A Multi-Agent System for the Integration of Distributed Environmental Information

Martin Purvis, Stephen Cranefield, Roy Ward,
Mariusz Nowostawski, Dan Carter, and Geoff Bush
Information Science Department, University of Otago
Dunedin, New Zealand

Abstract

This paper describes a multi-agent platform to be used for the integration of environmental information that may be distributed over a network. The system is designed to work as a collection of collaborating agents. Information sources are encapsulated as data source agents that accept messages in an agent communication language. Here we describe how queries can be entered into the system and information collected from multiple sources. A key component of the query module is the planner agent, which takes a query and transforms it first to a calculus, then to an algebraic expression in order to break it into subqueries which an executor agent can send to the data source agents. As part of this process, the query is translated from user level ontologies to lower level ontologies relevant to the data source agents. Query results need not be returned within an ACL message, but may instead be represented by a CORBA object reference which may be used to obtain the result set. The architecture and operation of these agent components is described and an example is presented of how environmental information can be queried.

1 Introduction

Environmental information is collected and stored in a variety of differing storage formats, media types, and organised according to differing semantics. The objective of the New Zealand Distributed Information Systems Project (NZDIS) [Purvis *et al.*, 2000a; Purvis *et al.*, 2000b] is to be able to perform integrated queries in an open distributed environment that can access a heterogeneous collection of environmental information sources. In the following sections, we present several perspectives of the NZDIS software architecture. Section 2 describes briefly the infrastructual software platform, which is based on an architecture of collaborating software

agents. Section 3 discuses the model associated with the way the agents engage in conversations, *i.e.* exchange a sequence of messages in order to carry out a transaction between them. Section 4 covers an important component of the system architecture, the Query Processing Subsystem, in more detail and discusses some of the trade-offs involved. The final section offers some concluding remarks.

The notion of agent-based software interoperability is based on the idea of a loosely-coupled collection of agents that can cooperate to achieve a common goal. Each individual agent is presumed to be a specialist for a particular task, and the expectation is that, just as is in the sphere of human engineering, complex projects can be undertaken by a collection of agents, no one of which has the capability of performing all the required tasks of the project. In addition, if the system has an open agent architecture, then individual agents can be replaced by improved models, thereby enabling the system to improve gradually, grow in scope, and generally adapt to changing circumstances. For such an approach to work so that the agents work together effectively, all agents, including those newly introduced to the system, must not only possess a common understanding of possible messages and message types, but also must have an understanding of the kinds of dialogues that can take place between two agents or among groups of agents. These dialogues frequently follow commonly occurring patterns or *conversation protocols*, and communication can be enhanced if the two participants are explicitly aware of the particular pattern in which they are engaged.

In order to understand the range of possible messages that can be received, however, an agent must also have, in addition to a common means of characterising performatives and conversation policies, a model of the application domain with which the agent is associated. Such a model, called an *ontology*, characterises the relationships and constraints associated with possible entities in the given domain. Most work on ontology representation has so far been based on first-order logic or description logics, but our approach follows a slightly different line and will be discussed below.

The variety of environmental data collections in New Zealand available for integration efforts covers an assortment of types, and includes many maps and geographically-oriented data sets that have been assembled by means of automated data acquisition techniques. A large proportion of these collections are not organised according to standard database structuring, but instead are simply available as flat files. In order to provide software technology that improves access to these various collections, we have chosen to build the NZDIS system using industry standards from the object-oriented programming community, which enables us to take advantage of existing commercial implementations of the standards and enhances the stability and maintenance prospects for significant components of the system. The adopted object-oriented standards include:

- the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) as a communications infrastructure.
- the OMG's Unified Modeling Language (UML) for representing ontologies (for describing models of both the user-level and models of data sources). The advantages of using this

approach have been presented in other publications [Purvis and Cranefield, 1999; Cranefield *et al.*, 2000]

- the Object Data Management Group's (ODMG) Object Query Language (OQL) for expressing queries.

