Fuzzy Service Selection in a Distributed Object-Oriented Environment

Patrick Bosc, Ernesto Damiani, Member, IEEE, and Mariagrazia Fugini

Abstract—In network-centered execution environments, client objects can transparently invoke services offered by remote server objects, according to their published interface. The object selection problem requires the evaluation of the fitness of a pool of candidate server objects on the basis of the available information about their functional and nonfunctional features. Network-centered systems usually store such information in a trader agent that can be browsed or queried by client objects. In this paper, a fuzzy data model is proposed as the basis of the design of such a trader system, taking into account synergy between objects' features. Our trader is based on a fuzzy query algebra allowing for deriving operator definitions (therefore, query execution mechanisms) at run time, on the basis of user-selected semantics.

Index Terms—Fuzzy querying, object-oriented distributed systems, server selection.

I. INTRODUCTION

THERE past few years have witnessed a tremendous increase in the interest in network-centered execution environments, now a mainstream architectural pattern for software systems design [7]. In such environments, client objects explore an enterprise-wide Intranet or the global Net in order to access services offered by a number of distributed servers.

In this paper, we present a complete framework aimed at increasing the flexibility of network-centered systems, focusing on dynamic server selection, i.e., run-time evaluation of the fitness of candidate server objects with respect to a task on the basis of the available information about them.

It is widely recognized [30] that run-time server selection should take into account both the servers’ functional characteristics (i.e., the services they provide) and their nonfunctional features, such as the servers’ price or performance, or their current load.

A network-centered environment usually provides a trader service identifying server objects on the basis of functional information. The common object request broker architecture (CORBA) trading service specification [30] is a well-known example of a component brokerage facility in which clients specify queries in a simple language and the trader responds by providing zero, one or more matching object references which the clients can use to bind to a server of their choice. Traders are particularly useful in very large distributed systems containing thousands or even millions of objects. Indeed, on such a large scale, it is impractical to require that every object instance be identified by a unique name. Rather, instances must be differentiated by their properties.

For a trader to supply such a service, candidate server objects must inform it that they wish to be listed in its repository. Having received the registrations, the trader must then efficiently store all the relevant searchable properties, so that when a client connects to perform a search, the response can be generated efficiently.

Again, CORBA traders are good examples in as much as they offer standard interfaces allowing candidate servers to register themselves (i.e., their names and object references) as well as their properties. Other interfaces allow clients to compose their search predicates and submit them to the trader.

Much research [16] has been done on the syntax and semantics of these predicates. Generally speaking, existing trading services are limited in as much as their classification and retrieval models are based on Boolean logic: a server either totally satisfies a client’s needs, providing the desired services—or it does not.

In this paper we propose to compensate this lack of flexibility by using a fuzzy data model and query algebra as the basis of the design of an intelligent trader system. Such a trader must satisfy several requirements, some of which are listed below:

1) Query transparency: In order to preserve the standard trading interface as much as possible, neither servers nor clients must be forced to deal explicitly with fuzzy values.

2) Parametric semantics: Weights associated with features may have various semantics, for example, they may express “how well” or “at what cost” a server performs the service described by the features. Clients should be able to choose among several semantics when formulating a query to the trader. The selected semantics must dictate the operation of the trader’s query execution engine and, therefore, determine query results.

3) Ranked-list results: The trader should yield, for each available server, its degree of satisfaction relatively to the query. Clients should receive a ranked list of answers and be able to choose the best fit from the available servers.

4) Synergy management: Often, servers features are more relevant to the user request if they are available together with other features. We call this effect feature synergy. Users should be able to specify synergy requirements in their queries, so that the trader may take them into account while computing query results.
As we shall show, three of the above requirements can be satisfied using a suitable fuzzy data model and its associated query techniques as an *internal engine*, while communicating with servers and clients via a standard interface. A preliminary version of this simple model was presented in [17]. In our approach, a membership value (called *weight*) is associated with each server feature, either functional or nonfunctional, and is transparently computed inside the trader on the basis of crisp values provided by servers as a part of their standard registration. The same applies to weights associated with properties composing clients’ queries.

The fourth requirement, however, is seldom if ever encountered in database querying. Informally, dealing with *synergy* means managing situations where a company or organization owning a server may prefer to provide a service together with others and offer a discount to that effect; alternatively, clients may prefer a “single-stop” service and be prepared to pay a little more to obtain it. This is, in our opinion, a specific problem of distributed object-oriented systems and therefore needs new solutions.

In this paper, we use a function defined as a simple extension to the well-known notion of *fuzzy measure* [22] to structure the set of features associated with each component, in order to take into account interdependencies among features. Besides being used to fine-tune the trader’s operation, our synergy can also be exploited to assess the total synergy value of a given server with respect to a set of invocations composing an application process by computing a straightforward extension of the *discrete Choquet integral* [24]. Such a computation implements the notion of analytical query introduced by previous research in the field of information systems [8].

We first provide a formalization of the service selection problem and propose a general solution satisfying the requirements listed above. Then, we give some design and implementation guidelines for a CORBA-compliant trader system incorporating our approach. The paper is organized as follows. In Section II, we outline the general framework of a service selection system, while in Section III we describe how user-selected semantics can be seamlessly incorporated in our approach.

Section IV introduces a function defined over the partition lattice of a feature set aimed at modeling synergy, i.e., the fact that features may acquire value when they are simultaneously offered by a service. It is also shown that an extension of the discrete Choquet integral can be used to support the computation of analytical queries. Section V outlines how the synergy notion can be integrated in the framework of our fuzzy trader system. Section VI deals with the design and implementation of a CORBA trader service supporting our selection techniques, while Section VII surveys some related work in this field. Finally, Section VIII draws the conclusion.

## II. THE GENERAL FRAMEWORK OF A SERVICE SELECTION SYSTEM

We begin by establishing some notation. Let FL be a language (henceforth, called *feature language*) built on a finite lexicon L. This language allows the client to formulate queries toward the network specifying the servers’ desired features. Here we shall make no assumptions about FL, in the simplest case it will be a finite set of terms of L, i.e., a standard domain-dependent vocabulary [15].

Each distributed service available on an active network is associated with a fuzzy descriptor D, defined as a pair of fuzzy sets \(F;NF\) where \(F = \{(f_i,w_i), i = 1,2,\ldots,n\}\) and \(NF = \{\{w_j,j = 1,2,\ldots,m\}(\text{with } w_i, w_j \in [0,1])\}\) are sets of respectively functional and nonfunctional weighted features (i.e., \(\forall i,j f_i, w_j\in FL\), interpreted as a fuzzy set on the universe FL.

