Advertising Database Capabilities for Information Sharing

Suzanne M. Embury[1], Jianhua Shao[1], W. Alex Gray[1], and Nigel Fishlock[2]

[1] Department of Computer Science, Cardiff University,
P.O. Box 916, The Parade, Cardiff, CF24 3XF, Wales, U.K.
{S.M.Embury|J.Shao|W.A.Gray}@cs.cf.ac.uk
[2] Pirelli Cables Limited Communication Cables,
Wednesbury Street, Newport NP9 0WS, Wales, U.K.
fishlocn@pirellicables.co.uk

Abstract. The development of networking technology has resulted in a computing environment which is highly distributed, heterogeneous and dynamic. In order for the autonomous software components in such an environment to share their information, and hence to collaborate with each other, they must be able to advertise their capabilities — that is, to express what they have got to offer — in a form that can be understood by other resources.
In this paper, we study the problem of advertising the capabilities of an important class of resources within distributed information systems: namely, databases. We argue that current advertising mechanisms have limitations when applied to database systems, and propose an approach which overcomes these limitations. The resulting advertisement language is flexible and allows database components to advertise their capabilities in both a general and a specific way. We have demonstrated its utility by using it to advertise capabilities within a network of product databases created for Pirelli Cables Ltd.

1 Introduction

The development of networking technology has resulted in a computing environment which is highly distributed, heterogeneous and dynamic. Users of this environment, and developers of software components for use within it, increasingly expect to be able to make use of the facilities provided by other software components *without prior agreement*. In other words, if a component A needs the results of some computation or query which can already be performed by component B, then A should be able to request that the service be performed on its behalf by B. Moreover, A should be able to make the decision to exploit B's services at the point at which the need for them arises — this decision should not be hard-coded in at the time that component A was created. If, at some later date, a faster, more reliable and more accurate component C comes on-line, A should have the option of making use of the higher quality services this new agent offers, even though the original developer of A had no idea that C would ever exist.

B. Wangler, L. Bergman (Eds.): CAiSE 2000, LNCS 1789, pp. 47–63, 2000.

In order to achieve this sort of flexibility, some mechanism is needed by which the various software components in a distributed information system (DIS) can publish details of their capabilities, and by which components that require some functionality can discover which other systems might be able to provide it. In response to the same need within multi-agent systems, some authors have proposed the use of a special kind of software agent which can perform resource discovery tasks of this sort on behalf of other agents. This type of agent is called variously a *match-maker agent* [7], a *middle agent* [4] and a *facilitator* [9]. Typically, a match-maker agent will receive *advertisements* from agents, describing the services they are willing to perform. When an agent wishes to discover who is capable of undertaking some particular task, it sends a description of that task to the match-maker, which compares the agent's requirements against the advertisements stored in its local database. The advertisements that match indicate the set of agents which can perform the task. Details of these are then returned to the requesting agent.

Components performing match-maker functions can offer similar advantages for distributed information systems (DISs): they can provide some degree of protection against change in the availability or capabilities of components, and they can ease the design of new components by acting as a machine-manipulable representation of the existing capabilities of the network. However, current advertising mechanisms have limitations when applied to a class of components that are of particular importance in DISs — database systems.

In this paper, we present an approach to match-making in DISs that takes into account the special requirements of database components, while retaining the flexibility to advertise the capabilities of other kinds of software component. In particular, we suggest that an accurate representation of the capabilities of a database system must include both *domain-specific* aspects (e.g. the classes and relationships stored by the database) and *operational* elements (e.g. the general capabilities of the query language supported by the DBMS). The proposed approach allows database components to advertise their capabilities in both a general and a specific way. We have tested it in the context of a small prototype network of information sources developed for Pirelli Cables Ltd.

The remainder of the paper is organised as follows. We begin, in Sect. 2, by considering the particular problems posed by the need to advertise database capabilities. We then summarise previous approaches to match-making in Sect. 3. Section 4 outlines our proposed advertisement format, and Sect. 5 presents the process by which advertisements in this format are matched to requests. Finally, conclusions are given in Sect. 6.

2 Advertising Database Capabilities

In order to illustrate the problems involved in advertising database capabilities, we will present some examples based on KQML [5], a standard framework for agent communication. The issues raised, however, are not specific to KQML, which is used here simply as a convenient notation for the purposes of illustration.