2 Software Agent Platform

A schematic representation of the NZDIS system architecture is shown in Figure 1. All of the solid, directed lines shown in the figure represent agent messages expressed in the Foundation for Intelligent Physical Agents Agent Communication Language (FIPA ACL) [FIPA, 2000]. Not shown in this figure is the System Facilitator which maintains a directory of all agents in the system and with which each agent must register when it is incorporated into the system. A query from the user agent is passed to the Query Processing Subsystem, which endeavors to query multiple data sources in order to satisfy the user's query. In the following section, the operations of the Query Processing Subsystem are described in greater detail. A query from the user agent is passed to the Query Processing Subsystem, which endeavors to query multiple data sources in order to satisfy the user's query. In the following section, the operations of the Query Processing Subsystem are described in greater detail.

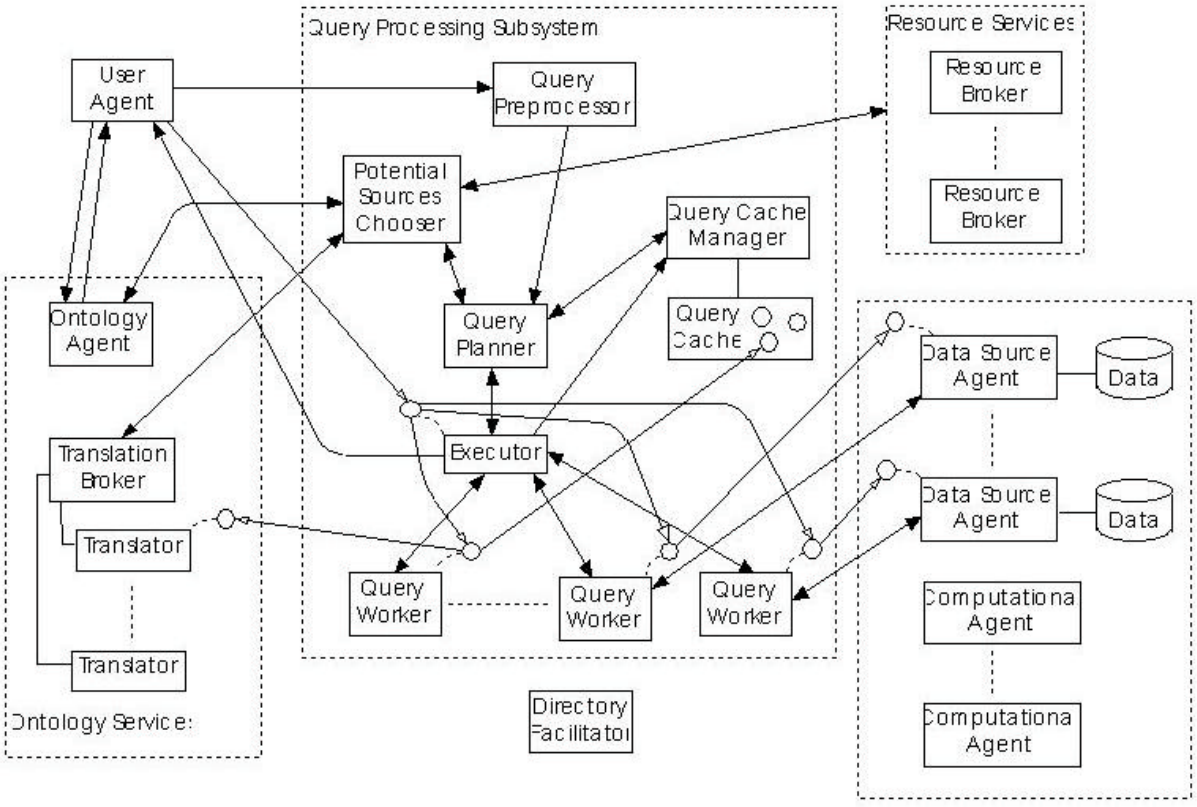

Figure 1. NZDIS multi-agent architecture for environmental queries.

2.1NZDIS agents

The NZDIS agent platform on the agent application level consists of Directory Facilitator, which is responsible for managing the address-book of all running agents, Agent Manager (Platform Monitor), which is responsible for keeping track of the agent life cycles. All agents make use of an underlying a messaging and data transfer infrastructure based on CORBA technology.