Intuitively, functional features describe what the remote service can do, while nonfunctional ones might specify where, when, and at what cost the service is provided.

As we shall see, weights \(w_i \in [0,1]\) are computed by fuzzifying crisp values provided by the services themselves using suitable fuzzy predicates.

Each trader contains a set of descriptors \(\{D(s)\}\), forming a descriptor base, where \(s\) is a generic service available over the network.

In turn, a user query is composed of two crisp sets of couples, namely \(P = \{(p_i, I_i), i = 1,2,\ldots,n\}\) and \(N = \{(n_i, I_i), i = 1,2,\ldots,m\}\), where \(p_i, n_i\) are features belonging to FL, and labels \(I_i\) are values associated with features.

From the client point of view, \(P\) includes the desired (positive) server’s features while \(N\) specifies those features (negative) the server should not have, giving the possibility to identify servers that should be excluded from the search. Obviously, for a query to be valid, \(p_i \neq n_i \forall i\).

### A. A Fuzzy Data Model for the Selection Service

We are now ready to present the design of a descriptor base \(\{D(s)\}\) storing fuzzy descriptors of the services available on the network. \(\{D(s)\}\) is a structured collection of descriptors providing semantics-aware descriptions of servers’ properties in the line of [12].

We consider each trader to be associated with an application domain or a specific theme, for instance, image processing, hypermedia applications, and cartographic systems. Although, the issue of interconnection between traders is beyond the scope of this paper, it is interesting to observe that the traders on the network can be organized according to a suitable ontology based on evolvable taxonomies, like the one proposed by the CommerceNet Consortium for Electronic Commerce [20].

Indeed, the set of tunable fuzzy predicates stored at each trader, together with a suitable hierarchy, could be used to model a distributed, organization-wide knowledge-base with uncertainty. In the sequel, however, we shall focus on the operation of a single trader.

In our approach, a simple fuzzy relational model is used. Fuzzy relations are obtained by applying a monotonic fuzzy predicate on a crisp database relation \(R\) built on a fixed set of domains \(DOM = \{DOM_i\}\), namely \{object identification, type, feature\}, where type denotes functional and nonfunctional features.

In order to obtain a fuzzy relation, every tuple of \(R\) is augmented with its membership degree \(\mu_R\), from 0 to 1, interpreting
Fig. 1. An example of fuzzy relation in the descriptor base.

<table>
<thead>
<tr>
<th>OID</th>
<th>Feature</th>
<th>μ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>video</td>
<td>0.8</td>
</tr>
<tr>
<td>1</td>
<td>service</td>
<td>0.4</td>
</tr>
<tr>
<td>2</td>
<td>video</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>audio</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Fig. 2. The Video service fuzzy predicate.

how each tuple satisfies a fuzzy predicate \( P \) applied to the relation \( R \).

Fig. 1 shows a fuzzy relation describing some features of two audio/video servers.

For the sake of simplicity, here we consider only functional features. \( OID \) stands for Object Identifier.

For the video service feature shown in the example above, the fuzzy predicate could be defined on the universe of the frame rate values, that is an integer interval from 0 (corresponding to still images) to 30 (corresponding to broadcast quality).

Analogously, the audio service predicate can be defined on the universe of audio sampling rates from 0 to 64 kHz.

A sample definition of the video service predicate is reported in Fig. 2.

According to the video service predicate definition, crisp frame rate values provided by the servers are transformed into membership values for the fuzzy relation defined above.

It is important to observe that this computation is entirely transparent to servers and clients alike and can take place both when the servers sign on, i.e., communicate the availability of their services, and periodically, as a consequence of new load or network traffic conditions [15].

In passing, we also remark that the definition of the fuzzy predicates associated with FL terms can evolve in time according to domain or technology related policies.

The trader’s role is to help the client to choose among the available servers. Upon receipt of a query, the trader selects a fuzzy context-dependent predicate for each feature in the positive part and computes the fuzzy membership values corresponding to the crisp values specified by the client.

Each property is associated with a certain fuzzy predicate and weighted by a value between 0 and 1 (obtained by transformation of absolute values according to the fuzzy predicates’ definition). The list of properties, together with the features in the negative part (usually with membership value 1) defines a fuzzy request to the trader, which is nothing but another fuzzy relation \( Q \), as shown in Fig. 3.

Informally, we shall write \( Q = FP \cup FN \) to express the fact that the fuzzy request \( Q \) can be interpreted as the union of two fuzzy sets \( FP \) and \( FN \), corresponding respectively to the (fuzzy-fied) positive and negative part of the user query.

III. QUERY EXECUTIONS FOR INDEPENDENT FEATURES

In this section, we shall assume all features in the query to be mutually independent and show how fuzzy query techniques can be employed to provide query transparency, parametric semantics, and ranked list results.

We define the degree of satisfaction \( CV \) (Confidence Value) of a fuzzy request \( Q = FP \cup FN \) performed against a component descriptor in the descriptor base \( \{D(s)\} \) as follows:

\[
CV(Q, D(s)) = (FP \subseteq D(s)) \land (FN \cap D(s) = \emptyset).
\]

Using the concept of cardinality of a (finite) fuzzy set, fuzzy set inclusion used in (1) can be written as follows:

\[
\text{Inc}(FP \subseteq D(s)) = \frac{\sum_{x \in \text{FL}} \text{Count}(FP \cap D(s))}{\sum_{x \in \text{FL}} \text{Count}(P)}
= \frac{\sum_{x \in \text{FL}} T(\mu_{FP}(x), \mu_{D(s)}(x))}{\sum_{x \in \text{FL}} \mu_{FP}(x)}
\]

where \( T \) is a triangular norm operator and \( FL \) is the finite feature language introduced in the previous section. In the denominator of (2), the fuzzy cardinality of \( FP \) is computed.

In the numerator, the cardinality of the intersection is classically computed by applying a norm \( T \) to the membership values of each feature appearing both in \( FP \) and in \( D(s) \), and adding the results. In this setting, the choice of the norm \( T \) determines the strength of the intersection, i.e., the degree of co-occurrence of a feature in both \( FP \) and \( D(s) \) required to provide a given contribution to the intersection cardinality, the classical min intersection being the weakest.