In a KQML-compliant system, agents communicate by exchanging messages. Each message is a list of components. The first component, known as the *performative*, indicates the type of communication (e.g. `tell` for passing information to an agent, and `ask-one` or `ask-all` for requesting information from an agent). The subsequent components, known as *arguments*, indicate the content of communication. For example, the following KQML message illustrates the performative by which an agent informs another agent (usually a match-maker) of its capabilities:

```
(advertise,
    :sender    testManager
    :receiver  matchMakerAgent
    :language  KQML
    :content   (ask-one,
                   :sender    <anyone>
                   :receiver  testManager
                   :language  Prolog
                   :ontology  Cables
                   :content   ‘‘stress_test_result($date,
                              $engineer, $specNo, $result)’’))
```

Here, the `:sender` and `:receiver` arguments specify the agent which sent the message and the agent to which it is sent, respectively. The `:language` argument indicates the language in which the `:content` is expressed and `:ontology` defines the terms used in the content. The message in the above example states that the `testManager` agent is able to respond to `ask-one` messages whose arguments match with those given. That is, if the content of an `ask-one` message is expressed in Prolog and in terms of the `Cables` ontology, and if it matches the pattern given in the innermost `:content` slot, then the `testManager` agent can handle this message.[1]

When a request for some capability is received, the match-maker tries to match the content portion of the request against the content patterns in the advertisements. The success of this process depends on the following factors:

- *The precision of the advertisement.* The pattern given in an advertisement acts as a general description of a set of similar requests, any of which can be serviced by the adverting component. If a component over-advertises its capabilities (i.e. uses too general a pattern) then it may be asked to serve requests that it is not capable of handling. On the other hand, if a component under-advertises its capability (by using too strict a pattern) then it may not be asked to serve requests that it is quite capable of serving. In either case, matching performance is degraded.
- *The intelligence of the match-maker.* It is possible that the same capability may be specified in a number of ways. This means that the match-maker must

[1] The pattern takes the same form as a Prolog request, except that variables (specified here with the $ prefix) are used wherever we don't care about the form of the request.

be intelligent enough to recognise, for instance, equivalence or subsumption relationships between advertised capabilities, and between the capabilities and the request.

These two factors are inter-dependent. For example, if the match-maker is capable of learning from its experience, then over-advertisement may not be a serious problem. In this paper, however, we do not consider the issue of match-maker intelligence but focus instead on the problem of how database components can advertise their capabilities both precisely and at a sufficient level of generality.

To appreciate the problems associated with advertising database capabilities, consider the following example. Suppose that a database has the schema:

```
opticalCable(specNo, cableType)
stressTest(testDate, testEngineer, specNo, result)
```

and it wishes to advertise that it can answer all SQL queries expressible against this schema. Under the standard KQML advertisement scheme, the database component would be forced to advertise its capabilities by generating one advertisement for each form of query it can answer. For example:

```
(advertise,
    :sender    dbAgent
    :receiver  matchMakerAgent
    :language  KQML
    :content   (ask-one,
                 :sender     <anyone>
                 :receiver   dbAgent
                 :language   SQL
                 :ontology   Cables
                 :content    ‘‘SELECT count(specNo)
                               FROM opticalCable’’))

(advertise,
    :sender    dbAgent
    :receiver  matchMakerAgent
    :language  KQML
    :content   (ask-one,
                 :sender     <anyone>
                 :receiver   dbAgent
                 :language   SQL
                 :ontology   Cables
                 :content    ‘‘SELECT testEngineer, cableType
                               FROM opticalCable c, stressTest t
                               WHERE c.specNo = t.specNo’’))
:
and many more
:
```

This need to anticipate all the query forms that might be sent to the database is unsatisfactory for the following reasons:

- For any realistic database system, trying to advertise *all* the queries that it can handle in this way is almost impossible. As a result, some queries will be left out, resulting in an inevitable under-advertisement of its capabilities.
- Even if we can tolerate a degree of under-advertisement, the match-maker will be loaded with a large number of highly-specific advertisements. This will degrade matching performance.
- Such specific advertisements are difficult to analyse intelligently. The best a match-maker can do is to match incoming requests syntactically and exactly.

All these points suggest that an approach to advertising database capabilities based purely on pattern-matching is unworkable. We require a more general and more expressive advertisement language in which a range of functionalities, rather than just service signatures, can be described. Indeed, the limitations of the pattern-based approach have been recognised by other researchers, and in the next section we examine the suitability of more advanced forms of agent-based match-making for advertising database capabilities.