The NZDIS agent platform follows the FIPA specification. Messages are internally represented as objects. Ontologies are represented as native object structures, and we are developing a native OO representation for the ACL, which has been been proposed to FIPA. For abstract object-oriented modelling UML is being used, and for the object oriented native language representation Java has been chosen. Of course, agents implemented in different languages can cooperate inside a single NZDIS platform.

3 Agent Conversations

For agents to keep track of where they are in the context of a conversation, they must model the state of the conversation. Most work in this area has employed finite state machines as a modelling device [Bradshaw *et al.*, 1998; van der Aalst, 1998], but we prefer to use coloured Petri nets. Petri nets have advantages because of their clear graphical representation, well-defined semantics and sound mathematical formalism (that allows formal analysis and transformations). Just as they are suited for modelling and simulating workflow processes [van der Aalst, 1998], they can be used for modelling, simulating, analysing, monitoring and debugging conversations between agents in multi-agent systems.

Generally, tokens represent messages, arcs represent message passing and delivery mechanisms, and transitions represent message processing units. Roles are organized into subnets, and roles are separated by horizontal dashed lines. Arcs crossing role boundaries, i.e. arcs which cross dashed lines represent physical message passing actions (the process of sending and receiving a single message in the agent system). The arcs within roles are left up to the implementation and usually, for efficiency purposes, are implemented as method calls. This is how we have implemented it. Places represent message containers or intermediate containers and represent a message folder containing messages already processed or being processed.

There is always one initiator of a conversation, a role which starts the conversation by issuing the very first message, and this role (and only this role) always has the start place, which enables the very first transition to fire. All roles have separate dedicated terminated places, which collect the tokens when no further message processing is scheduled to occur.

A conversation is a whole Petri Net composed of a set of subnets (i.e. protocols), where at least one role has the start place (initiator) and is connected to an arbitrary number of other conversation participants. A conversation state is a current net marking. A conversation policy is encoded via arc inscriptions and guards inside particular roles, and can be changed and manipulated dynamically during the course of a `conversation.

It is natural to compose more complex conversation models out of simpler conversations or sets of protocols by connecting appropriate elements by arcs. It is important to note that complex conversations do not change the semantics of the protocols (subnets) and do not interfere with individual roles. For consistency, all basic action exchange schemas are defined via protocols, even if only a single communicative action is executed between two agents (single act without a response). That means that all communicative acts defined in an Agent Communication Language (ACL) have at least one protocol defined for them .

4 Query Processing Subsystem

4.1 Overview and components

The query processing subsystem's task is receive a query at one end, convert it into small queries that a set of data sources can handle, and then return the results of the query back to the user. The query processing subsystem consists of a planner agent, executor agent, a resource broker and several data source agents.

- It is necessary to perform some planning, since a query might require the data from several data source agents (DSAs) and have ordering constraints on how such data is accessed. Efficiency is also an important issue – there can be many orders of magnitude difference in execution time and bandwidth use between a poor plan and a good plan.

- The executor agent takes the plan and coordinates subqueries sent to the data source agents.

- The data source agents each handle information relating to a single data source. They receive a query in OQL relating to some data that they contain or wrap, transform the query into a form it can handle internally.

- The resource broker has some metadata about each of the data sources (provided by each data source) which it can make available to the potential sources chooser and the planner.

- An ontology is a description of the terms and sematics used to describe a particular domain. Our system has two sets of ontologies. There are user level ontologies that are used to formulate the query in a high level form as possible, and there are low level ontologies used by the data source agents to represent the avaialable data. The ontology translation agents converts from these high level ontologies to low level ontologies. Ontology translation is currently handled by the planner in a somewhat ad hoc manner, but it is planned to have separate ontology agents to perform this task in future.

