *role label* (such as coordination), *results to provide*, and offered and required *services* when playing the role. To carry out their operations, agents need to cooperate and exchange information.

Services provided by a framework satisfy specific users' needs such as browsing, query formulation, information retrieval, etc. When a service is invoked by a user, the framework's supervisor agent activates a scripting procedure, called a *realization scenario*, which specifies the characteristics and the interaction protocols of the teams of agents that will perform

the different operations required to carry out the service. Interactions can be defined between different kinds of components: frameworks, teams of agents and software agents. According to that realization scenario, the framework supervisor creates teams that will play roles specified in the scenario. At their own level, team supervisor agents activate realization scenarios in order to coordinate the activities of their software agents.

3. Presentation of the SIGAL environment

This section introduces the SIGAL environment [7]. We present the multiframework architecture developed for GDLs interoperability. We also describe the knowledge base (called Ontology) used to reconcile informational discrepancies among GDLs.

3.1. An Overview of GDLs

Within an organization that manages spatially referenced data, several types of *documents* are used to describe the presence and the nature of these data. Such documents include cover maps, aerial photographs, etc. Managing such documents is a complex task; each one is characterized by its own scale, content, quality, sources, and format. GDLs are information bases describing the available geodocumentary resources. As a result, GDLs can improve knowledge of the nature of data, identify the responsibility of "who does what, when, and how", and inform about the physical location of the documents.

There exists a number of well known GDLs projects, such as Alexandria Digital Library (Santa-Barbara University) and GEOREP (Laval University). Users are accustomed to use these GDLs independently from each other. Yet, a more complex task is the *simultaneous use* of services provided by different GDLs. This new reality requires new alternatives for GDLs interoperability. If a person visits the several available GDLs, she will easily notice that [14]: the content of the GDLs differs from one to another; the data standards are different, if not absent; different words are used to represent the same concept and *vice-versa*; the user interfaces always differ; etc. All these differentiating elements would need that users adapt to each GDL's requirements and understand their different information and structural characteristics. However, this seems quite impossible.

3.2. SIGAL's Architecture

The SIGAL environment (Fig. 1) is an interoperable architecture characterized by the use of a *client/server* approach, the implementation of *mirror sites*, the use of three types of frameworks (server, client, local-source), and the dynamic generation of client frameworks, based on users' needs [7].

The design of the SIGAL architecture applies principles elicited in the field of information systems interoperability [4, 5, 16], namely that an interoperable environment must: maintain the autonomy and independence of the systems to be integrated, while allowing these systems to interact despite their disparities; reduce the informational disparities of the various interconnected systems by using a knowledge which is understood by all these

Figure 1. Multiframework architecture of the SIGAL environment

systems; and, help users satisfy their needs without worrying about the distribution and disparities of the information provided by the interconnected systems.

By analogy to information agents [16], the local-source frameworks maintain the autonomy and the independence of the GDLs in the interoperable environment. Therefore, local-source frameworks interface GDLs with communication networks, and process the data requests sent by client frameworks. The server framework is the backbone of the SIGAL environment; it monitors all the operations needed to support services offered to users and to other frameworks. To avoid *overloading* the server framework, we suggest to duplicate it on *mirror sites*. In highly distributed environments, the technique of mirror servers is used to orient users toward less loaded servers. However, in such an architecture, it is important to maintain the coherence of the information duplicated on the server frameworks and consequently to define *reliable update protocols*. When users need information from several GDLs, they invoke relevant services from the server framework. The invocation of such services initiates the creation of a *client framework* on the *user's machine*. This client framework is composed of the teams of software agents needed to fulfill the service. Hence, the server framework delegates operations to the client framework and limits its involvement to the monitoring of these operations. Once all operations are completed, the client framework can either be deleted or stored for later use.

3.3. SIGAL's frameworks

The SIGAL environment uses three types of frameworks: server, client, and local-source.

The Server Framework is responsible for carrying out services required by users, or necessary to maintain SIGAL's architecture (Fig. 2). The server framework contains a single team composed of several agents: *Coordinator*, *Help*, *Scenario*, *Interaction*, *Domain*, and *Learning*. Moreover, this team has access to three informational resources; the *Service*, *Scenario*, and *Ontology* Bases.