Whatever the choice of the norm, (2) provides an additive view of inclusion that will compensate low (or even zero) membership values for some features in the descriptor if other features exhibit a high enough level of membership both in the query and the descriptor. In other words, even if an object’s descriptor \( D(s) \) has a zero membership for some features mentioned in the query (i.e., the object does not offer the associated service) it is still possible the degree of inclusion of \( FP \) in \( D(s) \) to be high. This depends on the membership of other features in \( D(s) \) with respect to the corresponding ones in \( FP \).

This general property of cardinality-based inclusion makes it awkward to use (2) to map various user-provided semantics of the features membership values, where weights in the query and in the descriptor have different interpretations.
Alternatively, set inclusion can be defined using a fuzzy implication, namely

\[ FP \subseteq D(s) = \min_{x \in F} \mu_{FP}(x) \rightarrow \mu_{D(s)}(x). \quad (3) \]

As we shall see, the choice of the implication to be used for computing set inclusion in (1) can be made based on the query semantics as specified by the client according to the interpretations of membership values in \( D(s) \) and \( FP \).

The semantics supported by our system are simple to understand as, from the user’s point of view, they correspond to instructions on how the crisp values originally included in the query are to be used by the query execution engine.

Computation of CV via (1) and (3) can be readily described in terms of a fuzzy relational division [2], [3]. This description will also turn out to be useful for the design and implementation of the trader agent.

Let us consider two relations \( R(X, A) \) and \( S(Y, A) \) where \( A, X, \) and \( Y \) are sets of attributes. The (crisp) division of \( R \) by \( S \), denoted \( R[A/A]S \), is a relation on \( X \), which can be defined as follows:

\[ x \in R[A/A]S \quad \text{if} \quad \forall a \in S[A], \quad (x, a) \in R. \]

The operation of division of \( R \) by \( S \) can also be defined as a set inclusion

\[ x \in R[A/A]S \iff S[A] \subseteq \Gamma^{-1}(x) \quad \text{where} \quad \Gamma^{-1}(x) = \{ a : (x, a) \in R \}. \quad (4) \]

Following [2], we now examine the extension of the division to fuzzy relations.

The extension of the division to the fuzzy case can be written as follows:

\[ \mu_{R[A/A]S}(x) = \min_{a \in A} \mu_{S[A]}(a) \rightarrow \mu_{R}(x, a). \quad (5) \]

Thus, the computation of a ranked list of descriptors according to their CVs with respect to a given fuzzy request \( Q = (FP, FN) \) can be made according to the following procedure:

1) The fuzzy division result \( RES(Q+) = D(s)[\{ \text{feature} \} / \{ \text{feature} \}]P \) is computed using (5).

2) Unwanted servers are filtered out by computing \( F = \pi_{\text{OID}}(\sigma_{\text{Feature}} \bigwedge_{\text{FN}}(\sigma_{\text{ OID}} \bigwedge \text{RES}(Q+))(D(s))) \) and finally \( RES(Q) = \sigma_{\text{ OID} \neq F}(\text{RES}(Q+)) \). The net effect of these steps is deleting from the query result all objects whose features were included in the negative part \( N \) of the query.

3) The tuples of the result table \( RES(Q) \), whose single column is \( \text{OID} \), object identifier, will then be ranked according to their membership values and presented to the client.

In the next subsection we shall see how the above execution mechanism can be customized according to user-defined semantics by using different implications in step 1 of the above procedure.

A. Query Execution Semantics

Our system supports two main types of semantics: both are based on fulfillment [2], and are application specific in that they provide two interpretations particularly suited for distributed systems’ applications, i.e., ideal value (or price-based) and threshold (or performance-based) interpretation. After presenting these semantics from the user point of view, we shall briefly discuss how they can be associated with different fuzzy implications in query execution.

1) Ideal value (or price-based) fulfillment means that the user expects a minimum distance between the values specified in the positive part of the query and the ones found in the descriptor of the retrieved server.

2) Threshold (or performance-based) fulfillment stands for a cut-off vision: the value specified in the query has to remain as close as possible to the server’s one, while being greater than it. In other words, while only lower values than those specified in the query are acceptable, values that are much lower are less desirable than values closer to those specified in the query.

For both our visions of fulfillment, the client can possibly ask for absorption. This means rejecting any server, which does not offer all the features in the positive part.

Considering the previous example and according to the semantics definitions given above, a user can for instance request a video streamer service with the following features:

a video service with a rate of 30 frames per second and
an audio service with a sampling rate of 16 kHz.

The request is made specifying price semantics with absorption. This means that servers whose feature values are comparable (though not exactly equal) to those specified in the query will be equally acceptable, regardless if those values turn out to be smaller or greater than the ones specified in the query. However, it is imperative that retrieved servers offer both the required features. This specific semantics, known as Quality of Service semantics [33] is typical of distributed multimedia services.

When the user indicates the desired semantics, the system selects an implication to be used to compute the query result.

B. Interpretations of Fuzzy Implication

We are now ready to discuss the association between our user-centered query semantics and fuzzy implications. Three basic approaches have been proposed for the definition of fuzzy implication.

1) Classical view of implication (S-implications), where \( a \rightarrow b \) is classically equivalent to \( \neg a \lor b \). In the fuzzy framework, this can be written as \( a \rightarrow b = S(\eta(a), b) \) where \( S \) is a triangular co-norm modeling a disjunction and \( \eta \) is an involutive negation operation. If we associate \( S \) with the maximum we get the standard Kleene–Dienes implication (defined as \( a \rightarrow b = \max(1 - a, b) \)).

2) Partial-ordering view, (R-implications), based on the idea that implication reflects a partial ordering on propositions such that \( a \rightarrow b = 1 \iff a \leq b \). Such implications are defined as the residuation in the proposition lattice structure with respect to a semi-group operation \( T \) modeling a conjunction. Namely, implication is denoted as
follows: \( a \rightarrow b = \sup \{ c \in [0, 1], T(a, c) \leq b \} \). Among \( R \)-implications, we get the classical Goguen implication \( a \rightarrow b = 1 \) if \( a \leq b, b/a \), otherwise, if we associate \( T \) with the multiplication operation. Gödel implication (defined as \( a \rightarrow b = 1 \) if \( a \leq b, b \) otherwise) is obtained when \( T \) is the minimum. \( R \& S \)-implications belong to both the above categories. For instance we get Lukasiewicz implication, defined by: \( a \rightarrow b = 1 \) if \( a \leq b, 1 - a + b \) otherwise, as an \( R \)-implication with Lukasiewicz norm \( T = \max(a + b - 1, 0) \) and as an \( S \)-implication with \( S = \min(1, a + b) \).

