Using Api-Calculus for Formal Modeling of SDIAgent: A Multi-Agent Distributed Geospatial Data Integration System

Shahram Rahimi
Department of Computer Science
Southern Illinois University
Mailcode 4511, Carbondale, IL 62901, USA
rahimi@cs.siu.edu

ABSTRACT This paper presents a formal description of SDIAgent, a novel agent-based distributed geospatial-data integration system using Api-calculus. Api-calculus, an extension to Pi-Calculus, provides a suitable basis for formally defining the behavior of the components of distributed systems which contain mobile processes. It is not our intention at this stage to establish correctness of the components or the completeness of the model. These will be addressed later using existing tools and by developing new ones.

KEYWORDS: GIS, Multi-Agents, Geospatial data, Api-calculus, Pi-calculus, Formal Modeling, SDIAgent.
1. Introduction

During the past two decades, computer architecture has moved from stand-alone systems to local and wide-area networks. A natural extension to this development and to the increasing needs for geographic information has been the study of the effects of distributed technology on geographic information. When geographic information is shared in a distributed environment, the important technical issues such as data integrity, edge matching, conflation, integration, and generalization are still relevant. In particular, the issues associated with conflation have serious consequences for many applications. The term “conflation” is used to refer to the integration of data from different sources. Without the ability to conflate/integrate data from different sources, users are faced with extensive duplication of effort and unnecessary cost. When geospatial data from different sources are combined, it is quite a challenge to preserve the semantics inherent in the component data sets.

The objective of this paper is to present an Api-calculus (Rahimi et al. 2002a) model for SDIAgent, an autonomous multi-agent system that retrieves, filters, and integrates/conflates geospatial data from multiple heterogeneous sources. This system is introduced in (Rahimi et al. 2002b). In this system, it is assumed that the data can be stored in a number of formats, the geospatial databases can be relational as well as object-oriented, different software vendors can be the sources of the database environments, and the data sources are distributed.

In design of SDIAgent, software agent technology was selected as the implementation paradigm, mainly because of the distributed and complex nature of the problem (Rahimi et al. 2002b). A software agent, which has been around for approximately twenty five years, is a software program that performs tasks for its user within a computing environment. There are distinct characteristics that collectively constitute a software agent. Software agents, or agents for short, are differentiated from other applications by their added dimensions of mobility, autonomy, and the ability to interact independent of its user's presence. The ability to move processing to the source of activity, i.e., a database server, reduces the network overhead (Rahimi et al. 2003). Moreover, multiple agents can simultaneously process information stored in multiple data locations. Such agents can communicate and cross-reference distributed data to support the distributed conflation process. The ability to deploy software to a remote site in an unobtrusive manner extends the functionality of the local server and allows for utilization of additional available resources.

Beyond the facilitation and basic engineering challenge in design and implementation of agent-based systems, there are several security, validation, verification, and performance related questions that need to be addressed (Serugendo et al. 1998). In order to obtain a conclusive
answer to these and many other questions, it is necessary to employ formal modeling tools useful for specifying and verifying such systems.

In this article, Api-calculus is used for formal modeling of SDIAgent. Api-calculus is an extension to pi-calculus, and is formulated out of the need for expressing mobile agent-based infrastructures in a precise manner, for experimenting with different underlying algorithms, and for reasoning about them. Api-calculus is capable of addressing intelligence, natural grouping and migration aspects of software agents (Rahimi et al. 2002a). Moreover, it has the potential of tackling the security issues of agent-based systems.

For the remaining parts of the paper, we start with a brief overview of the agent technology and SDIAgent architecture. Then, a summary of the Api-calculus is given. Next, the Api-calculus formal model for SDIAgent is presented. Finally, a conclusion section wraps up the article.

2. Software Agent Technology

Software agent can loosely be defined as intelligent programs that have some degree of autonomy (Rahimi et al. 2001a). Agents may be permanently fixed on a given host (stationary agents), or they may be capable of moving from host to host (mobile agents) in order to perform a given task or series of tasks. The autonomous, and perhaps even disconnected, nature of agents enables operation for a designated task or series of tasks.