3 Current Approaches to Advertising

The approach to advertisement discussed above is based on that proposed for facilitator agents, within the Knowledge Sharing Effort [9]. The role of a facilitator is to present the appearance of an agent which has the combined capabilities of all the other agents in the distributed system. Individual agents need communicate with just this single facilitator agent, rather than having to know the addresses and capabilities of all other agents on the network. In order to present this illusion, each facilitator maintains a database of advertisements, given in KQML, which are matched against incoming requests. However, the KSE researchers recognised the limitations of a purely pattern-based approach and allowed constraints to be placed on the values of pattern variables. When a request is matched against an advertisement, these variables are instantiated with values from the request and the constraints are checked against them.

This pattern matching style is most suitable when the capabilities of the advertising agent can be described as a small number of named services. It is much less practical for advertising very broad and generic services, such as database query answering, which may require the creation of a large (possibly infinite) number of advertisements. Later authors (e.g. [7]) have tried to redress this balance by extending the richness of the facilitator's pattern matching capabilities.

More recent work has generalised the concept of pattern-based content languages for advertisements to allow the description of many database queries in a single advertisement. Vassalos and Papakonstantinou, for example, have proposed a language called p-Datalog [13] for advertising database queries. This language generalises Datalog programs by introducing a distinguished form of variable that can be used in place of constants within rule definitions. In itself,

this is no more useful than the KQML patterns introduced in Sect. 2. However, a further development of p-Datalog, called the Relational Query Description Language (RQDL) [12], allows pattern variables to be used in place of the functors of terms, as well as their parameters. This allows a wider set of queries to be described by a single specification. In particular, query types can now be specified independently of any domain facts (e.g. "fetch the <A> of all <B> whose <C> attribute is greater than <D>"). This language allows many different query forms to be described using a single advertisement, and so contributes to solving the problems of database advertising. However, many kinds of agent can only handle requests which are relevant to a particular domain. This is certainly true of database components, which can only answer queries that are expressed in terms of their own particular schema. Rich pattern matching capabilities, if used without any regard to the domain of interest of the query, will result in over-advertisement for database systems, with a corresponding degradation in the performance of the network as a whole.

Other authors have focussed trying to provide richer constructs for modelling the domain-specific aspects of agent capabilities, rather than on extending pattern matching. DISs such as SIMS [1] and TAMBIS [2] make no use of pattern matching and instead use only domain information to carry out resource discovery tasks. For example, in both SIMS and TAMBIS a common domain model expressed using a description logic is used to describe the combined domain models of all resources in the network. Relevant resources are identified by classifying the incoming request against the common model, and identifying those resources which commit to the concepts occurring within the request. This form of advertisement, while appropriate in relatively homogeneous agent networks, can also result in over-advertisement, since it assumes that every agent which commits to the concepts occurring within a request is able to answer that request, regardless of whether its querying capabilities are up to the task.

In a variant on this domain-centred approach, Decker *et al.* have proposed an advertising format for database agents based on KQML advertisements, in which the advertisement language includes a specification of the schema over which queries can be evaluated [3]. Under this approach, a resource is considered to be a suitable recipient of a request if it involves no schema elements which are not also recorded within the advertisement. While the recognition of the importance of the schema in advertising database capabilities is a distinct step forward, we believe that it is not appropriate to embed details of a database component's schema explicitly within advertisements in this way. Firstly, the schemas of many real database systems (particularly legacy database applications) are very large, containing hundreds of tables and thousands of attributes. Duplication of all this information within an advertisement results in highly complex capability descriptions, which are difficult to reason with efficiently. Secondly, this approach commits the database system to advertising requests relating to its schema *at the time of the advertisement* and not as it may be at the time a request is to be serviced. Over time, use of an advertising scheme that is intolerant of updates

to source schemas will mean that advertisements become out-of-date, resulting in both under- and over-advertisement.

More recently, the LARKS language for the advertisement of agent services [11] has been proposed, which combines a variety of different types of matching. LARKS assumes the availability of an ontology of terms, which is used to compute similarity measures between the terms used within a request and an advertisement. It also uses ideas from component-based development to determine whether the parameters of the advertised service are compatible with those in the request, and whether pre- and post-constraints on the parameters in the advertisement are compatible with those in the request. LARKS represents the most sophisticated approach to the advertisements of general agent services proposed to date. However, the use of "plug-in" matching to determine compatibility of parameters is not relevant to matching of database queries, which can have very similar parameters but very different semantics. Moreover, LARKS takes the unusual approach of requiring users to convert their requests into the same format used for advertisements before matching. Effectively, the user is forced to guess what the advertisement for the service they require might contain. While this is reasonable for named agent services, where very little semantics is available from analysis of the bare procedure call, it is less appropriate for database queries, where much of the semantics is present in the request itself.