Figure 2. Server framework internal architecture

The Service Base is accessible through the Help-Agent and contains a list of user and SIGAL services offered by the server framework. The Scenario Base is accessible through the Scenario-Agent and contains the realization scenarios available to the team. The Scenario Base contains several components: (1) formulation patterns used to help a user specify his needs according to the vocabulary and the knowledge contained in the Ontology Base, (2) characteristics of agent roles needed to carry out the scenarios, and (3) procedures used to build the plans that are executed by the agents during the scenario. The Ontology Base provides a detailed description of the knowledge contained in each GDL. In a server framework, the Coordinator-Agent acts as a global supervisor of realization scenarios initiated by this framework: it monitors the advance of the agent teams executing these scenarios. The Interaction-Agent allows the server framework to interact with client frameworks, local-source frameworks and other server frameworks (mirror and/or principal sites). Finally, the Learning-Agent acquires new knowledge and transfers it to the agents of this framework.

The Client Framework is created by a server framework, when accessed by a user requiring a service. The client framework (Fig. 3) is set up *temporarily* on a user's computer in order to fulfill his needs.

Figure 3. Client framework internal architecture

The client framework is composed of a *Coordinator-Agent* and of one or several teams according to the number of GDLs necessary to process user's request. The Coordinator-Agent is created by the server framework. In turn, this agent creates the *main team* of software agents and assigns it the activities prescribed in the realization scenario which identifies the invoked service. When required, the Coordinator-Agent also creates secondary teams. The main team comprises three agents (Interface, Resolution and Interaction) and is responsible for processing the user's request as well as for identifying which GDLs must be accessed. Based on the number of GDLs needed, the main team asks the Scenario-Agent of the server framework for additional agents. One or several *secondary teams* are created; each containing an Interaction-Agent and a Resolution-Agent. The main team is the only team allowed to communicate with the server framework. Each team (main or secondary) interacts with a specific local-source framework in order to solve the sub-problem at hand.

The Local-Source Framework (Fig. 4) interfaces with a GDL of the SIGAL environment (Fig. 4). It is assigned to a specific server framework in order to keep it informed about the changes occurring in the informational content of the GDL. When such a change occurs, the Domain-Agent of the server framework updates its *Ontology Base*; these updates are then propagated to the other mirror servers. This mechanism preserves the *coherence* of the various copies of the Ontology Base through the network. All update operations are specified by *update scenarios*, that belong to SIGAL services. A local-source framework also processes *data requests* directed to its associated GDL.

Figure 4. Local-source framework internal architecture

The local-source framework is made up of three agents set in one team of agents (Coordinator, Interaction, and Knowledge). The *Coordinator-Agent* sets the other agents and monitors the operations performed in the local-source framework. The *Knowledge-Agent* interacts with the GDL in order to get the data requested by the *Resolution-Agents* of the client framework. This data is transmitted by the *Interaction-Agent*, which allows the local-source framework to interact with server and client frameworks.

3.4. SIGAL's Agents

Based on SIGAL's architectural characteristics (cf. Section 3.2) and on the services offered by the SIGAL environment (cf. Section 3.5), different types of agents have been identified [7]: *Coordinator-Agent*, *Domain-Agent*, *Help-Agent*, *Interaction-Agent*, *Interface-Agent*, *Knowledge-Agent*, *Learning-Agent*, *Resolution-Agent*, and *Scenario-Agent*. Among these agents only those needed in a global query processing are hereafter described:

Domain-Agent resolves knowledge disparities among GDLs by using the common knowledge of the SIGAL ontology.

Interface-Agent assists users in specifying their needs by proposing a set of *formulation patterns*.

Knowledge-Agent knows the protocols through which a GDL accepts requests and provides back information. This agent also monitors informational updates occurring in the GDL.

Resolution-Agent processes user's global queries. The resolution process may require the decomposition of a query into sub-queries, each of which is sent to the appropriate GDL.

Interaction-Agent mediates interactions between either teams or frameworks, by handling the exchange of informational and structural components required by the frameworks and their agents. The Interaction-Agent is located in all the types of frameworks.

3.5. SIGAL's services

Two categories of services are offered by the SIGAL environment; i.e., the server frameworks [7]:

1. User services correspond to global query processing. They fulfill needs involving several GDLs. The user formulates his information query independently from the distribution and heterogeneity constraints of the GDLs.
2. SIGAL services support the insertion or deletion of a GDL, as well as the modification of GDLs' informational content.

User services fulfill needs that involve several GDLs. The user formulates his information request independently from the distribution and heterogeneity constraints of the GDLs. Figure 5 presents the interaction diagram describing how to realize a user service. In the following paragraphs, numbers refer to operation numbers in Figure 5.