Mobile agents offer several unique advantages over stationary programs (Kotz et al. 1997). First, the ability to move to the source of activity, i.e. a database server, reduces the additional network overhead involved in remote communication. For instance, by executing a series of database queries locally, intermediate results are not required to be transmitted to the remote system; rather, transmission is delayed until the final query has been executed, thus reducing overall execution time and required bandwidth (Rahimi et al. 2001b). In conjunction with mobility is the ability to easily deploy software to a remote site. This merely involves instructing the agent to move to the remote site and begin execution. In this manner, a server’s functionality may be extended easily and unobtrusively. Another advantage is the autonomous nature of agents. Once an agent has moved out of the client machine to another host, the client machine may safely shut down. Upon restarting, the client machine needs merely to re-establish contact to the mobile agent to regain control. Alternatively, an agent may be programmed to periodically attempt to return to its originating site.
Because of the above and many other advantages (Rahimi et al. 2001b), the software agent technology is utilized for SDIAgent the DGDC system.

3. SDIAgent System Architecture

SDIAgent’s general system architecture (Rahimi et al. 2002b) consists of a primary, centralized database and multiple, heterogeneous “feeder” databases. It is presumed that the feeder databases are under the control of trusted data collectors, and that all updates to the centralized database will be derived only from these trusted feeder databases. Within this overall architecture, there exist multiple agent classes, both stationary and mobile, and including both intelligent agents, and less intelligent software agents. We refer the reader to Figure 1 for the following discussion of the agent system.

Figure 1 depicts the general system architecture. Before processing to the system operation, we will list the types of agents (together with a brief description of each) that are currently used in the system. We begin with the agents that reside in the centralized database:

- **ROI Agents (RA):** Each RA is responsible for managing updates for a particular Region of Interest (ROI). For instance, the United Nations (UN) divides the world into 10 such regions. In our design, we follow this strategy. RAs are static, remaining on the central database during the entire process.

- **Queue Manager (QM):** The QM is responsible for supervising a priority queue of updates generated by the RAs. The QM has two roles. First, in case of an emergency, such as a natural disaster, or political/military action, it will assure that the pertinent information is promptly updated. Second, in the case of standard operational mode, it assures that all information sooner-or-later becomes updated (no starvation).

- **Conflation Manager (CM):** This agent is a static agent, which is located in the central database and is responsible for generating a conflation agent for each update request entry in the queue and initiating the conflation process.

- **Conflation Agents (CA):** Each CA, generated by the CM, is responsible for a single update request. The CA is a superclass of many specialized agent classes that have extensive knowledge about their domain relevant to the conflation process (see Figure 2). CAs are intelligent mobile agents, traveling to the feeder databases to perform conflation in a Round Robin fashion.
Figure 1. Overall scheme of the system.

- **Query Agents (QA):** These are released from the central database and by the conflation agents to gather information for conflation-related queries. Prior to the CA, they arrive in all pertinent databases (for each request a subset of all feeder databases), perform initial queries and post the result to be used in the conflation process by the CA. The query agent cooperates with the wrapper agent (below) to translate the data to the common data format of the system.

Next, we list the agents that reside on the feeder databases:

- **Change Detection Agents (CDA):** Relatively small, unintelligent agents that simply log changes to the database, query for necessary information and interact with the wrapper agents to create an update object for transfer to the region of interest agent (RA) on the central database.