If a capability is to be precisely defined, then both its *operational* aspects (i.e. "what the agent can do") and its *domain-specific* aspects (i.e. "what it can do it to") must be specified. Recognising this point, researchers on the InfoSleuth project have made some preliminary steps towards combining both elements in a single advertisement format [8]. Following the approach described above, InfoSleuth advertisements contain information on the subset of the common ontology that a particular resource can handle embedded within the advertisement. This allows consideration of domain-specific aspects. For the operational aspects, InfoSleuth advertisements may also contain terms ¿From a special *service ontology*, which defines different typ of operational capability (e.g. relational algebra). It is not yet clear what role these service terms will play in matching requests in the Infosleuth system. However, what is clear from our own investigations so far is that a *combined* approach to advertising both domain and operational elements is required if capabilities from a wide variety of agents and resource types are to be advertised. This is particularly true when attempting to advertise the potentially infinite capabilities of database systems in a precise, yet flexible manner. In the next section, we present our own approach to this problem.

4 An Advertisement Language for Database Capabilities

How does one describe the "domain aspects" and "operational aspects" of a component's capabilities? The domain of a database is defined by the entity types and the relationships (or other data model constructs) that appear in the database's schema. The operational aspect is defined by the underlying DBMS. Different DBMSs (particularly legacy systems) will offer different functionali-

ties, and it is this that determines how the available data may be manipulated. Both aspects must be known to the match-maker if incoming requests are to be matched accurately. To accommodate this, we propose the following basic framework for advertisements:

```
(advertise,
    :sender      <component>
    :receiver    match-maker
    :language    Advertise
    :content     (ask-*,
                     :sender     <anyone>
                     :receiver   <component>
                     :language   <language>
                     :ontology   <ontology>
                     :domain     d-constraint(<expr>)
                     :operation  o-constraint(<expr>)
                     :content    <expr> ))
```

As in KQML advertisements, the outermost `:content` slot contains a specification of a collection of requests the advertising component is willing to service.[2] The innermost `:content` slot contains a pattern expression which describes the syntactic structures the advertising component is willing to handle.

Unlike KQML, however, there is also a `:domain` slot and an `:operation` slot, each of which specifies additional constraints on the pattern expression. The constraints given in the `domain` slot describe the characteristics of requests whose domain of interest matches that of the advertising component. The `:operation` slot, on the other hand, indicates that the component can only serve requests whose semantics satisfy the conditions it specifies. Thus, even if a request r matches a pattern expression e, it may still not be servable by the advertising agent if r fails on either the operational or domain constraints. Match-making is therefore based on semantic as well as syntactic considerations.

4.1 Simple Pattern Expressions

Before describing the form that the domain and operational constraints may take, we present the format of the pattern expressions that may be specified in the `:content` slot. We first define the concept of "capability" formally and then discuss how a capability may be described using simple expressions.

Definition 1. *A* request *is a recursive structure of the following form:*

$$\alpha(\tau_1, \tau_2, \ldots, \tau_m)$$

where α is a constant term and each $\tau_i, i = 1, \ldots, m$ is either a constant or a structure of this same form.

[2] `ask-*` matches with any of the range of `ask` performatives (`ask-one`, `ask-all`, etc.).

This definition approximates a syntax tree for the request language, and is intended to model the internal representation of the request within the match-maker agent. For example, the following SQL query:

```
SELECT specNo FROM opticalCable WHERE cableType = 'Unitube'
```

can be expressed as:

```
select(tables([opticalCable]), attrs([specNo]),
                          op(=, attr(cableType), 'Unitube'))
```

A *well-formed request* is a request whose structure represents a legal expression in the content language.[3] It is trivial to define this notion of legality of expression formally, and we do not define it here. However, we note that the arguments in the request structure are "positional", i.e. different permutations of arguments results in different requests. For example, the following:

```
select(tables([opticalCable]), attrs([specNo]),
                          op(=, attr(cableType), 'Unitube'))

select(tables([opticalCable]), attrs([specNo]),
                          op(=, 'Unitube', attr(cableType)))
```

are considered to be distinct requests.[4]

We define a component's capability in terms of the set of requests that it claims to be able to serve.