3) Quantum Logic view (QL-implications), leading to the form of implication used in quantum logic, defined as \( a \rightarrow b = S(n(a), T(a, b)) \) where \( S \) is a co-norm, \( n \) a strong negation and \( T \) is the \( n \)-dual of \( S \). i.e., \( T(a, b) = n(S(n(a), n(b))) \). When \( S \) is the maximum, we get Zadeh implication, defined as \( a \rightarrow b = \max(1 - a, \min(a, b)) \). We shall not deal with this kind of implication in the remainder of the paper.

In all the above cases, fuzzy implications are symmetrical, reflexive and piecewise linear monotonic functions. They also provide the basic property \( 1 \rightarrow a = a \).

1) Performance Fulfillment: In the case of performance fulfillment, membership values in \( P \) are considered as fulfillment degrees to be reached, i.e., thresholds. For this semantics, we need

\[
\mu_{R/\lambda \leq S}(x) = 0 \Leftrightarrow (\exists a, \mu_S(a) > 0 \land \mu_R(x, a) = 0)
\]

and

\[
\mu_{R/\lambda \geq S}(x) = 1 \Leftrightarrow (\forall a, \mu_S(a) > 0 \leq \mu_R(x, a)),
\]

Both Gödel and Goguen implications satisfy these conditions and also provide the absorption property.

In order to choose the implication more suitable for association with performance semantics, we need to compare their behavior when a tuple \((x, a)\) exists which satisfies the current feature to a degree less than the required one (i.e., if \( \exists a, \mu_S(a) > \mu_R(x, a) \)). Obviously, Gödel implication ensures that an element \( x \) will be retrieved with a degree as high as \( \mu_R(x, a) \) when \( \mu_S(a) \) is larger than \( \mu_R(x, a) \). In this case, the result does not depend at all on \( \mu_S(a) \). Goguen implication, in turn, gives the ratio between \( \mu_R(x, a) \) and \( \mu_S(a) \), and therefore yields the relative level of fulfillment of the considered feature.

For performance semantics without absorption we need \( \mu_S(a) \rightarrow \mu_R(x, a) \) to be greater than 0 even if \( \mu_R(x, a) = 0 \), in order to prevent the CV of a descriptor to be zero whenever a feature of the positive part of the query is lacking in a server’s descriptor. This behavior excludes a clear-cut threshold effect; intuitively however, the higher the distance between the user-provided threshold and the value in the descriptor, the lower we would like the result to be.

Following the same empirical reasoning as before, we first consider Lukasiewicz implication. We note that, when \( \mu_R(x, a) = 0 \), the result is \( 1 - \mu_S(a) \), which prevents absorption as desired. Below threshold, the Lukasiewicz implication gives 1 and therefore does not depend at all on \( \mu_S(a) \).

Dienes implication gives \( 1 - \mu_S(a) \) when \( \mu_R(x, a) = 0 \), but in general does not provide any threshold effect. Then, we conclude that Lukasiewicz implication fits the (nonabsorption) threshold vision better than Dienes implication.

2) Price Fulfillment: The case of price fulfillment turns out to be less straightforward. Suppose the query contains a feature whose membership value is \( b \); given two services where the same feature is offered with membership \( \alpha' = b + \delta \) and \( \alpha'' = b - \delta \) the ideal value behavior would require \( \alpha' \rightarrow b = \alpha'' \rightarrow b \).

In other words, if the required value for a feature is \( b \), a descriptor having the feature with membership \( \alpha' = b + \delta \) should have, all other features’ membership values being equal, the same CV as one including the feature with membership \( \alpha'' = b - \delta \).

Obviously, this “pure” symmetric ideal value vision cannot be provided by any standard monotonic implication. Defining a function \( \psi(a \rightarrow b, b \rightarrow a) \) to guarantee the desired behavior is indeed possible for any given implication, but such definition must be tailored to the particular fuzzy implication used (this problem has been studied in detail in [4]).

Here we follow a different line, defining a general pseudo-implication as follows:

\[
a \leftrightarrow b = \min(a \rightarrow b, b \rightarrow a).
\]  (6)

Whatever the fuzzy implication plugged in (6), our pseudo-implication is piecewise linear and \( 1 \leftrightarrow a = \min(a, 1 \rightarrow a) = a \).

The rationale for (6) is to provide a behavior reasonably close to the pure ideal value vision, while allowing for smoothly plugging in different implications to get the desired behavior with respect to absorption. Such a definition also allows for modularity in the execution engine implementation. Using Goguen implication, we get \( \alpha' \leftrightarrow b = b - \delta/b \) (for \( b > \delta \)) while \( \alpha'' \leftrightarrow b = \delta/b + \delta \). The lack of symmetry is given by \( \Delta(b, \delta) = \delta^2/(b+\delta) \), which is acceptable provided that \( \delta \ll b \) (note that, by construction, it is always the case that \( \delta < b \)). In other words, as one would expect, the interval of \( \delta \) values giving nearly symmetric behavior is larger for higher values of \( b \). Fig. 4 shows the graphic of \( \Delta(b, \delta) \) for some values of \( b \) and \( \delta \). The lower right part of the picture, where \( \text{Error} < 0.1 \), corresponds to the safe area, i.e., the zone where our pseudo-implication exhibits a nearly symmetric behavior.

Therefore, employing Goguen implication in (6) requires additional caution as it can only be done if values of \( b \) and \( \delta \) are inside the safe area.

With Lukasiewicz implication there is no such worry, we get exactly the desired behavior, since \( \alpha' \leftrightarrow b = 1 - b = \alpha'' \leftrightarrow b \).