- **Wrapper Agents (WA):** These agents are responsible for translating different geospatial data formats to the common data model of the system. At this time, we have two versions of WAs, for VPF (Vector Product Format) and for ARCGIS Map data model.
3.1 The Behavior of the System - Autonomous Update Scenario

We now describe a typical scenario that utilizes the above-named agents for autonomous management of geospatial data. Before we proceed, we assume that at a given time the data in the feeder databases is almost synchronized with the central database. This means that only a very small amount of new information is available in the feeders. Suppose, from Figure 1, that an update to an attribute value of a spatial feature is performed on database 1 (DB1). The resident CDA notes the change and queries the database for the information needed for an update object, namely, the database identifier, the feature ID and type information, bounding box coordinates, and type of update performed (metadata, attribute, topology, geometry or unknown). This information is given to the WA to create the update object, which is then sent to the proper RA on the centralized database (flows 1, Figure 1). At this moment, the WA can self-suspend, while the CDA returns to watch the database updates.

When the update object arrives at the central database, it is placed in a special data structure known as a ROI-tree. The ROI-tree is a quad-tree that contains predefined region objects represented by the RAs. For more information on the topic of spatial indexing, the reader is referred to (Samet 1989). Each RA manages updates for a particular area of the earth, represented by bounding box coordinates, and each RA has knowledge of all feeder databases that contain information within its domain. The update object is placed in the ROI-tree at a place under the domain of the RA for the region in which the update occurred. In our approach, as a default, we pursue 10 regional divisions of the world; however, other spatial arrangements are also possible. The hierarchical concept of sub-regions is also supported, and users can define their own regions of interest.
When the timing/frequency of updates is considered, users are allowed to set priorities for selected regions and sub-regions, or may use system-generated defaults. These specified priorities dictate the rate at which the gathered updates must be conflated and incorporated into the centralized database. As an example, an update for a low-priority region may be allowed to remain in the tree for several hours or even days, while high-priority region updates may be processed several times an hour, or even immediately upon arrival.

Whenever the priority scheme of the supervising RA determines that it is time to check for updates, a traversal of the sub-tree below the RA is performed, and all the update objects are placed in a conflation queue in order of their respective priorities. The QM is responsible for supervising this priority queue. The QM has two main responsibilities; first, for standard operation, it assures that all information sooner-or-later becomes updates, and no starvation happens while the priorities are considered. Second in emergency cases, it will assure that the pertinent information is promptly updated by moving its request to the top of the queue.

The CM agent removes the highest priority update request from the queue and initiates the conflation process by generating an appropriate CA, based on the information in the update request object related to the feature type (point/line/area) and its class (e.g., building, railroad, vegetation). The CM agent also directly accesses the ROI-tree to retrieve other required information to complete the CA generation process. Now, the CA is ready to carry out the conflation process on the object. The CA uses its specialized knowledge to generate QAs to retrieve all available data needed for performing conflation on that object. The QAs are sent in parallel to all the feeder databases in which a potential conflict with the update exists (flow2, Figure 1), and begin the data extraction process. The QA communicates with the wrapper agent to translate the data to the common data format of the system. After this step, the data in all the participating databases are ready for CA to start the conflation process.

The CA first moves to the database from which the update originated (flow 3, Figure 1). The QA then passes the already-collected/prepared conflation data to it. The CA traverses all of the relevant databases, collecting the information and executing the knowledge-based conflation algorithm described in the following section (flows 4 and 5, Figure 1). It assembles the conflated data in a Round Robin fashion and then brings the results back to the central database for updates (flow 6, Figure 1). For more information on the architecture and the behavior of the system, please refer to (Rahimi 2002b).

The task of the remainder of this paper is to formalize the above in terms of the Api-calculus, but before that, we give an overview on Api-calculus basics.
4. API-Calculus Basics

In this section, an overview of the syntax and semantics of Api-calculus is given. Api-calculus, introduced in (Rahimi 2002a), is a way of describing and analyzing systems consisting of agents that interact among each other and whose configuration or neighborhood is continually changing. It is an extension to Pi-calculus which introduces three new concepts over higher order Pi-calculus (Sangiorgi 1993). These new concepts address the intelligence, natural grouping and security aspects of mobile agents.