4.2 An example query

In order to illustrate how the system works, we set up an example query :

```
Given a particular suburb and landuse, return the property IDs of
all the properties in that suburb with that landuse.
```

We use this query throughout this section to illustrate the process and issues of planning, and arrive at an XML plan that can be passed on to the executor and have a result returned. Thus we have three data sources available encapsulated inside agents used for this query:

- REINZ (Real Estate Institute of New Zealand) is a Real Estate database that contains information about properties. It is a full datanbase in that it can handle OQL queries.
- Geocode is a service that accepts an address and returns New Zealand Map Grid Coordinates. It is a lookup service -- it cannot accent coordinates and return an address.
- NZLRI (New Zealand Land Research Institute) is some ArcInfo polygon-based information that divides New Zealand into polygons and defines a land use for each polygon. The metadata item "2w 2" refers to a particular sort of flattish, high producing land used for intensive grazing or forage cropping (along with some other information too detailed to include here). We have a wrapper that can take coordinates and return a land type.

The query can be translated into OQL in the following way:

```
select distinct struct(N:p.ID)
from Property p
where p.landInfo.landUse="2w 2" and p.Suburb="Mosgiel"
```

4.3 Planning and Query Translation

The core of the planner agent is implemented in Mercury [Somogyi *et al.*, 1995]. Mercury is a high level but optimised logic/functional programming language, which combines declarative programming with advanced static analysis and error detection features. The planner agent takes a query and transforms it first to a flattened monoid comprehension as described in [Fegaras and Maier, 2000]. Ontology translation is performed on this comprehension to convert from a user level ontologies to ontologies suitable for the data source agents. The monoid comprehension is converted to an intermediate algebraic expression as described in [Fegaras and Maier, 2000]. This expression is rearranged and annotated to form another expression in a physical algebra. Finally, this is converted into an XML form to create a plan. We explore these steps in detail using our example query.

First the OQL is translated into a monoid comprehension. A monoid comprehension is more general than a set comprehension - for example, it can be used with lists and bags.

```
∪{p.ID | p ← Property, p.landInfo.landUse = "we 2".
                       p.Suburb = "Mosgiel"}
```

We use a slightly different computer-generated notation to represent this:

```
[set]{p.ID |
      p <- Property ,
      p.landUse="we 2",
      p.Suburb = "Mosgiel"
```

Flattening of the comprehension is then performed (there is no change in the example).

4.3.1 Query Translation

There are then three stages to translating a query from the user level ontology to the low level ontology. These translations are based on our UML ontologies but are currently hard-wired as a set of coded rules the planner.

1. path expansion,
2. choosing data sources (ontology translation),
3. converting to primitive types.

The first step is *path expansion*. In many cases, a user level query might have specify a field for an object that is not present in the object's class, but is nevertheless reachable by some many-to-one mappings. For example, the user might want to know the land use of a particular property which is not in the Property database, but from a property we can get the address, from the address we can get the geographic co-ordinates, from the co-ordinates we can get the land use. The path expansion phase of query translation is done entirely within the user level ontology and automates this expansion.

```
[set]{p.ID |
    p <- Property ,
    p.address.coordinates.landInfo.landUse="we 2",
    p.Suburb = "Mosgiel"
```

4.3.2 Choosing the Data Sources

The second step is *choosing data sources*. An important part of the translation is choosing which data sources will be used, and converting the query into the ontologies used by those data sources. This would ideally be done at the planning level when the query is divided into sub-queries, but this would result in inseparably mixing the translation and planning. The resource broker in interrogated as to what data source agents are available to handle subqueries, and data sources are chosen and the query translated into the appropriate ontologies. This is a non-deterministic step - if there is more than one data source that may be able to fulfil a particular function, it is quite possible that the planner will need to backtrack to this point to try other choices of data source if planning fails later. Although this step will most commonly only involve name substitution, this would be the step at which any complex translations (such as converting an age to a date) would be performed.

```
[set]{struct(N:p.ID)|
    p <- REINZ ,
    l <- NZLRI ,
    g <- Geocode ,
    g.address = p.address ,
    g.coordinates = l.coordinates ,
    g.easting = l.easting ,
    l.landUse = "we 2" ,
    p.Suburb = "Mosgiel"
```

4.3.3 Converting to primitive types

A restriction of the current system is that we can only deal with primitive types, so we need to expand address and coordinates into the primitive types of strings and numbers. This is done as a simple substitution:

```
[set]{struct(N:p.ID)|
    p <- REINZ ,
    l <- NZLRI ,
    g <- Geocode ,
    g.address = p.StreetNumName ,
    g.fine = p.Suburb ,
    g.coarse = p.zone.description ,
    g.northing = l.northing ,
    g.easting = l.easting ,
    l.landUse = "we 2" ,
    p.Suburb = "Mosgiel"
```

This ends the translation.