In passing, we note that were the implication in the right hand side of (6) Dienes implication, we would get \( \alpha' \leftrightarrow b = \max(1 - \delta/b + \delta \).
Fig. 5. Semantics versus implications.

\[
\begin{array}{|c|c|c|}
\hline
\text{Semantics} & \text{Absorption} & \text{Non-absorption} \\
\hline
\text{Performance} & \text{Goguen} & \text{Lukasiewicz} \\
\text{Price} & (\text{Pseudo})\text{Goguen} & (\text{Pseudo})\text{Lukasiewicz} \\
\hline
\end{array}
\]

Fig. 6. A Naïve algorithm for server selection.

```
while \( C \), again acceptable for \( b \).

C. A Naïve Algorithm for Server Selection

Whatever (pseudo) implication is used to perform the division, we can give the naïve algorithm of Fig. 6.

For each element \( x \) of the divided relation \( R \), our algorithm sequentially seeks the corresponding tuples \((x, a)\) for each element \( a \) of the relation \( S \).

This algorithm is very costly in terms of memory accesses (when the tuple \((x, a)\) does not exist the algorithm examines the whole relation \( R \)). Improvements, based on heuristics and indexing of table \( R \), are therefore necessary.

For example, supposing the existence of a threshold \( t \) that the servers’ final weights must reach in order to be selected, some heuristics can be used.

An heuristic of failure is valid for any implication: element \( x \) will not be retrieved if \( \exists a \in S, \mu_S(a) \rightarrow \mu_S(x, a) < 1 \), since the division computes a minimum value. The second heuristic concerns Goguen implication: for a given element \( x \), if there exists an element \( a \) in \( S \) such that the tuple \((x, a)\) does not exist in \( R \), then we obviously have \( \mu_R(x, a) = 0 \) and \( \mu_S(a) \rightarrow \mu_S(x, a) = 0 \).

D. A Worked-Out Example

For this example, we shall refer to the descriptor base of Fig. 7.

First of all, we recall that the trader has computed the above descriptor base by applying suitable fuzzy predicates to the crisp values supplied, for each feature, by the remote services. The descriptor base can be updated by remote services reflecting changing conditions in their computational power or load. Now suppose a user query to be \((\text{audio service, 16 KHz; video service, 22 fps})\). Upon reception of the query, the trader uses its internally defined fuzzy predicates to translate it into a fuzzy request, namely \((\text{audio service/0.3, video service/0.7})\). In the example of Fig. 8, we compute the ranked query results for all possible semantics.

First of all, we observe that performance semantics ranks OID 1 above OID 2, as one would expect since object 1 supplies all the required services with better performance than required, while this is not the case for object 2. Dropping the absorption property only inserts object 3 in the list in the last place, again as expected.

Intuition suggests that price semantics should rank object 1 and object 2 equally.

Indeed, the acceptably small difference (0.05) in ranking for price semantics with absorption is only due to the definition of our pseudo implication [Section 3]. However, price without absorption ranks object 1 and object 2 exactly exaequo, as Lukasiewicz implication is plugged in (6).

Dropping absorption, object 3 is added to the query result, but it trails behind the others, again as one would expect.

Note, however, that if we decrease the required audio sampling in the query (taking for instance \((\text{audio service/0.1, video service/0.7})\)), we get OID 3, \( \text{min} (\text{min}(1-0.7+1), \text{min}(1-0.1,1)) = 0.9 \) and object 3 gets an advantage, overtaking the other objects even if it does not provide all the services requested by the client.
IV. INTERDEPENDENT FEATURES AND SYNERGY

The approach outlined in Section III does not take into account the rather frequent and interesting case of interdependency or synergy between the remote service’s desired features.

In this section, we shall introduce a fuzzy synergy function defined on the lattice partition of a set to model this situation.

The rationale behind this synergy function is twofold.

1) First of all, it is aimed at improving the fuzzy selection mechanism described in previous sections, by taking into account the added value a server provides when it offers, simultaneously, all services needed by the client. Such an improvement will be described in detail in Section V.

2) Secondly, it allows for analytical retrieval, aimed at exploring the set of services needed to perform a given business procedure and assessing their individual fitness to the task.

The latter analysis, which has been extensively studied by information system research [8], is more oriented to human interaction with the trader, in order to map the high-level procedures to lower level network services. An example of an analytical query is “analyze all the services needed to complete a LoanManagement business process,” whose purpose is to explore why and how a server is involved in the process. Suppose that the LoanManagement process consists of a sequence of service requests, such as

1) request for an electronically signed document;
2) activation of a remote workflow;
3) activation of an e-mail service.

Intuition suggests that adding the fuzzy measure of synergy across all the requests gives a cumulative measure of a server’s fitness with respect to the process under analysis, whose result can be compared to actual usage patterns measured on the network. As we shall see, this analysis involves evaluation of a functional on the synergy function, playing the same role of a fuzzy integral [19] to rank services according to domain-specific heuristics.

A. Preliminary Examples

From the client’s point of view, the synergy notion is easily described; for instance, a feature could be considered useful only when provided by the same server together with another one, and much less valuable otherwise. From the server’s point of view, two or more features could be provided jointly at a discounted price, or offering some features simultaneously could be advantageous for architectural reasons.

We shall first model the synergy effect in a crisp data model setting. We use CP and CD(s) to denote, respectively, the crisp support sets of FP and D(s); in Section V, we will show how synergy can be used in the context of the trader system described in the previous sections.

Let us start with some introductory examples. Suppose first that a conference server object offers an audio service and a video service, as well as a whiteboard service.