Api-calculus introduces \textit{knowledge unit}. A knowledge unit consists of a knowledge base and a set of facts. Agents have the capability to add/drop facts to/from the fact list and modify the knowledge base by adding new rules or eliminating existing ones. Each mobile agent is capable of carrying one or more knowledge units and sending and receiving them to/from other agents. A knowledge unit also may be defined to be a different knowledge facility, for instance, a neural network.

Api-calculus also introduces the notion of \textit{term}. A term is consisted of a name or a function, where a name can be a channel, a variable, or a rule/fact (this can be different for other knowledge facilities, for example in case of neural networks, this may be a learning element). In the standard Pi-calculus names are the only terms.

Moreover, Api-calculus introduces \textit{milieu}, a new level of abstraction, that is in-between single mobile agents and the system as a whole. Milieu focuses on families of processes. Thus, it addresses related questions such as, how we can specify a collection (family) of mobile agents working towards some common goal. Milieu is a closed environment that may consist of none or many agents or other milieus that cooperate to solve a computational problem.

The rest of this section provides summarized descriptions of the elements of Api-calculus. For more details on Api-calculus theory, please refer to (Rahimi et al. 2002a).

4.1 Term

A term can be a name or a function:

- A name can be a channel, a variable name, or a fact/rule
- A term may be a function. A function may have \( l \) parameters. \( f \) ranges over the functions of \( \Phi \) and one matches the arity of \( f \).
Term Definition:

\[ R, T \equiv \{ \text{term} \} \]
\[ x, y, z, \ldots \] (name)
\[ a, b, c, \ldots \] (functions)
\[ f(x, y, z, \ldots) \] (functions)

We use \( \bar{T} \) to abbreviate tuple \( T_1, T_2, \ldots T_l \).

4.2 Process

Api-calculus allows processes to be passed as terms in a communication (as in higher order Pi-calculus). After a process has been transmitted, it can begin its execution. This is a process-passing mechanism.

Process Definition:

\[ P \equiv \begin{align*}
0 & \quad \text{(no action)} \\
\alpha.P & \quad \text{(action prefix)} \\
P_1 + P_2 & \quad \text{(summation process)} \\
[T = R]P_1 : P_2 & \quad \text{(conditional process)} \\
\nu x P & \quad \text{(name restriction)} \\
(K_i)P & \quad \text{(knowledge name restriction)} \\
!P & \quad \text{(replication)} \\
D_{L} & \quad \text{(constant)}
\end{align*} \]

Letters \( P, P_1, P_2 \ldots \) and \( Q, Q_1, Q_2 \ldots \) are used to denote processes. 0 is the no action process. This is the process that does internal computations. \( \alpha \) is called an action prefix (definition V). The expression \( \alpha.P \) performs the action \( \alpha \) and then behaves like \( P \). The summation process \( P_1 + P_2 \) acts like either \( P_1 \) or \( P_2 \). \( [T = R]P_1 : P_2 \) is a conditional process, but we should stress that \( T = R \) represents equality of terms \( T \) and \( R \), rather than strict syntactic identity. We abbreviate it to \( [T = R] : P_1 \) when \( P_2 \) is 0. The expression \( \nu x P \) makes a new, private name \( x \) (local) then behaves as \( P \)
process. \((K_i)P\) indicates that \(K_i\) is a knowledge unit name local (restricted) to process \(P\), which means that we may have more than one \(K_i\) in a multi-agent system (more in examples). !P is the replicated processes (means \(P \parallel P \parallel \ldots\)). For instance \(P_1 \parallel P_2\) consists of \(P_1\) and \(P_2\) acting in parallel. The components may act independently; also, an output action of \(P_1\) at any output port \(x\) may synchronize with an input action of \(P_2\) at \(x\), to create a silent, \(t\), action of the composite agent \(P_1 \parallel P_2\).