Figure 5. Interaction diagram of user services

The user consults the list of services proposed by the Help-Agent (1). When a service is identified, this agent initiates its realization (2) by informing the Coordinator-Agent of

the server framework. The Coordinator-Agent performs the operations characterizing the service and asks the Scenario-Agent to select the relevant scenario (3). After the scenario has been identified the client-framework components are selected (4) and sent to the client's platform by the Interaction-Agent (5). The client framework is composed of a Coordinator-Agent and a main team of agents (6) which is composed of an Interface-Agent, a Resolution-Agent and an Interaction-Agent. The client framework must fulfill user's request (also called a *global query*). Once this request has been formulated (7), the Interface-Agent sends it to the Resolution-Agent (8), in order to determine which GDLs are needed to satisfy the request. To perform this operation, the Resolution-Agent uses the information provided by the Domain-Agent of the server framework (9, 10, 11, 12, 13, 14). Depending on how the selected GDLs are distributed, the Resolution-Agent asks the Scenario-Agent for new teams for the benefit of the client framework (15, 16, 17, 18, 19). The number of agent teams set up in the client framework depends directly on the number of GDLs to be queried; in fact, each team (including the main team) is assigned to only one local-source framework. The goal is to request relevant data from these selected GDLs (20, 21, 22, 23, 24, 25, 26). After processing these data, all the teams send their results to the Resolution-Agent of the main team (27). The final answer is sent to the user by the Interface-Agent services (28).

SIGAL Services support operations that allow insertion or deletion of a GDL, as well as the modification of GDLs' informational content.

Modification of GDL-Informational Content any change occurring in a GDL must be pointed out by its local-source framework to the assigned server-framework (Fig. 6). The objective is to keep a total coherence between the informational content of the GDLs and that of the Ontology Base.

Figure 6. Interaction diagram of SIGAL services, modification case

The Knowledge-Agent of the local-source framework monitors the behavior of a specific GDL. During each update, it contacts (1, 2) the Interaction-Agent of the associated server framework. Once it gets this information, the Interaction-Agent sends it (3) to the Domain-Agent of the server framework. Subsequently,

this agent performs the maintenance operations (4) and transmits a request for spreading the update to the other server frameworks (principal and/or mirror sites) thanks to the Interaction-Agent (5, 6). Since the SIGAL environment brings about a total replication situation, it is important to keep the Ontology Base coherent by managing confirmations of update operations (7, 8).

New GDLs Insertion any new GDL that has to be integrated into the SIGAL environment is identified by a local-source framework which is assigned to a server framework (Fig. 7). The insertion of a new GDL is triggered by the invocation of the appropriate service of the server framework, which is done by the person (named system administrator) responsible for managing this framework. According to this service, the Help-Agent (1) has to contact the Coordinator-Agent to inform it about the need for setting a scenario modifying the SIGAL's architecture (2). This scenario is managed by the Scenario-Agent and is available in the Scenario Base (3). The realization scenario in this context is characterized by two elements: processing programs and formulation patterns for the Domain-Agent of the server framework, and processing programs and software agents for the new local-source framework. After sending the components of the new local-source framework and after its assignation to a server framework (4, 5), the Scenario-Agent sends the update scenario to the Domain-Agent in order to update the Ontology Base (6,7). When these update operations have been executed, the Domain-Agent sends a request for spreading the update to the other server frameworks (8, 9) thanks to the Interaction-Agent. Since the SIGAL environment brings about a total replication situation, it is important to keep the Ontology Base coherent by managing confirmations of update operations (10, 11).

Figure 7. Interaction diagram of SIGAL services, insertion case

3.6. SIGAL's Ontology

An ontology offers a common terminological basis for the various interconnected systems. Hence, it reduces the risks of exchanging inconsistent information.

In SIGAL's case, *ontological disparities* exist at different levels (cf. Section 3.1). First, being generally developed independently, GDLs may use different terms to describe their data: this difference makes it difficult for users to simultaneously consult several GDLs. Second, current GDLs do not help users when formulating their requests. Moreover, a user has to express his needs according to his own vocabulary and his own understanding of the application domain.

When defining the SIGAL's ontology, we used the concept of *meta-data* [5, 6], defined as *data which describe data of the analyzed domain*. The description allows the specification of the data structures, their domain values and their functional and semantic interpretations. We have developed a meta-data model which provides a concise description of the information manipulated by each GDL. This meta-model, called *Ontology Base*, is managed by the Domain-Agent of the server framework. It is illustrated by Figure 8, based on OMT [15] formalism. OMT is a method to design object-oriented systems.

Figure 8. The meta-model of the SIGAL environment

The Ontology Base is made up of *meta-entities* and *meta-relationships*. For example, the meta-entity *Entity* describes all the existing entities in the GDLs data models; whereas, the meta-entity *Rule* represents *semantic-equivalence links* that exist between different GDLs.