4.3.4 Converting to an algebra

The expression is converted to an intermediate algebraic expression as described in [Fegaras and Maier, 2000].

```
reduce[set,
  join[set,
      join[set,
          select[set,
               REINZ , p ,
               p.Suburb = "Mosgiel],
          select[set,
               NZLRI , l , true], true],
      select[set,
           Geocode , g , true],
      g.address = p.StreetNumName and
      g.fine = p.Suburb and
      g.coarse = p.zone.description and
      g.northing = l.northing and
      g.easting = l.easting and
      l.landUse = "2w 2"],
  {p.ID}, _ , true]
```

Note that there are many possible options for this depending on what order the collections and constraints appear in the monoid comprehension. For example the expression above would start by taking the cross product of all the properties in Mosgiel with the land use data for all points -

this needs rearrangement before a viable plan can be produced. It may not be possible to execute such a query, as the data sources have constraints on what sort of sub-queries they can process. Also, we can convert some of the joins to semijoins, and work out what the data flow through the plan is.

4.3.5 Metadata

Each data source agent has certain capabilities in what data it contains, and what sort of queries it can execute. In the data sources used in our example, NZLRI can only perform lookups, and REINZ is a full database that can process OQL queries. Each data source has one of more hunks of XML metadata specifying possible queries. This may be present in addition to other metadata.

```
<source name="accnz" collection="NZLRI">
  <flags>
    <flag>do_field</flag>
  </flags>
  <ops>
  </ops>
  <fields>
    <f field="northing" name="northing"
           ontology="geo"     type="northing"/>
    <f field="easting"  name="easting"
           ontology="geo"     type="easting"/>
    <f field="landUse"  name="landusestring"
           ontology="landuse" type="landusestring"/>
  </fields>
  <semijoin>
    <required>
      <f>northing</f> <f>easting</f>
    </required>
    <optional/>
    <output>
      <f>northing</f> <f>easting</f> <f>landUse</f>
    </output>
    <unique_key>
      <key>
        <f>northing</f> <f>easting</f>
     </key>
    </unique_key>
  </semijoin>
</source>
```

This example is the metadata for the "accnz" data source agent containing the NZLRI collection. It indicates that this data source can only handle semijoin queries with no extra predicates, has northing and easting as required inputs, no optional inputs and several outputs. It is essentially a

lookup table on northing and easting, with the extra information that northing and easting form a unique key.

We use the data source information, backtracking with reordering the order of joins to match the constrainsts in the metadata physical plan. We also convert joins to semijoins (as they are usually cheaper) wherever possible.

```
Executor:reduce[set,
   Executor:select[set,
       accnz:semijoin[set,
           accge:semijoin[set,
               accod:select[set,
                    REINZ , p ,
                    p.Suburb = "Mosgiel],
           Geocode , g ,
           {g.address=p.StreetNumName,
           g.fine=p.Suburb,
           g.coarse=p.zone.description}, true],
       NZLRI , l ,
       {northing=g.northing,
       l.easting=g.easting}, true],
       l.landUse = "2w 2"],
   {p.ID}, true]
```

This is effectively the plan. There is additional work to perform - this plan needs to be converted into an exact set of steps to be sent to the executor, including what the flow of data is. This process is essentially an implementation detail so not covered here.