Definition 2. *Let* $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ *be the set of all well-formed requests. A* capability $\mathcal{C}$ *is a set of such requests; that is,* $\mathcal{C} \subset \mathcal{R}$*. Each* $c_i \in \mathcal{C}$ *is referred to as a* capability element *(or* element *when the context is clear).*

Advertisement of a capability is therefore specification of the set $\mathcal{C}$. A naive approach is to enumerate all the elements as a disjunction. That is, the component anticipates all possible requests that it can answer and specifies each of them in detail. However, as we have argued, this approach is unworkable for database resources (or any component where $\mathcal{C}$ is large). To advertise a capability more concisely, we require a means of advertising several capability elements in one expression, and hence introduce the use of simple pattern expressions.

Definition 3. *A* simple pattern expression *takes one of the following forms:*

[3] In order to simplify the presentation, we assume a reduced dialect of SQL that can be represented by syntax trees of the form given. However, the issues discussed are largely independent of the choice of content language.

[4] This assumption is not a serious limitation. If, for example, our match-maker is intelligent enough to recognise the commutative property of the = operator, then it could treat both requests as equivalent. In this paper, however, we focus on the specification of capabilities, rather than the intelligence of the match-maker.

1. *a constant pattern* $\alpha(\tau_1, \tau_2, \ldots, \tau_m)$*, where each* α*,* τ_1*, ...,* τ_m *is a constant,*
2. *a variable pattern* $\alpha(\tau_1, \tau_2, \ldots, \tau_m)$*, where at least one* $\alpha, \tau_1, \ldots, \tau_m$ *is a pattern variable, or*
3. *a single pattern variable* v*.*

Note that if any τ_i*,* $1 \leq i \leq m$*, is a recursive structure, then rules 1) and 2) apply recursively.*

We denote a pattern variable in an advertisment by a literal with a $ prefix. For example, the following are all valid pattern expressions:

```
select(tables([opticalCable]), attrs([specNo]),
                          op(=, attr(cableType), 'Unitube'))

$opn(tables([opticalCable]), attrs([specNo]),
                          op(=, attr(cableType), $value))

$expr
```

The use of pattern variables in this way allows agents to abstract their capabilities (i.e. describe them more concisely). To capture this notion of abstraction formally, we give the following definition.

Definition 4. *Let* c_1 *and* c_2 *be two simple pattern expressions. We say that* c_1 abstracts c_2 *in the following mutually exclusive cases:*

1. *if* c_1 *is a single pattern variable* v
2. *if* c_1 *is a constant pattern* $\alpha(\tau_1, \ldots, \tau_n)$ *and* c_1 *lexically matches* c_2
3. *if* $c_1 = \alpha(\tau_1, \ldots, \tau_n)$*,* $c_2 = \alpha'(\tau'_1, \ldots, \tau'_n)$ *where* α *is a variable or* $\alpha = \alpha'$*, and for each* $\tau_i, i = 1, \ldots, m$*,* τ_i *is a variable or* $\tau_i = \tau'_i$

Again, if any τ_i *is a recursive structure, then rules 2) and 3) apply recursively.*

Definition 5. *Let* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *be two capabilities.* $\mathcal{C}_1$ subsumes $\mathcal{C}_2$*, denoted by* $\mathcal{C}_1 \succ \mathcal{C}_2$*, if for every* $c_2 \in \mathcal{C}_2$*, there exists a* $c_1 \in \mathcal{C}_1$ *such that* c_1 *abstracts* c_2*.*

Obviously, an ideal capability advertisement is a pattern expression that subsumes the intended set of requests both concisely and precisely. The use of pattern variables in expressions allows us to achieve this at various levels. At the lowest level, we have *domain abstraction.* That is, we allow pattern variables to be used wherever domain values are used. For example, the following is an abstraction of a class of capability elements that all have the same structure, but use different domain values:

```
select(tables([opticalCable]), attrs([specNo]),
                          op(=, attr(cableType), $value))
```

Here, `$value` is a pattern variable which can be instantiated with any constant value. As this example shows, the abstraction achieved at this level is fairly limited. The next level of abstraction, *argument-level abstraction*, overcomes this limitation to an extent. With this form of abstraction, we allow pattern variables to be used for the arguments of a capability element, e.g.

```
select(tables([opticalCable]), attrs([specNo]), $cond)
```

Here, a pattern variable (`$cond`) is used to abstract the class of SQL selection conditions. Thus, the above expression subsumes both:

```
select(tables([opticalCable]), attrs([specNo]),
                          op(=, attr(cableType), 'Unitube'))
```

and

```
select(tables([opticalCable]), attrs([specNo]),
                          op(<>, attr(cableType), 'Unitube'))
```

Note that as the capability elements become more abstract, the likelihood of over-advertisement increases and the accuracy of the advertisement becomes more dependent on the constraints given in the `:operation` and `:domain` slots.