4. Global Query Processing in the SIGAL environment

In the SIGAL environment, global query processing consists of four steps. In the first step, a user formulates his needs using formulation patterns (provided by the Interface-Agent of the client framework) and the Ontology Base (provided by the Domain-Agent of the server framework). In the second step, the needed GDLs are identified according to the *autonomy criteria* (cf. Section 4.1) and the adequate sub-queries are generated. In the third step, these sub-queries are processed by requesting data from the relevant local-source frameworks. In the last step, the answer is provided[2]. All these steps rely on the Ontology Base.

4.1. User's Needs vs. Needed GDLs

Using the Ontology Base (Fig. 8), the Domain-Agent of the server framework determines which GDLs should be queried to answer the user's global query. The number of agent teams set up in the client framework depends directly on the number of GDLs to be queried; in fact, for each GDL we correspond one team.

In an interoperable environment, a global query is usually decomposed into a set of *sub-queries* [1]. Depending on how the different systems interoperate, the sub-queries will take different *syntactical* and *semantically* forms. Specifically the type of an interoperable environment will define how sub-queries are generated and how these sub-queries relate to one another. To assess the interoperable environment type, we propose the *autonomy* criteria and the *complementary* criteria as follows:

Autonomy criteria : local systems are independent and do not need to cooperate. A sub-query, identical to the global query, is generated for every system which is able to satisfy the query. Hence, the sub-queries obtained from the global-query decomposition are semantically *identical*, but use *different* concepts in their formulation.

Complementary criteria : local systems are not independent and need to exchange data. A sub-query is generated for each system which contains a certain amount of the information needed to satisfy the query. Hence, the sub-queries are semantically *different* and use *different* concepts in their formulation.

The SIGAL environment satisfies the autonomy criteria.

4.2. Global Query Processing Approach

Four steps are needed to process a global query: query formulation (cf. Section 4.2.1), query refinement (cf. Section 4.2.2), determination of join links (cf. Section 4.2.3), and execution of data requests (cf. Section 4.2.4). In order to illustrate these steps, we will use the following global query: *Find all the maps that have a 1/10000 scale, contain information about rivers, and are available on a magnetic support.*

4.2.1. Query Formulation In the SIGAL environment, a user service is processed by a *realization scenario* [7]. Once a scenario has been identified, the server framework sends a

set of components, including *formulation patterns*, to the client framework. These patterns are instantiated through information obtained from the Ontology Base (Fig. 8). Hence, for each pattern, a set of meta-instances[3] (or instances of meta-data) is defined.

Figure 9 represents SIGAL's user interface. Our objective is to use the SIGAL's ontology to guide the user when formulating his query. For each field to fill, some attributes are proposed, along with their description. The user selects the relevant attributes in order to define his query.

When the user has selected the relevant attributes, he can instantiate them by providing their values. In that case, the user needs information from the GDLs. We suggest three ways to perform the instantiation operation:

1. Return the list of all attribute instances, when the list is small.
2. Assist the user by describing the possible instances using examples and definitions.
3. Tell the user that the number of instances is too large, and that he has to decide whether he wants to get them or not.

4.2.2. Query Refinement The global query analysis has to consider the *semantical disparities* that could exist between the knowledge contained in the different GDLs. A part of the Ontology Base provides a detailed view of the GDLs' data models (Fig. 10). We know what are the available attributes; to which applications, entities, and relationships these attributes belong to; what attributes are equivalent; etc.

From the meta-model of Figure 10, we define a conceptual relation called *LINK(a,b,c)*. It is illustrated using Sowa's conceptual graph formalism. The relation specifies that an attribute *a* belongs to entity (or relationship) *b* of application *c*.

relation LINK(a,b,c) is
[BELONG-TO]
→ (AGT) → [ATTRIBUTE: a]
→ (OBJ) → ([ENTITY: b] (OR) [ASSOCIATION: b])
↓
(PART) → [APPLICATION: c]

Identifying relevant GDLs The global query can be decomposed into two parts:

1. The data to retrieve; i.e., the attributes' values to be returned to the client framework. The set of these attributes is the *projection* set P. In our example, P = {map_id} ("find all the maps ...").
2. A condition (possibly absent) characterizing the data to retrieve. The attributes used in specifying the condition constitute the *condition* attributes set C. For our query, C = {map_scale, map_theme, support_type} ("... which have a 1/10000 scale, contain information about rivers, and are available on a magnetic support").