\(X\) is a process variable, \(\vec{L}\) stands for any tuple of processes or terms (i.e. \(L_i\) is a term or a process), and \(\vec{T}\) stands for any tuple of terms. The constant \(D\) is defined as \(D = (\vec{T})P\). Constants are to be seen as functions whose parameters can be processes or other functions. For example, consider \(x(L)\) to be an input prefix which receives a term or a process \(L\) from channel \(x\) and \(\vec{x}L\) to be an output prefix that sends a term or a process \(L\) through channel \(x\). Now in \(\vec{x}P.Q \mid x(X).X\), once the interaction between the two processes has taken place, the resulting process is \(Q \parallel P\). Indeed, process \(x(X).X\) was waiting for \(X\) to be sent along channel \(x\), i.e., it was waiting for a process \(X\) defining its subsequent behavior.

If \(x : s \mapsto (s_1, s_2)\), then for \(x(\vec{T}).P\) and \(\vec{x}T.P\) (the first one sends term tuple \(\vec{T}\) through channel \(x\) and the second one receives \(\vec{T}\) from channel \(x\); more later) to respect \(Sf\) (sorting function, explained above), it must be that \(\vec{T} = T_1, T_2\), for some \(T_1 : s_1\) and \(T_2 : s_2\). Moreover in a matching \([T = R]\) we require that the tested terms \(T\) and \(R\) belong to the same sort. We also assign an object sort to agents: processes take the sort (\(\) ), whereas if \(D = (\vec{T})P\) and \(\vec{T} : \vec{s}\), then \(D\) and \((\vec{T})P\) take the sort(\(\vec{s}\)). Now the requirement on \(D(\vec{R})\) is that \(\vec{s}\) exists for instance \(\vec{R} : \vec{s}\) and \(D : (\vec{s})\) (Sangiorgi 1993). Before we talk about the actions, \(\alpha\), we need to define the concept of knowledge unit.

### 4.3 Knowledge Unit

A knowledge unit can be any knowledge facility, but here we consider it to be a knowledge base (set of rules) and a set of facts. A knowledge unit reacts to any new fact(s) added to its facts list. \(K_i\), \(K_2\), \ldots represents knowledge units. \(K\) denotes the set of knowledge units that belong to process \(i(P_i)\). Here is the grammar of knowledge units:

\[
K = \begin{cases} 
0 & \text{(empty knowledge unit)} \\
| r & \text{(a single rule, or a single quantity)} \\
| K_1 + K_2 & \text{(knowledge units summation)} 
\end{cases}
\]
0 is called an empty knowledge unit. An empty knowledge unit is produced if all the rules and facts are deleted from it. A knowledge unit may consist of a single rule \((r)\). Expression \(K_1 + K_2\) is called a knowledge unit summation, which means that both of the knowledge units react to a fact at the same time. This means that \(K_1\) and \(K_2\) join and act as a single knowledge unit. We use variables \(U\) and \(V\) to range over terms and knowledge units. Also \(L_1, L_2, \ldots\) represent processes, terms, and knowledge units.

4.4 Actions
In addition to send and receive actions in Pi-calculus, Api-calculus includes knowledge and milieus actions. Moreover, instead of just having ‘names’ (as in Pi-calculus), we have terms and processes in actions. Knowledge actions consist of knowledge unit calls, receiving knowledge units, sending knowledge units and adding/dropping facts and rules to/from a knowledge unit. Later, we will see that milieus’ actions includes join and leave milieus. Letters \(\alpha_1, \alpha_2, \ldots\) are used to represent action prefixes. Let \(A\) denote the set of all actions in the calculus:

- \(\tau\) is a no external, but an internal action.
- \(x(L)\) is an input prefix. Variable \(x\) stands for a name of an input port (channel) of a process which contains it; \(L\) stands for any tuple of processes, terms, or knowledge units. \(x(L).P\) inputs arbitrary terms or processes \(L\) at the port \(x\) and then behaves like \(P\{L_1 / L\}\). All free occurrences of the names \(L\) in \(P\) are bound by the input action prefix \(x(L)\) in \(P\).
- \(\pi L\) is an output prefix. A name \(x\) is thought of as an output port of a process which contains it; \(\pi L.P\) outputs the tuple of terms or processes \(L\) at the port \(x\) and then behaves like \(P\).
- \(K_i\langle\bar{a}\rangle (\bar{R})\) is a knowledge unit call. Expression \(K_i\langle\bar{a}\rangle (\bar{R}).P\) calls the knowledge unit, \(K_i\), passing a list of facts, \(\bar{a}\). The result of this call is placed in \(\bar{R}\). All free occurrences of \(\bar{R}\) in \(P\) are bound by the prefix \(K_i\langle\bar{a}\rangle (\bar{R})\) in \(P\).
- \(K_i(\bar{a})\) is a prefix that adds tuple \(\bar{a}\) to the facts list of \(K_i\) if \(\bar{a}\) is tuple of facts, or to the rule list of \(K_i\) if \(\bar{a}\) is a tuple of rules. Expression \(K_i(\bar{a}).P\) adds \(\bar{a}\) to the facts list or rule base of \(K_i\) and then acts like \(P\).
• $K_a$ is a prefix which drops $a$ from the facts list (if $a$ is a fact) or from the rule base (if $a$ is a rule). Expression $K_a.P$ drops $a$ from the facts list or the rule base of $K$, and then acts like $P$.

• $join \ m.P$ makes process $P$ to join milieu $m$ (a closed environment, definition VI) and then acts like $P$ inside of the milieu $m$.

• $leave \ m.P$ makes process $P$ to leave milieu $m$ and then acts like $P$ outside of milieu $m$.

4.5 Milieu
The existence of separate locations is represented by a topology of boundaries. A milieu is an environment (a bounded place) in which processes live and computations take place.

A milieu is surrounded by a border, which needs to be passed to join or leave it. A whole milieu can move together with its whole content (all the processes inside the milieu). The concept of milieu not only introduces a new level of abstraction to the pi-calculus, but also can be used to address the problem of the natural grouping and the security of the system. Here is the syntax of milieu:

$$M \equiv \begin{cases} 0 \\ M[O] \\ M[O_1|O_2] \\ M_1 + M_2 \\ \beta.M_1 \end{cases}$$

$$\beta \equiv join \ m \mid leave \ m \mid open$$

An empty milieu is declared as 0. Variables $O_1, O_2, \ldots, O_n$ are used to range over processes and milieus. $M[O]$ is a milieu in which process or milieu $O$ exists. A milieu may be consists of other milieus or processes acting in parallel, $M[O_1|O_2]$. Expression $M_1 + M_2$ indicates that milieu $M$ is composed from the merge of milieus $M_1$ and $M_2$. Letter $\beta$ is a milieu action prefix. Expression $\beta.M$ performs the action $\beta$ and then acts as $M$.

$M[O]$ exhibits a tree structure induced by processes and the nesting of milieu brackets, i.e. $M[P_1[P_2[M_1]\ldots\ldots[M_q]\ldots]]$. In Api-calculus, process mobility is represented as crossing of milieus’ boundaries. This is important to point out that in this calculus interaction between
processes is possible when they share location within a common boundary or outside of any boundaries.

- \( \text{join } m.M \) makes milieu \( M \) to join milieu \( m \) and then acts like \( M \) inside the milieu \( m \).
- \( \text{leave } m.M \) makes milieu \( M \) to leave milieu \( m \) and then acts like \( M \) outside the milieu \( m \).
- \( \text{open}.M \) dissolves the boundaries of milieu \( M \) and makes it to open up its boards and cease its existence. Then the processes and the milieus, inside the \( M \), act as they don’t belong to \( M \) anymore.