The resulting XML plan is rather large, so we just present a single node of it for illustration.

```
  <QueryNode Name="1" Type="SelectAt">
    <UseInput>
      <Compare>
        <GetNode Name="0" Index="1"
                          RefersTo="p.StreetNumName"/>
        <CompareWith>g.address</CompareWith>
      </Compare>
      <Compare>
        <GetNode Name="0" Index="2"
                          RefersTo="p.Suburb"/>
        <CompareWith>g.fine</CompareWith>
      </Compare>
      <Compare>
        <GetNode Name="0" Index="3"
```

```
                          RefersTo="p.zone.description"/>
        <CompareWith>g.coarse</CompareWith>
      </Compare>
    </UseInput>
    <Select>
      <se Index="0">g.easting</se>
      <se Index="1">g.northing</se>
      <se Index="2">
        <GetNode Name="0" Index="0" RefersTo="p.ID"/>
      </se>
    </Select>
    <From>
      <fe Extent="Geocode" Variable="g"/>
    </From>
    <Where/>
  </QueryNode>
```

This node uses the Geocode data source agent to convert from addresses to coordinates using a semijoin. It takes `p.StreetNumName, p.Suburb` and `p.zone.description` found previously in node 0, and looks to match them with `address, fine` and `coarse` in the Geocode data so that the coordinates (northing and easting) can be determined. In addition, the property ID as found previously is passed through to be used later.

4.4 Query Execution

The executor's basic function is to perform the actual queries to data sources, receive the results merging them into the final result and then passing that result back to the user. It is a reactive agent that interacts with the planner and data source agents (DSAs). When the executor receives a message from the planner, the message contains a detailed plan in XML format. The plan details precise operations for the executor to perform, and its contents form a graph of connected query nodes. The executor starts with the final node, the result of this final node will form the answer to the users original query.

For each query node the executor looks to see which other nodes are linked to as input nodes and moves on to process these. If there are multiple input nodes these will be processed in parallel. Eventually the executor will reach a query node that has no inputs, an OQL query string will be formed for this node and sent to a data source agent for processing. The executor does not find appropriate data source agents for each node, this is done by the planner and the executor reads from the XML plan the address for the DSAs that each node should be sent to. Agent conversations are an important feature supporting the operation of the executor, conversations allow the executor to keep track of simultaneous interactions with many different agents with very little effort.

Once a DSA has executed its query it returns a reference to the resultant data in a message to the executor. Once the executor receives this message, processing of the node is complete and it

moves back up the graph to the previous query node to see if all input nodes are now available. Once all input nodes are available the executor performs the query operation for this current node, this will often involve some operation on the input data, such as a join on two inputs, or using one input to perform a semijoin query to another data source agent. This process continues until the executor has returned back to the final query node, at this point a result to the user's query is available, and a reference to this result is passed to the user agent.

To access the data, both for the user agent accessing the final result and the executor accessing the results of individual queries to DSAs, it was decided that using agent messaging was not appropriate. The requirement to stringify each individual piece of data inside of an ACL content parameter represented a significant performance issue to what would be a high volume part of the system. The result of a query is a collection object modelled after the ODMG and Java2 collection frameworks. The agents pass a reference to the collection in the form of a CORBA IOR. To receive the contents of the collection an iterator is provided. The iterator allows individual elements of the collection to be retrieved, it can also retrieve multiple elements at once, this allows the user to choose from progressive streaming or faster block transfer of results.

4.5 Data Sources

A Data Source Agent (DSA) is responsible for providing access via agent communication language messages to a single data source. The first responsibility of a DSA is to register itself with the Resource Broker (DSA), which is responsible for keeping track of the DSAs that are available to the system. The DSA notifies the RB about the ontology for the data it wraps, the meta-data associated with the data, and with what sort of operations it can perform. This information is used by the Planner Agent as detailed in Section 4.3.5.

The ontology for a DSA will be specified by system developers at the time that the data is integrated with the system, although isome of this information is corrently coded into the planner. Part of this process is to provide a mapping from the data-source level ontology to a user-level ontology. The meta-data is also specified when the data is integrated with the system, but this will usually require less work as meta-data will usually be supplied with the data.