For reasons related to the server software architecture, the organization running the server might prefer to provide video and audio service together rather than separately.

Thus, from the server’s point of view, a query requesting both audio and video service is to be considered preferable with respect to another query requesting a single one of them and the whiteboard service. Intuitively, letting CP′ = \{audio service, video service\}, CP′′ = \{audio service, whiteboard\}, and CD(s) = \{audio service, video service, whiteboard\} we could write:

\[ SV(CP′, CD(s)) > SV(CP′′, CD(s)) \]

(where SV stands for synergy value), although in this case \[ CD(s) \cap CP′ \neq CD(s) \cap CP′′ \].

Let us now repeat the same line of reasoning from the client’s point of view.

For instance, a client requiring all three services might prefer to get both the audio and video service from the same server object, while being ready to purchase the whiteboard service from a separate server if necessary.

So, if two servers have descriptors CD(s′) = \{audio service, video service\} and CD(s′′) = \{audio service, whiteboard\}, and supposing CP = \{audio service, video service, whiteboard\}, again according to intuition we could write

\[ SV(CP, CD(s′)) > SV(CP, CD(s′′)). \]

Although, here we have again

\[ |CD(s′) \cap CP| < |CD(s′′) \cap CP|. \]

Obviously, in a real application setting, both situations outlined in our introductory examples might happen simultaneously.

In the next subsection, we shall examine in detail the problem introduced above and provide a formal definition of SV.

B. A Fuzzy Synergy Function

In order to formalize this problem, let \( E = \mathcal{P}(FL) \) be the (crisp) power set of a feature language FL and \( CD(s) \in E \) be a crisp descriptor, i.e., the set of features associated with component \( s \).

Let \( \Pi_{CD} \) be the lattice of the (disjoint) partitions of CD. Finally, let CP be the crisp support set of the positive part of a user query and \( \Pi_{CT} \) the lattice of the (disjoint) partitions of CP.

Like the Boolean algebra, the partition lattice associated with a set of cardinality \( n \) is a uniform, supersolvable lattice.

At each level \( k = 1, 2, \ldots, n \) of the lattice we find partitions of the support set in \( n - k + 1 \) blocks. Intuitively, the lattice join operation \( \vee \) corresponds to putting together two blocks of two partitions in \( k \) blocks to form the same coarser one in \( k + 1 \) blocks.

The meet operation \( \wedge \) corresponds to separating a block of two partitions in \( k \) blocks to form the same finer decomposition in \( k + 1 \) blocks.

The lattice top is the partition in a single block (i.e., the support set itself), while the bottom is the partition into singletons.

We shall not give here a formal definition of partition lattices and their strict relationship to Boolean algebras; the interested reader is referred to [10] and [11].
A sample partition lattice for $CD(s) = \{A, V, W\}$ is reported in Fig. 9.

We intend to model the synergy value $SV$ of each partition $S = \{s_1, s_2, \ldots, s_m\} \in \Pi_{CP}$ of a given query $CP$ with respect to a partition $T = \{t_1, t_2, \ldots, t_n\} \in \Pi_{CD}$ of a crisp descriptor $CD(s) \in E$ by means of a lattice function $SV_T : \Pi_{CP} \rightarrow [0, 1]$. This function is an extension to the partition lattice of the usual fuzzy measures, defined on Boolean algebras.\(^1\)

Intuitively, we expect this function to behave like a lexical distance based on the cardinality of the intersection of the flat sets $CP$ and $CD(s)$, while giving a premium when the synergy requirements of the user query are satisfied by the server descriptor and vice versa.

In this setting, we can impose the analog of the monotonicity axiom of fuzzy measures

$$\forall A, B \ A \subseteq B \Rightarrow |B| \geq |A|$$

expressing the fact that, obviously, the more features of $CP$ are included in $CD(s)$, the better the server $s$ will satisfy the query $CP$. We are now ready to formalize our synergy requirements. In order to denote interdependent features, we suppose server $s$ to specify its descriptor $CD(s)$ as a partition $T$. In other words, $CD(s)$ is declared as a union of disjoint subsets $CD_i(s)$

$$CD(s) = \bigcup_i CD_i(s) \quad i = 1, 2, \ldots, b_{CD}$$

With reference to Fig. 7, object 1 could describe itself as $T = \{\{\text{audio service}, \text{video service}\}, \{\text{whiteboard}\}\}$, expressing the fact that the server, for architectural reasons, wishes to encourage clients to use jointly its audio and video features.

From the client’s point of view, interdependent features can be specified again by declaring the crisp support set of the positive part of the query $CP$ to be the union of $b_{CP}$ disjoint subsets $CP_j$ composing a partition $S$

$$CP = \bigcup_j CP_j \quad j = 1, 2, \ldots, b_{CP}$$

A query like $S = \{\{\text{audio service}\}, \{\text{video service}, \text{whiteboard}\}\}$ expresses the fact that the client would prefer to use a single server for video and whiteboard services.

Now we are ready to express a measure of the matching between interdependent features as declared by the server and as requested by the client as follows:

$$SV_T(S) = \frac{\sum_i f(\max_j \{CD_j(s) \cap CP_i\})}{|CD(s)| + |CP|}$$

where $f$ is defined as $2|CP|$ if $CP_i \subseteq CD_j(s)$ and $|CD_j(s) \cap CP|$ otherwise.

We observe that in the above equation $\max \bigcup_i$ computes the intersections of all descriptor partition blocks $CD_j(s)$ with the single query partition block $CP_i$ and returns the intersection that has the maximum cardinality.

The rationale for (7) is the following: function $f$ doubles the cardinality of the maximum intersection between a query partition block $CP_i$ and the descriptor partition blocks $CD_j(s)$ ($j = 1, 2, \ldots, b_{CD}$) if $CP_i \subseteq CD_j(s)$, i.e., as a premium for requesting together features that the server declared to be part of an interdependent subset.

The values computed by $f$ are then added up for all partition blocks $CP_i$ in the user query.

Generally speaking, (7) behaves as a lexical proximity in as much as it gives zero whenever $CD(s) \cap CP = \emptyset$ and weakly increases with the cardinality of the intersection $CD(s) \cap CP$. In fact, when $b_{CD} \geq b_{CP}$, it is always the case that

$$\sum_i f(\max_j \{CD_j(s) \cap CP_i\}) \geq |CD(s) \cap CP|$$