### 4.6 Reduction Rules

The reduction relation, \( \rightarrow \), is defined by the rules in table 1. As an example of applying these rules, let us see how to infer the reductions for the process in the following example:

\[
P = \forall x ((x(u).Q_1 + y(v).Q_2)|\overline{x}a.0) | (\overline{y}b.R_1 + \overline{x}c.R_2)
\]

First, \( (x(u).Q_1 + y(v).Q_2)|\overline{x}a \) matches the left side of the \text{REACT} rule, so we have \( (x(u).Q_1 + y(v).Q_2)|\overline{x}a \rightarrow Q_1|0 \). Then using \text{STRUCT}, \text{RES} and \text{PAR} we can infer a reduction in \( P \):

\[
(x(u).Q_1 + y(v).Q_2)|\overline{x}a \rightarrow Q_1 \quad \text{REACT}
\]

\[
\forall x ((x(u).Q_1 + y(v).Q_2)|\overline{x}a \rightarrow \forall x Q_1 \quad \text{STRUCT}
\]

\[
\forall x ((x(u).Q_1 + y(v).Q_2)|\overline{x}a) | (\overline{y}(b).R_1 + \overline{x}(c).R_2) \rightarrow \forall x Q_1 | (\overline{y}(b).R_1 + \overline{x}(c).R_2) \quad \text{PAR}.
\]

### Table 1. Reduction Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{TAU} ):</td>
<td>( \tau.P + Q \rightarrow P )</td>
</tr>
<tr>
<td>( \text{REACT} ):</td>
<td>( (x(L_1).P + P')</td>
</tr>
<tr>
<td>( \text{PAR} ):</td>
<td>( O_1 \rightarrow O'_1 ) ( O_2 \rightarrow O'_2 ) ( O_1.O_2 \rightarrow O'_1.O'_2 )</td>
</tr>
<tr>
<td>( \text{RES} ):</td>
<td>( P \rightarrow Q ) ( P \rightarrow \forall T P \rightarrow \forall T Q )</td>
</tr>
<tr>
<td>( \text{RES-K} ):</td>
<td>( P \rightarrow Q ) ( (K)P \rightarrow (K)Q )</td>
</tr>
<tr>
<td>( \text{MIL} ):</td>
<td>( P \rightarrow Q ) ( M[P] \rightarrow M[Q] )</td>
</tr>
<tr>
<td>( \text{STRUCT} ):</td>
<td>( O_1 \rightarrow O'_1 ) ( O_2 \rightarrow O'_2 ) if ( O_1 = O_2 ) and ( O'_1 = O'_2 )</td>
</tr>
</tbody>
</table>

Now, we are ready to present the formal model for SDIAgent system.
5. The Formal Model of the Agent-Based DGDI System

We now present the formal model for SDIAgent using Api-calculus. This section is divided to three subsections. The first subsection models the feeder databases (data repositories). The second one models the master (central) database, and finally the last one models the conflation process. The modeling of the whole system includes ten steps, which have divided among these three subsections. In this model, we have left out the knowledge units distribution and the reason is that their structural design is still not finalized in our architecture.

5.1 Data Repositories (feeders)

We start with the modeling of the feeder databases. Here is the list of the abbreviations we use in this subsection:

- Data Repositories (feeders): $M_{f1}, M_{f2}, \ldots, M_{fn}$
- Data Integration Agents (Wrappers): $P_{w1}, P_{w2}, \ldots, P_{wn}$
- Change Detection Agents (Watcher Agents): $P_{cd1}, P_{cd2}, \ldots, P_{cdn}$
- Input-info objects sent from $M_{fi}$: $z_{i1}, z_{i2}, \ldots, z_{il}$

**Step 1:** The data repository, $M_{fi}$, before engaging in any action:

$M_{fi}[P_{cdi} \mid P_{wi}]$

where $P_{cdi}$ is the change detection agent which is watching the database for any update.

**Step 2:** If an update happens then agent $P_{cdi}$ sends $z_{uo}$ (update object) to the wrapper agent, $P_{wi}$, through channel $x_{fi}$. Then after format adjustment, $z_{uo}$ will be sent to the appropriate region of interest agent RA:

$M_{fi}[\pi_{fr1}z_{uo}, P_{cdi}] \tau \mid x_{fr1}(z_{uo}'), \tau \gamma_{fr}z_{uo}' \mid j \mid P_{wi}]$

$\tau$ is the data integration process of the wrapper agent (an internal process).