For each attribute mentioned by the user in sets P and C, the Domain-Agent identifies the entity (or relationship) and GDL in which the attribute exists. This operation is performed

Figure 9. SIGAL's user interface

Figure 10. Part of the SIGAL environment meta-model

using the relation *LINK(attribute_name,?b,?c)*. Then, the Domain-Agent determines the attributes which are *equivalent* to those already presented in the query. We note by E_{att} the set of attributes equivalent to attribute *att* (including *att* itself), and by GDL_{att} the set of GDLs containing the attributes in E_{att}. Once all the attributes have been identified, the set of *needed GDLs*, named A, is determined. Two cases exist:

1. The condition only uses the operator "and" ($\wedge$)
   $A = \bigcap_{att \in (P \cup C)} GDL_{att}$
2. The condition uses both the operators "or" ($\vee$) and "and" ($\wedge$)
   We do not consider this case, since it is equivalent to formulating two separate queries (cf. point 1).

The variable A, therefore, is the set of GDLs which includes all the attributes of the global query. If this set is empty (A = ϕ), the query cannot be satisfied. Indeed, some GDLs do not contain all the attributes required by the user. Table 1 shows the different attributes used in the query, either from the projection set P or from the condition attributes set C. For each of these attributes, we present E_{att} and GDL_{att}, the set of equivalent attributes and GDLs containing these equivalent attributes, respectively.

The set A of the needed GDLs which satisfy the global query is obtained as follows:
$A = \{\text{GDL}_{att_1} \cap \text{GDL}_{att_2} \cap \text{GDL}_{att_3} \cap \text{GDL}_{att_4}\} = \{\text{GDL}_1\}$.

Resolving data values compatibility problems After the set A has been determined, it is necessary to solve a *compatibility* problem related to the *restriction values* (attribute$_i$ = value$_1$, attribute$_j$ <> value$_2$, etc.). At the beginning, the user has chosen attributes to retrieve and has formulated a selection condition in his query. The condition contains value constraints to apply on instances of the GDLs attributes (example, map_scale = 1/10000). The Domain-Agent has also added attributes equivalent to those already selected by the user. These added attributes belong to other GDLs and probably have *different* data structures and *different* value domains. For example, suppose that a user is looking for maps having an attribute

Table 1. Attribute links with GDL

Sets	Attribute att_i	Equivalent attributes E_{att_i}	Application GDL_{att_i}
P	att_1 = map_id	map_id	GDL_1
		map_key	GDL_2
		key	GDL_3
C	att_2 = map_scale	map_value	GDL_3
		map_scale	GDL_1
	att_3 = map_theme	map_theme	GDL_3
		map_content	GDL_1
			GDL_2
	att_4 = support_type	support_type	GDL_1

map_content with the value "road". If other attributes are semantically equivalent to the attribute map_content, as map_role and map_information but have respectively "path" and "highway" values, do we have to consider or to ignore instances of these new attributes? We propose a solution to this problem: attribute domain values represent properties of elements found in the real world. But different systems may represent those values differently. Hence, we need to find *semantic equivalence links* between *domain values* of equivalent attributes belonging to different systems. The definition of such equivalence links is a complex task for the following reasons:

1. The number of instances to be dealt with can be quite large. We have to analyze each attribute with respect to its equivalent attributes.

2. The interpretation of an instance attribute changes from one person to another.

Figure 11 shows an example of two attributes ($Attribute_1$ and $Attribute_2$) semantically equivalent. Each one has its own set of instances (($A_{11}, \cdots, A_{1n}$) and ($A_{21}, \cdots, A_{2m}$)). Is it possible to find a link between the content of these instance sets? Four cases are possible as illustrated by Table 2.

Table 2. Cases of semantic equivalencies at the instance level

Attributes		Syntactical Form	Semantic Interpretation
A_{1i}	A_{2j}	Same	Same
A_{1i}	A_{2j}	Same	Different
A_{1i}	A_{2j}	Different	Same
A_{1i}	A_{2j}	Different	Different

Figure 11. Equivalence management at the instance level of attributes

Independently from the syntactical form, we have to determine, for each instance A_{1i}, all the instances that have the same semantic interpretation. An element to consider in this global query-refinement step is the categorization of the attributes with respect to the following criteria:

1. For certain attributes, it is relevant to consider semantic equivalencies at the instance level. Example : ..._description?, ..._content?, etc.

2. For other attributes, it is not relevant to consider semantic equivalencies at the instance level. Example : ..._name?, ..._address?, etc. If the user decides to instantiate the attribute ..._name? with the value *Jean*, it is not feasible to find an equivalence for this value.

This categorization reduces the number of attributes to be considered. However, it only partially solves the equivalence problem at the *instance level*. As a solution, we suggest to use an approach similar to that proposed for the selection of attributes by the user (cf. Section 4.2.1). When an attribute is instantiated, values to be applied to the new attributes can be determined as follows[4]:

1. If the number of instances of the new attribute is inferior to a given threshold (it can be fixed by the designer), the instance list is sent to the user who is asked to choose the relevant values.