As Vassalos and Papakonstantinou [13] have pointed out, it is also possible to use pattern variables to generalise operations. For example, the following expression illustrates this *operation-level* abstraction:

```
$db-op(tables([opticalCable]), $attrs, $cond)
```

The element in this example advertises that it is willing to serve any operation involving only the table `opticalCable`. This expression abstracts both of the following requests:

```
select(tables([opticalCable]), attrs([specNo]),
                          op(=, attr(cableType), 'Unitube'))

delete(tables([opticalCable]), (),
                          op(<>, attr(cableType), 'Unitube))
```

Operation-level abstraction is very general. Careful specification of constraints on such abstractions is therefore important if software components are not to be flooded with requests that they cannot answer.

All the abstractions we have introduced so far are limited in that they specify a fixed structure. In cases where a software component wishes to advertise a capability that is made of capability elements with different structures (e.g some with three arguments and some with two), we allow *structure abstraction* — the highest level of abstraction in our language. Consider the following example,

```
$any-req
```

Here the expression is a single pattern variable `$any-req` which abstracts, and hence can match, any request. Without appropriate constraints, this form of pattern leads to an extreme form of over-advertisement in which the agent claims to be able to serve any request whatever.

4.2 Composite Pattern Expressions

If we view matching as evaluation of a truth-valued function that returns true if the request matches the pattern and false otherwise, we can combine simple pattern expressions using the standard logical operators `AND`, `OR` and `NOT`. For example, the following composite pattern expression shows how two simple expressions may be combined using the `OR` operator:

```
select($tables, $attrs, op(=, $attr, $value)) OR
select($tables, $attrs, op(<>, $attr, $value))
```

This pattern expression matches a request if either of the two simple expressions match it. In other words, the set of requests that the component is willing to handle is the union of the request sets specified by the two simple expressions.

We can also use the `AND` and `NOT` operators to combine patterns. For example, a component `C1` may advertise that it can handle the retrieval of any attributes from `opticalCable` relation, except the `cableType` attribute alone:

```
select(tables([opticalCable]), $attrs, $cond) AND NOT
select(tables([opticalCable]), attrs(cableType), $cond)
```

Of course, a similar effect could be achieved by adding a constraint to the `:domain` slot. We will now describe how such constraints can be used to limit the over-advertising tendencies of patterns.

4.3 Constraining Pattern Variables

The pattern variables introduced in the previous section are useful in that they allow database resources to specify a superset of the requests they are willing to handle very concisely. While, in general, over-advertisement is to be preferred to under-advertisement, it is clearly beneficial for all parties if advertisements can be specified as accurately as possible. Pattern variables must therefore be further constrained so that the match-maker agent is aware of any additional restrictions on the domain of interest or the semantics that can be handled by the advertising component. For example, a database component that stores details of optical cables cannot answer queries which relate to copper cables, even if some aspects of the query also relate to optical cables. Similarly, consider a wrapped component that is capable of evaluating certain SQL queries against a flat file, by converting them into instructions to the Unix Grep program. This component will only be capable of answering a restricted set of SQL queries; namely, those consisting of a series of simple projections and selections on the structure of the file.

Definition 6. *A* domain constraint *in an advertisement relating to a component* c *is a truth-valued expression, parameterised by the pattern variables appearing in the advertisement. The constraint determines the subset of requests that match with the pattern expression that are concerned wholly with domain elements known to the component.*

Definition 7. *An* operational constraint *in an advertisement relating to a component* c *is a truth-valued expression, parameterised by the pattern variables appearing in the advertisement. The constraint determines the subset of requests that match with the pattern expression that the component has the operational capacity to evaluate.*

For the purpose of this paper, we assume that both domain and operational constraints (as stated in the `:domain` and `:operation` slots of the advertisement format respectively) are first order logic expressions over pattern variables.[5] Consider the following fragment of an advertisement sent by component `C1`:

```
:operation  (simple_query $e)
:domain     (subset (domain $e) (commits_to C1))
:content    $e
```

Here, `(simple_query $e)` is a predicate which is true if the value of `$e` is equivalent to a request fragment with the semantics of a "simple" relational algebra query. We define a "simple" query to be one in which the condition of the query is a simple conjunction of comparison operators on attributes.

The domain constraint states that the domain elements occurring in the request must be a subset of the domain elements to which the advertising component commits. In other words, the component cannot answer queries asking about domain elements which are not contained within its schema. Notice that we have *not* embedded any details of the actual schema within the advertisement itself. Instead, we assume that the match-maker component has the responsibility of discovering what the current set of commitments of component `C1` are — perhaps by interrogating some ontological knowledge agent or data dictionary.