5.2 Master Database (Central Database)

Here, the formal model of the central database is given. The following is the list of the abbreviations we use in this subsection:

- ROI agents (10 regions): $P_{R1}, \ldots, P_{R10}$
- Queue manager: $P_{qm}$
- Conflation manager: $P_{cm}$
- Conflation agents: $P_{ci}, \ldots, P_{ci}$
- Query agents: $P_{q1}, \ldots, P_{qk}$

**Step 3:** In the central database, RAs ($P_{R}$) receive update objects from the feeder databases $P_{di}$:

$$M_{\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wage...
5.3 Conflation Process

Finally the conflation process is modeled as follows:

**Step 7:** The QAs ($P_q'$) move to the feeder databases ($k$ number of QAs go to $k$ databases).

$$M_{i}[P_{R_i} | P_{em} | P_{qm} | P'_{ci}] \rightarrow M_{i}[P_{R_i} | P_{em} | P'_{ei}] | M_{j}[P_{wj} | P_{cdj} | P'_{qj}]$$

**Step 8:** The QAs ($P_q'$) query the feeder databases and then send the data to the local wrapper agents ($P_{wj}$). The wrapper agents send the integrated data back to the QAs, which in turn provide the formatted data to CA for conflation process.

$$M_{j}[P_{cj} | x_{j1}(z_{qj}^*), x_{j2}(z_{qj}^*) , P_{wj} | x_{j2}(z_{qj}^*) | P'_{qj}]$$

**Step 9:** The conflation agent ($P_{ci}'$) migrates to the feeder databases one by one, receives the formatted geospatial data from QAs and assembles the result in round-robin fashion. It executes the knowledge-based conflation algorithm described earlier.

$$M_{f1}[P_{ci} | P_{w1} | x_{j1}(z_{qj}^*)] \rightarrow M_{f1}[P_{ci} | x_{j1}(z_{qj}^*) | P'_{qj}]$$

$$M_{f2}[P_{ci} | P_{w2} | x_{j2}(z_{qj}^*)] \rightarrow M_{f2}[P_{ci} | x_{j2}(z_{qj}^*) | P'_{qj}]$$

$$;$$

$$M_{fk}[P_{ci} | P_{wk} | x_{jk}(z_{qj}^*)] \rightarrow M_{fk}[P_{ci} | x_{jk}(z_{qj}^*) | P'_{qj}]$$

CA’s $\tau$ is the actual conflation process.

**Step 10:** Finally CA brings the conflated data back to the central database and sends it to the appropriate RA for an update.
6. Conclusion

We introduced an agent-based distributed geospatial data integration system, SDIAgent, and employed the Api-calculus to produce an innate formal description for the system. Api-calculus is a powerful tool for describing and analyzing systems consisting of agents when configuration and/or neighborhoods are continually changing. We showed how easy and natural is the modeling of such a complicated agent-based system using Api-calculus.

In case of using other extensions of Pi-calculus for modeling such an architecture, we would have to consider the feeder and central databases to be processes instead of locations. Moreover, it would be difficult to model the communication mechanisms between different agents as well as representing intelligence in the system. Currently, we are in process of implementing a framework capable of analyzing the models implemented in Api-calculus (Ahmed et al. 2003). The software would include a compiler and a mechanism for evaluation of the model based on security, validation, verification, and performance. Implementation of such framework would greatly uncover the importance of Api-Calculus formal modeling, since using it would make fast prototyping possible.

SDIAgent has been implemented and the results of its preliminary evaluation look promising (Rahimi et al. 2003). SDIAgent demonstrates great advantages by decreasing the network traffic and dividing the tasks efficiently.

References


Samet, H., 1989, The Design and Analysis of Spatial Data Structures (Addison-Wesley, Reading, MA).