2. If the new attribute is described on how we can instantiate it, then the user is prompted to use this description.

3. If the number of instances of the new attribute is superior to a given threshold, then the user is prompted to take the default value.

4.2.3. Determination of Join Links Once the problem of restriction values is resolved, we must specify *join mechanisms*. They allow to link all the attributes (including their equivalent attributes) that constitute the global query and belong to the same GDL.

Traditional languages used in query formulation, such as SQL, require the user to *explicitly* indicate the links (join conditions) that can exist between the relevant data objects (entity or relationship). Using the Base Ontology, these links are obtained automatically. Attributes and entities/relationships which are not initially integrated in the global query, but are necessary for join operations are added in order to complete the query. We use Cheung's work [2, 3] in order to design a *detection algorithm for join links*. The goal of the algorithm (cf. Annex) is to find a path that links all the needed data objects. A GDL data model can be viewed as a *connected graph*. Its components are obtained from the meta-concepts *Attribute*, *Entity*, *Relationship*, etc. contained in the Ontology Base (Fig. 10). Entities and relationships that contain the attributes in sets P and C of the global query are the *nodes to connect*.

The principle behind the algorithm consists of repetitively sending messages from a node to its adjacent nodes (those that are directly connected). A message is structured as follows: the identifier of the message[5] (id_message); the name of the transmitter (name_transmitter); and the name of the receptor (name_receptor). At the beginning, the number of messages to generate is equal to the number of nodes to connect. Moreover, the value Lambda (Λ) is assigned to the field transmitter (name_transmitter $\leftarrow \Lambda$). Since entity and relationship labels are unique in a data model, we can identify messages by these labels. When a message is sent, the value of the field transmitter takes the value of the receptor. Hence, the value of the field receptor takes the value of the new receptor.

When the termination tests are verified (cf. Annex), we process messages in order to find the solution path (starting from the receptor to the transmitter) until the value of the transmitter is equal to Λ. After obtaining the messages that constitute the path, we use the commutative relations to indicate that a link from node n_i to node n_j is the same as the link from node n_j to node n_i. In our example, the attribute to find is map_id and the condition attributes are map_scale, map_content and support_type, providing the selection criteria. We suppose that attribute support_type is localized in Entity_{12}, while the other attributes map_id, map_content, and map_scale are localized in Entity_{11}. There is a problem with attribute support_type. In that case, the system has to find the join links between these two entities Entity_{11} and Entity_{12} using the detection algorithm.

4.2.4. Data Requests Execution Once all the previous steps are accomplished, the obtained information (i.e., attributes to get, selection criteria and join links) are sent to the local-source framework of each GDL that belongs to the set A. Knowledge-Agents of these local-source frameworks receive this information through the Interaction-Agents in order to generate the appropriate sub-queries.

In our example, the needed information are sent to the local-source framework of GDL_1. Taking into account the characteristics of the GDL's data management system, the Knowledge-Agent accesses GDL_1 and executes the sub-query (with respect to the *autonomy criteria*). Once this sub-query is processed, the Knowledge-Agent sends the results back to the Resolution-Agent of the client framework, in order to provide the user with answers.

5. An Implementation of the SIGAL Environment

The SIGAL architecture and the global query process described in this paper have been implemented through the SIGAL's prototype which is still under improvement. This prototype uses the *JAVA* language (particularly, its applet mechanisms) to specify framework functionalities, *VisiBroker for JAVA* Object Request Broker (ORB) to manage communication between frameworks, and a JDBC driver [13] to connect the SIGAL's available informational resource (Ontology Base and GDL's database).

The following points are the relevant aspects of the SIGAL's prototype (Fig. 12):

Figure 12. Operating mechanisms of the SIGAL's prototype

- The client applet is the client framework and runs on the user computer. This applet integrates a JDBC driver, is able to invoke the server framework services and sends user's requests to the local-source frameworks, after their specifications by the server framework has been accomplished.
- The server framework is developed on a Web site and enhanced with a JDBC driver.
- VisiBroker for JAVA is used to establish communications between the client applets, the server frameworks, and the local-source frameworks.
- Microsoft's Access database system is used to manage the Ontology Base.
- All the agents are implemented as Java classes.

Figure 12 illustrates the three types of implemented frameworks. They compose a distributed system. Hence, these frameworks need to communicate in order to exchange information about users' needs, GDL's data, etc. Each exchange operation[6] is associated to a specific object called *Interface*[7]. The file called "Sigal.idl" shows the instructions needed to the communication between the client and the server frameworks. The file is written in

the OMG's *Interface Definition Language* (IDL).