The operations that a DSA can perform depend on several factors. The type of data that is wrapped may constrain the possible operations, for example the geocoding data source is only capable of converting street addresses to northing and easting coordinates, not vice versa. There may be privacy or security issues that restrict the type of queries that can be performed, for example medical case data provided for research should not allow queries by a patient's name. And lastly the resources available at the DSA's location may constrain the queries that can be performed, for example a data source located on a dial-up connection may refuse queries that will result in a large amount of data to be transferred.

In its normal mode of operation the DSA waits for the Executor Agent to send queries to it. When a query is received the DSA converts the query to some form that can be used to query its internal data, and then performs the query. The results of the query are then packaged into CORBA, this is detailed further in Section 4.4.

The way that the DSA converts the query depends on the sort of data that it wraps. Currently three different types of data have been used with the system; relational databases, ESRI Shapefiles and socket based geocoding data. To query the relational database an SQL query is created. This is done using a schema file that maps the concepts in the data-source ontology to the relational entities that exist in the database. The ESRI Shapefile is somewhat more complex; there is no well-specified query language or query processing engine for this sort of data, so this has been developed by-hand. Currently a small number of possible queries on a Shapefile have been implemented, for example returning a code associated with the polygon that a point specified in the query is inside. The socket-based geocoding data is perhaps the most restricive. The only query that can be performed on this data source mirrors the socket-based protocol. This protocol allows a single address to be sent, and the coordinates for that address are returned in a text format. While this is a rather simple data source, its value is that it demonstrates that the DSA is able to wrap unusual types of data sources.

5 Summary

The NZDIS architecture is designed to provide an open, agent-based environment for the integration of disparate sources of environmental information. The intended use of the system is expected to lie between two ends of a spectrum of possible data gathering applications: at one end are tightlyintegrated database systems, where existing distributed database system techniques could be used, and at the other end are information sources distributed so widely that only web-based information discovery systems are practically feasible. Since many of the information sources to be integrated in the New Zealand context are expected to consist of flat or unstructured data files, the system does not presume to be a distributed database system and does not perform optimisations based on such an assumption. Instead, the NZDIS system offers the infrastructural components for integrating heterogeneous environmental data sources with known, but differing, collections of data and metadata.

6 References

Bradshaw, J. M., Dutfield, S., Benoit, P., Woolley, J. D. 1998. Kaos: Toward an Industrial-strength Open Agent Architecture. In: Bradshaw, J. M. (ed) Software Agents, AAAI/MIT Press.

Cranefield, S., Purvis, M., Nowostawski, M. 2000. Is it an Ontology or Abstract Syntax? – Modelling Objects, Knkowledge, and Agent Messages. In: Proceedings of the Workshopon Applications of Ontologies and Problem-solving Methods, European Conference on Artificial Intelligence, pp. 16.1-16.4.

Fegaras, L., Maier, D. 2000. Optimizing Object Queries Using an Effective Calculus. ACM Transactions on Database Systems.

FIPA. 2000. Foundation for Intelligent Physical Agents. http://fipa.org/specifications/, 2001/4/29.

Purvis, M., Cranefield, S. 1999. UML as an Ontology Modelling Language, In: Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden.

Purvis, M., Cranefield, S., Nowostawski, M. 2000a. A Distributed Architecture for Environmental Information Systems. In: Denzer, R., Swayne, D., Purvis, M., Schimak, G. (eds) Environmental Software Systems – Environmental Information and Decision Support. Kluwer Academic, Dordrecht, The Netherlands, pp. 49-56.

Purvis, M., Cranefield, S., Bush, G., Carter, D., McKinlay, B., Nowostawski, M., Ward, R. 2000b. The NZDIS Project: an Agent-based Distributed Information Systems Architecture, In: Sprague, R. H. (ed) Proceedings of the Hawai`i International Conference on System Sciences (HICSS-33), (CD ROM) IEEE Computer Society Press, Los Alamitos, CA.

Somogyi, Z., Henderson, F., Conway, T. 1995. Mercury: an Efficient Purely Declarative Logic Programming Language. In: Proceedings of the Australian Computer Science Conference. pp. 499-512.

van der Aalst, W. M. P. 1998. The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems, and Computers, 8(1), pp. 21-66.