as the minimum of the summation occurs when no $CP_2$ is included in any $CD_j(s)$. In this case, the summation accumulates for each $CP_i$ in a disjoint partition $S$ of $CP$, the cardinality of the maximum intersection of $CP_i$ and one of the $CD_j(s)$ in a disjoint partition $T$ of $CD(s)$. But this is exactly the cardinality of the intersection between (flattened) sets $CP$ and $CD$.

Now suppose that $|CD(s) \cap CP| = k$, while $|CD(s)| = n$ and $|CP| = m$. If $CD(s)$ and $CP$ are flat sets, i.e., both their partitions contain a single block, and $CP \not\subseteq CD(s)$, from (7) we get

$$SV_{CD(s)}(CP) = \frac{|CD(s) \cap CP|}{|CD(s)| + |CP|} = \frac{k}{n + m}$$

in fact, in this case $f$ gives the cardinality of the intersection $CD(s) \cap CP$ and we get a measure of lexical proximity whose maximum is $n/m + n$ when $CP$ contains all the features in $CD(s)$, plus others. When $CP \subseteq CD(s)$ and, therefore, $m = k < n$, we get $2m/n + m$. If $CP = CD(s)$ we obviously have:

$$SV_{CD(s)}(CP) = \frac{(2n)(2n)}{2n} = 1$$

If disjoint partitions $T$ and $S$ denoting synergy with $b_{CD} = b_{CP}$ have been specified for $CD(s)$ and $CP$ we have $SV_T(S) \geq k/n + m$, up to a maximum of 1 when $S = T$; i.e., $CD(s) = CP$ and both the query and the descriptor are equally partitioned. In this case, the number of the partition blocks is $b = b_{CP} = b_{CD}$ and we get

$$SV_T(S) = \frac{2 \sum_{i=1}^{b} |CP_i|}{2n} = 1.$$
features and the synergy value only denotes the matching between the granularity requested by the client and the granularity preferred by the server.

In passing we note that, for any given $\mathbf{CD}(s)$, SV values induce a partial order on the disjoint partitions of CP. The order is not total as two partitions may well have the same SV value w.r.t. $\mathbf{CD}(s)$.

As an example, SV values for $\mathbf{CD}(s) = \mathbf{CP} = \{\text{video service}, \text{audio service}, \text{whiteboard}\}$ are depicted in Fig. 10.

Again in passing, we note that if $\mathbf{CD}(s)$ and CP are always flat sets and we set $\mathbf{FP}$ we define the trivial fuzzy measure on the Boolean algebra defined as 0 everywhere except on $\mathbf{CP}$, where it coincides with the degree of inclusion of (the fuzzy set) $\mathbf{FP}$ in (the fuzzy set) $\mathbf{CP}$, all features are considered independent from one another, and we are back to the computation of division as described in Section 3. Note that this measure takes the usual value 1 on the universe $\mathbf{CP}$ when $\mathbf{CP} = \mathbf{CD}(s)$.

C. Analytical Retrieval

As far as analytical retrieval is concerned, given a set $\mathbf{B} = \{S_1, S_2, \ldots, S_m\}$ of partitioned queries composing an application process, we need to aggregate synergy values to compute the total fitness. A straightforward way to compute the total fitness $F$ of a server with respect to $\mathbf{B}$ is to use a weighted arithmetic mean, as follows:

$$F_T(\mathbf{B}) = \sum_i \omega_i \sum_j \text{SV}_T(S_j)$$

where the round brackets ( ) indicate a permutation such that $\omega_1 \leq \omega_2 \leq \ldots \leq \omega_m$. Moreover, $\mathbf{A}_j = \{j, (j+1), \ldots, (n)\}$ and $\mathbf{A}_{(n+1)} = \emptyset$. Equation (9) gives the usual weighted mean only when the fuzzy measure $\nu$ is additive (we recall that a fuzzy measure $\nu$ defined on a Boolean algebra is additive whenever $\nu(S \cup T) = \nu(S) + \nu(T)$ for $S \cap T = \emptyset$; in that case $\nu(\mathbf{A}_j) - \nu(\mathbf{A}_{(j+1)}) = \nu((i))$. If a fuzzy measure is additive, it suffices to define it on the singleton sets, as the values it assumes elsewhere can be easily computed by repeatedly applying the additivity definition.

D. Aggregation of Partition-Based Queries

Additivity can be straightforwardly extended to functions defined on a partition lattice, requiring $\nu(X \vee Y) = \nu(X) + \nu(Y)$, where $X, Y$ are partitions and $\vee$ is the lattice-theoretical operation giving a coarser partition. Again in this case, if a function is additive it suffices to define it on lattice atoms, i.e., partitions composed of $n - 1$ blocks. In passing, we note that this is not the case for our fuzzy synergy function: for a given partition $T$ of $\mathbf{CD}(s)$, and a given $S$, we have

$$\text{SV}_T(S' \vee S'') \neq \text{SV}_T(S') + \text{SV}_T(S'')$$

For example, if $S = \{\mathbf{AW}\}$, $S' = \{\mathbf{AV}\} \{\mathbf{W}\}$, and $S'' = \{\mathbf{VW}\} \{\mathbf{A}\}$, we get (for $T = \{\mathbf{VW}\} \{\mathbf{A}\}$)

$$\text{SV}_T(S' \vee S'') = \frac{3}{2} 
eq \text{SV}_T(S') + \text{SV}_T(S'')$$

Indeed, this decrease in synergy makes sense, as $S$ is a coarser query than $S'$ and $S''$ and the policy of the server remains unchanged (i.e., $T$ is the same for both queries). To investigate this subject, we define a straightforward extension to the discrete Choquet integral $C_{\nu}(\mathcal{\omega}) : \Pi_N \to \mathbb{R}$ where $\nu$ is a synergy function defined on the partition lattice $\Pi_N$. Namely, we write

$$C_{\nu}(\mathcal{\omega}) = \sum_{i=1}^m \omega_i \left[ \nu(\mathbf{A}_i) - \nu(\mathbf{A}_{(i+1)}) \right]$$

where, again, the round brackets ( ) indicate a permutation such that $\omega_1 \leq \omega_2 \leq \ldots \leq \omega_m$. Moreover, $\mathbf{A}_i$ is the partition $i$ blocks $\{(i), (i+1), \ldots, (m-i+1), (m-i+1), \ldots (m)\}$ such that $\mathbf{A}_j \leq \mathbf{A}_{(i+1)}$ and $\mathbf{A}_{(m+1)} = \emptyset$. $\mathbf{A}_i$ includes singleton blocks starting from $\{(i)\}$ to $\{(m-i+1)\}$ and a final block including all remaining elements.

Informally, we can say that vector $\mathcal{\omega}$ characterizes the integral in as much as it weights how much going from a given partition to a coarser one according to a certain permutation (i.e., affecting some specific blocks identified by the permutation itself) affects the integral's value. When $\omega(i) = 1$ for $i = 1, 2, \ldots, m$, the permutation is irrelevant; using the fixed point permutation with reference to Fig. 10 and taking $T = \{\mathbf{AV}\} \{\mathbf{W}\}$, we get the following table for $\text{SV}_T(\mathbf{A}_i) - \text{SV}_T(\mathbf{A}_{(i+1)})$:

$$\begin{array}{ccc} \mathbf{A}(i) & \mathbf{A}(i+1) & \text{SV}_T(\mathbf{A}_i) - \text{SV}_T(\mathbf{A}_{(i+1)}) \\ \mathbf{AW} & \mathbf{AV} & 1/2 - 1/2 \\ \mathbf{VW} & \{\mathbf{A}\} \{\mathbf{V}\} \{\mathbf{W}\} & 0 \\ \{\mathbf{A}\} \{\mathbf{V}\} \{\mathbf{W}\} & \emptyset & 1 \end{array}$$

Giving $C_{\nu}(1, 1, 1) = 1/3$. Obviously, this is not the arithmetic mean of the SVT values for the three queries involved, which...

<table>