Operational and domain constraints like those given in the example are specified by building expressions using the usual logical connectives (`and`, `or` and `not`) and a number of pre-defined predicates and functions, such as `simple_query` and `commits`, stored in a special purpose *task ontology*. If a predicate such as `simple_query` is not defined by a formula (i.e. it is a ground definition), then we assume that it is a Boolean valued function stored in the task ontology that returns true if its arguments have the required properties and false otherwise. In our prototype, for example, we have used the "grammar" shown below to recognise the subset of legal syntax tree forms that correspond to the class of simple queries.

[5] It is, of course, possible to use various other formalisms to specify the constraints, such as a description logic.

```
<simple_query>    ::= select(<tables>, <attributes>) |
                      select(<tables>, <attributes>), <condition>)
<tables>          ::= tables(<table names>)
<table names>     ::= <table name> | <table names>,<table name>
<attributes>      ::= attrs(<attribute names>)
<attribute names> ::= <attribute name> |
                      <attribute names>,<attribute name>
<condition>       ::= op(=, <attribute name>, <attribute name>) |
                      op(=, <attribute name>, <constant>) |
                      op(=, <constant>, <attribute name>)
<attribute name>  ::= <name> | <table name>.<name>
<table name>      ::= <name>
<name>            ::= <identifier>
```

5 Matching Process

We now consider how a request from a user agent is matched with an advertisement by the match-maker. A request is first matched syntactically with the pattern expression and then semantically with the constraints. The process of syntactical matching is straightforward. An advertisement a matches a request r if it can be made identical with r by substituting the pattern variables in a with the corresponding values in r. In other words, a matches r if a abstracts r. For example, the advertised pattern:

```
select(tables([opticalCable]), attrs([specNo]), $cond))
```

can be made to match the following request:

```
select(tables([opticalCable]), attrs([specNo]),
                          op(=, attr(cableType), 'Unitube'))
```

by substituting the variable `$cond` by `op(=, attr(cableType), 'Unitube)`. Our match-maker therefore performs a simple pre-order matching process, recursively matching the request with a simple pattern expression, argument by argument from left to right. The complexity of the matching process is therefore linear in the size of the advertisement base (the number of advertisements). The matching of a request with a composite expression is a trivial extension to this process.

Once the request has been matched with the pattern expression, the match-maker will proceed to attempt a semantic match against the domain and operational constraints. The semantic matching process can potentially be very complex, depending on the form of the constraints and the reasoning power of the matchmaker. As our focus in this paper is the specification of advertisement, we have used standard first order logic specification and inference for domain and operation constraints, as discussed in the previous section. However, there are clearly opportunities for improving the efficiency of matching by using more

advanced algorithms (based on constraint logic programming, for example). A simple match-maker based on the proposal presented in this paper has been implemented within the context of a case study of a distributed information system at Pirelli Cables Ltd Newport factory [6]. The matching process is implemented in Prolog, with a Java front-end. The JATlite toolkit was used to provide KQML-based communication between software components.

6 Conclusions

The ability to advertise database capabilities concisely yet precisely is an important pre-requisite for the development of flexible DIS architectures, based on "location-independent" cooperation between components. A DIS based on match-making is more resilient to both short- and long-term change in its constituent components, as existing components can automatically make the decision to use services offered by new components. We have described an approach to advertisement that overcomes the limitations of previous proposals for describing database capabilities. The result is a capability description language that combines the following advantages:

- Pattern variables may be used at all levels to allow many different request forms to be advertised using a single pattern expression.
- Further flexibility in describing capabilities is provided by the ability to combine patterns using the `OR`, `AND` and `NOT` operators.
- Additional constraints can be placed on the semantics of the capabilities which are advertised, over and above the syntactic constraints given by pattern expressions. This allows agents to characterise their capabilities much more precisely than would be possible using pattern matching alone.
- Despite our focus on pattern matching as the basis of our advertisement format, the importance of matching on domain concepts as a means of reducing over-advertisement is not neglected. Additional constraints can be specified to ensure that incoming requests are matched only with resources which share their domain of interest.
- Our approach does not require that the entire schema of the resource be embedded explicitly within the advertisement. Thus, we do not place unreasonable demands on the match-maker's ability to cope with very large advertisements, nor do advertisements grow out of date as resources evolve.