File Sigal.idl

```
module Sigal {
        interface Server {
            string locate();};
        interface ServerFramework{
            Server find(in string user_need);};};
```

When the file Sigal.idl is parsed by the *idl2java* precompiler, we obtain what is called the *client-stub* and the *skeleton-server*. The role of these two programs is to allow programmers to specify all the communication aspects in an abstract way. Such aspects, How to connect a server? How to send and get back information? How to detect failure connections? etc. are handled by the communication protocol VisiBroker for Java. Once the needed client-stub and the skeleton-server are obtained, we respectively link them to the client and server programs that we have developed.

In the following points, we give some examples in order to illustrate how we specify software agent-oriented framework's concepts in the Java language.

1. All the framework types are declared as Java classes.
   Example: the server framework

```
public class ServerFramework
                extends Sigal._sk_ServerFramework {
    ServerFramework(String server_name_object) {
        // class constructor
        super(String server_name_object);}}
```

2. All the needed agents of a framework are declared as Java classes.
   Example: the Domain-Agent of the server framework

```
public class Domain-Agent extends Agent{
    // variables declaration
    String information_return;
    // methods declaration
    // an access method to a database
    String access(String _url, String _request){
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        _connect = DriverManager.getConnection(_url);
        _state = _connect.createStatement();
        _result = _state.executeQuery(_request);
        _result.next();
        return information_return;}}
```

3. All the communication between frameworks are handled by the VisiBroker for Java.
   Example: when the client framework requests the local-source framework, the client framework has to bind it first before sending the relevant information.

```
try{
    // ORB initialization
    CORBA.ORB orb = CORBA.ORB.init(this);
    // locate an object of the local-source framework
```

```
        _source_manager = Source.SourceFramework_var.bind("NAME");
        // processing
        ... }
    catch(CORBA.SystemException ex){
        ex.printStackTrace();
        System.out.println(ex);}
```

6. Conclusion

In this paper, we presented the major characteristics of a software agent-oriented framework and its application to the SIGAL project. A framework is made up of teams of software agents, that are able to fulfill services offered to users by the framework. These agent teams can be set up in a unique framework or in several frameworks leading, in the latter case, to the creation of a multiframework environment.

Furthermore, we presented the main steps and characteristics of the global query processing in the SIGAL environment. A global query characterizes the needs of a user who looks for information from several distributed and heterogeneous GDLs. We proposed the autonomy criteria and the complementary criteria, in order to identify the nature of sub-queries to be obtained from the decomposition of the global query. We also proposed some solutions to help a user in the formulation of his global query and the resolution of knowledge semantic disparities.

We conclude the paper by evaluating the global-query processing approach, according to the following points:

1. Query formulation: the user does not need to worry about the type of knowledge to use. He instantiates the formulation patterns and uses the knowledge of the Ontology Base.

2. Approach flexibility: the global-query approach does not depend on the availability or not of a specific GDL. If a GDL is added or removed from the SIGAL interoperability environment, the approach does not have to be adapted. The only thing to adapt is the Ontology Base.