Our proposal thus combines the flexibility of pattern expressions of RQDL (albeit without the rigorous formal underpinnings of that proposal) with the realistic approaches to matching large domains in DIS systems and the ontological approach to operational constraints hinted at within the InfoSleuth proposal [8].

A number of open questions regarding the advertisement of capabilities for database agents still remain. There is as yet very little understanding of the trade-off between advertisement complexity (to reduce over-advertising) and the time required to match requests with such complex advertisements. It may be that a certain amount of over-advertisement is beneficial to the agent system as

a whole if this means that the match-maker agent is able to operate efficiently. Further investigation is also required into the level of intelligence to be provided by the match-maker agent. For example, it may be useful to endow the match-maker with incremental learning capabilities to allow it to refine its database of advertisements based on experience. The match-maker can monitor the results of the requests it matches, to build up a knowledge base of the successful and unsuccessful cases. Initially, the system's performance would be relatively poor, as the match-maker would have very little knowledge of the participating resources, but it would improve over time as its experience grew. A facility for adaptive match-making of this kind could remove the responsibility for monitoring the capabilities of the network from individual components, and optimise the connections between consumer and provider components to improve overall performance of the DIS.

Acknowledgements

We are grateful to Martin Karlsson for the implementation, to Alun Preece for his helpful comments on a draft of this paper, and to the members of the OKS group at Cardiff University for their feedback on our advertisement format.

References

1. Y. Arens, C.-N. Hsu, and C.A. Knoblock. Query Processing in the SIMS Information Mediator. In Austin Tate, editor, *Advanced Planning Technology*, pages 61–69. AAAI Press, 1996.
2. P.G. Baker, A. Brass, S. Bechhfer, C. Goble, N.W. Paton, and R. Stevens. TAMBIS - Transparent Access to Multiple Biological Information Sources. In J. Glasgow *et al.*, editor, *Proc. of 6th Int. Conf. on Intelligent Systems for Molecular Biology*, pages 25–34, Montréal. AAAI Press, 1998.
3. K. Decker *et al.* Matchmaking and Brokering. In M. Tokoro, editor, *Proc. of 2nd Int. Conf. on Multiagent Sytems*, Kyoto. AAAI Press, 1996.
4. K. Decker, K. Sycara, and M. Williamson. Middle-Agents for the Internet. In *Proc. of 15th Int. Joint Conf. on Artificial Intelligence (IJCAI'97)*, pages 578–583, Nagoya, Japan. Morgan Kaufmann, 1997.
5. T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proc. of 3rd Int. Conf. on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, USA, 1994. ACM Press.
6. M. Karlsson. A Matchmaker Agent for Database Applications. Master's thesis, Cardiff University, September 1999.
7. D. Kuokka and L. Harada. Supporting Information Retrieval via Matchmaking. In C. Knoblock and A. Levy, editors, *Proc. of AAAI Spring Symp. on Information Gathering from Heterogeneous, Distributed Envs.*, pages 111–115. AAAI, 1995.
8. M. Nodine, W. Bohrer, and A.H.H. Ngu. Semantic Brokering over Dynamic Heterogeneous Data Sources in InfoSleuth. In M. Papazoglou, C. Pu, and M. Kitsuregawa, editors, *Proc. of 15th Int. Conf. on Data Engineering (ICDE'99)*, pages 358–365, Sydney. IEEE Computer Society Press, 1999.

9. N. Singh, M. Genesereth, and M.A. Syed. A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation. *International Journal of Cooperative Information Systems*, 4(4):339–367, 1995.
10. K. Sycara, J. Lu, and M. Klusch. Interoperability among Heterogeneous Software Agents on the Internet. Technical Report CMU-RI-TR-98-22, Robotics Institute, Carnegie Mellon University, October 1998.
11. K. Sycara, J. Lu, M. Klusch, and S. Widoff. Matchmaking among Heterogeneous Agents in the Internet. In S. Murugesan and D.E. O'Leary, editors, *Proc. of AAAI Spring Symp. on Intelligent Agents in Cyberspace*, Stanford, USA, 1999.
12. V. Vassalos and Y. Papakonstantinou. Describing and Using Query Capabilities of Heterogeneous Sources. In M. Jarke *et al.*, editor, *Proc. of 23rd Int. Conf. on Very Large Data Bases*, pages 256–265, Athens, 1997. Morgan Kaufmann, Inc.
13. V. Vassalos and Y. Papkonstantinou. Expressive Capabilities Description Languages and Query Rewriting Algorithms. *Journal of Logic Programming*, 43(1):75–122, 2000. Special Issue on Logic-Based Heterogeneous Information Systems.