Appendix, Detection Algorithm for Join Links

Let

Q_n	:	the set of nodes that have transmitted a message to the node n;
N	:	the set of nodes to visit in the next iteration;
O	:	the objective set; it indicates the nodes initially to be connected;
M	:	the set of all received messages;
R	:	the set that indicates the common nodes;
R_n	:	the set of the nodes that have received a message from the node n;
V	:	the set of the neighbor nodes;
C	:	the set that receives the structures (id,transmitter,receptor) after the processing in the opposite side of the set M;
val	:	termination value;

```
find-link-objects (O)
Begin
|    /* initialization phase */
```

```
|   N = O
|   R = φ
|   C = φ
|   val = 0
|   ∀ n ∈ N
|   Do Begin
|       |   create-message(n,Λ,n)
|       End
|   R = ∪R_n
|
|   /* processing phase */
|   While (||R|| ≠ val) ∧ (O ≠ φ)
|   Do Begin
|       |   R = φ
|       |   ∀ n ∈ N
|       |   Do Begin
|       |       |   R_n = φ
|       |       |   V = find-neighbors(n) /* access the Ontology Base*/
|       |       |   ∀ n' ∈ (V - C)
|       |       |   Do Begin
|       |       |       |   ∀ id ∈ Q_n
|       |       |       |   Do Begin
|       |       |       |       |   create-message(id,n,n')
|       |       |       |       End
|       |       |       End
|       |       End
|       |   If ∩R_n ≠ φ
|       |   Then Begin
|       |       |   R = ∩R_n
|       |       |   val = ||R||
|       |       End
|       |   Else Begin
|       |       |   R = ∪R_n
|       |       End
|       |   N = ∪R_n - (O ∩ (∪R_n))
|       |   O = O - (O ∩ (∪R_n))
|       |   C = C ∪ (O ∩ (∪R_n))
|       End
|
|   /* path detection phase */
|   If C ≠ φ
|   Then Begin
|       |   ∀ n ∈ C
|       |   Do begin
|       |       |   find-path(n)
|       |       |   path_final = path_final ∪ path_n
|       |       End
|       End
|   If (||R|| == val) ∧ (O ≠ φ)
|   Then Begin
|       |   If R ∩ C ≠ φ
|       |   Then Begin
|       |       |   R = R - C
|       |       End
|       |   /* select one element from R */
|       |   find-path(n)
|       |   path_final = path_final ∪ path_n
```

```
|           End
End

create-message(id, transmitter, receptor)
Begin
|   create m
|   m.id = id
|   m.transmitter = transmitter
|   m.receptor = receptor
|   If id ∉ Q_receptor
|   Then Begin
|           |   Q_receptor = Q_receptor ∪ id
|           |   M = M ∪ m
|           |   If transmitter = Λ
|           |   Then Begin
|           |           |   transmitter = receptor
|           |           End
|           |   R_transmitter = R_transmitter ∪ receptor
|           End
End

find-path(x)
Begin
|   path_n = φ
|   ∀ id ∈ Q_x
|   Do Begin
|           |   m = find-path(id,?,x)
|           |   While m.transmitter ≠ Λ
|           |   Do Begin
|           |           |   path_x = path_x ∪ (m.id,m.transmitter,m.receptor)
|           |           |   m = find-message(id,?,m.transmitter)
|           |           End
|           End
End
```

Termination tests The termination criteria of the detection algorithm are: finding one or several common elements to the nodes to be connected **or** when messages arrive to the nodes that have initially sent messages.

Notes

1. SIGAL stands for "Système d'Information Géographique et Agent Logiciel" (Geographic Information System and Software Agent).
2. Currently, we do not treat the redundant results from GDLs. However, it can be considered in the future.
3. To simplify the text and to facilitate comprehension, we use the term attribute instead of meta-instance.
4. Initially, the value chosen by the user is always used by default.
5. The message identifier is used to avoid an infinite loop; i.e., a node sending again the same message to its previous transmitter.
6. We have two types of exchanges: (client → server and server → client) and (client → local-source and local-source → client).
7. Each interface corresponds to a Java class.

References

1. G. Babin and C. Hsu. Decomposition of knowledge for concurrent processing. *IEEE Transactions on Knowledge and Data Engineering*, pages 758–771, October 1996.
2. W. Cheung. *The Model Assisted Global Query-System*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, November 1991.
3. W. Cheung and C. Hsu. The model-assisted global query system for multiple databases in distributed entreprises. *ACM Transactions on Informations Systems*, 14(4):421–470, October 1996.
4. M.R. Genesereth and A.P. Ketchpel. Software agents. *Communication of the ACM*, 37(7):48–53, July 1994.
5. C. Hsu. *Enterprise Integration and Modeling — the Metadatabase Approach*. Kluwer Academic Publisher, Boston, Mass., USA, 1996.
6. L. Liu and C. Pu. Metadata in the interoperation of heterogeneous data sources. In *Proceedings of the First IEEE Metadata Conference*, April 16-18 1996.
7. Z. Maamar. *Contribution à la résolution des problèmes d'interopérabilité des systèmes, une méthode de conception par frameworks orientés-agents logiciels*. PhD thesis, Computer Science Department, Laval University. Québec, Canada, October 1997.
8. P. Maes. Agents that reduce work and information overload. *Communication of the ACM*, 37(7):31–40, July 1994.
9. B. Moulin and B. Chaib-draa. An overview of distributed artificial intelligence. In Wiley, editor, *Foundations of Distributed Artificial Intelligence*, pages 3–55. G.M.P. O'Hare and N.R. Jennings editors, 1996.
10. H.-S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):1–40, September 1996.
11. OMG. Object Management Group, http://www.omg.org.
12. R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1996.
13. P. Patel and K. Moss. *Java Database Programming with JDBC*. Coriolos Group Books, 1996.
14. M.-J. Proulx, Y. Bédard, F. Létourneau, and C. Martel. Catalogage des données spatiales sur le world wide web: Concepts, analyse des sites et présentation d'un géorépertoire personnalisable georep. *Revue Internationale de Géomatique*, 7(1), 1997.
15. G. Rumbaugh, J. Blaha, W. Premelani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
16. K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert-Intelligent Systems and Their Applications*, 11(6):36–45, July 1996. http://www.cs.cmu.edu/.