<thead>
<tr>
<th>$\mathbf{CD}(s)/\mathbf{CP}$</th>
<th>$\mathbf{VAW}$</th>
<th>$\mathbf{V}$</th>
<th>${\mathbf{VA}}$</th>
<th>${\mathbf{VW}}$</th>
<th>${\mathbf{V}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>${\mathbf{AW}}$</td>
<td>1/3</td>
<td>1/3</td>
<td>1/3</td>
<td>1/2</td>
<td>1</td>
</tr>
<tr>
<td>${\mathbf{V}}$/ ${\mathbf{A}}$</td>
<td>1/3</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>${\mathbf{VW}}$/ ${\mathbf{W}}$</td>
<td>1/6</td>
<td>1/3</td>
<td>1/3</td>
<td>1/3</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 10. SV values for $\mathbf{CD}(s) = \mathbf{CP} = \{\text{video service}, \text{audio service}, \text{whiteboard}\}$. 
is higher, i.e., 11/18, as one would expect, since \( SV_T \) is not additive.

However, this fact also suggests that in our setting the notion of additivity needs some additional comments with respect to the classic case of Boolean algebras.

Indeed, in the Boolean case, the lattice-theoretical definition of additivity \( \tau(S \cup T) = \tau(S) + \tau(T) \) for \( S \cap T = \emptyset \) ensures that when disjoint sets \( S \) and \( T \) are evaluated separately, two measure values are obtained that add up to the total measure of their union \( S \cup T \).

In the partition lattice, this is not the case, as the lattice-theoretical join does not express a union: instead, join "merges" two partitions to obtain a coarser one.

This operation should indeed affect query synergy; for this motive, as we have just seen, we have \( SV_T(S' \cup S'') \neq SV_T(S') + SV_T(S'') \).

In other words, the lattice-theoretical notion of additivity is not to be expected, nor it desirable, for a synergy measure.

However, it is interesting to observe that a different notion of additivity can be defined.

In order to do so, given \( B = \{ S', S'' \} \), we write

\[
F(B) = SV(S' \bullet S'') = SV(S') + SV(S'')
\]  

(11)

where operator \( \bullet \) gives the multiset of partition blocks belonging to \( S' \) and \( S'' \), taken with their multiplicity.

Equation (11) contains an apparent abuse of notation, as of course \( S' \bullet S'' \) is not a set of blocks (i.e., a partition) but a multiset; however, it is easy to see that (7) can be computed in this case as well.

Once again, we remark that (11) is not equivalent to assume that our lattice function is additive in the lattice-theoretical sense; however, it gives some justification to the use of the arithmetic mean to compute server fitness with respect to processes.

Moreover, to use the general weighted mean of (8), instead of the arithmetic mean, all that is required is weighting the blocks of each query by the weight associated with the query itself.

Suppose, for instance, that a process is composed of two copies of the query \( S' = \{ AV \}\{ W \} \).

In order to assess a server’s global value for the process using the arithmetic mean we can go through the following procedure: first, we compute (7) on the multiset \( \{ AV \}\{ W \}\{ AW \}\{ W \} \), taking into account the contribution of each block according to its multiplicity; then, we divide the result by two.

In other words, to compute \( F \) we forget about the queries, considering the whole process as a multiset of the partition blocks (where, of course, blocks can be repeated), and compute (7) for each block in the multiset.

For instance, with reference to Fig. 11, supposing \( B = \{ \{ V \}\{ A \}\{ W \}\}, \{ \{ V \}\{ W \}\}, \{ A \}\} \) we get for the servers the following fitness values:

<table>
<thead>
<tr>
<th>OID</th>
<th>Feature1</th>
<th>Feature2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>audio</td>
<td>video</td>
</tr>
<tr>
<td>1</td>
<td>audio</td>
<td>service</td>
</tr>
<tr>
<td>1</td>
<td>service</td>
<td>service</td>
</tr>
<tr>
<td>1</td>
<td>video</td>
<td>video</td>
</tr>
<tr>
<td>1</td>
<td>service</td>
<td>service</td>
</tr>
<tr>
<td>1</td>
<td>whiteboard</td>
<td>service</td>
</tr>
<tr>
<td>2</td>
<td>whiteboard</td>
<td>whiteboard</td>
</tr>
<tr>
<td>2</td>
<td>video</td>
<td>whiteboard</td>
</tr>
<tr>
<td>2</td>
<td>service</td>
<td>service</td>
</tr>
<tr>
<td>2</td>
<td>video</td>
<td>service</td>
</tr>
</tbody>
</table>

Fig. 11. The synergy relation.

As one would expect, the first server shows the best fitness for process \( B \).

The second and fourth server qualify for a close second place \( ex aequo \), as both provide a full match to the requested granularity of one of the two queries composing \( B \).

The third server qualifies at the third place, as it provides no full match; however the process’ granularity requests do not conflict with the server distribution policy, as happens with the fourth server occupying the last place.

This sample ranking of servers could be compared to actual usage patterns measured on the network to assess the compliance of service requests to target distribution policies.

It should be noted that, using a weighted mean, we assign a unique weight to each individual query, regardless of the other queries that appear in the summation.

In terms of the application, this means that each query will contribute a constant synergy value, regardless of the other queries composing the process.

E. Synergy Computation

The computation of (7) can be easily carried out in a relational setting, using a crisp relation \( SYNERGY \), whose schema is \((OID, feature1, feature2)\) (see Fig. 11). The \( SYNERGY \) relation corresponds to the equivalence relations induced in the partition of features exposed by servers.

It is indeed very redundant in space, since it lists each couple of features belonging to each of the \( CD_j(s) \) subsets.

While heuristics can be applied, we will not discuss them here, as space is seldom an issue in a relational repository and this choice makes it easy to check subsets to be disjoint via a relational integrity constraint.

More importantly, the redundancy in \( SYNERGY \) allows the trader to compute the intersection between a user-supplied subset \( CP_i \) and all the subsets \( CD_j(s) \) composing \( CD(s) \) by computing the maximum cardinality of the results of simple relational queries, namely

\[
\tau_{Feature2}(\sigma_{Feature1 \geq x}(SYNERGY))
\]

where \( x \in CP_i \). If more than one pair \((CD_j(s), CP_i)\) give the maximum cardinality intersection, the one with the bigger
is chosen. Then, function $f$ is computed for the chosen pair $(\text{CD}_1(s), \text{CP}_2)$. Finally, results are added up and plugged in (7), to compute the total synergy measure.

**F. A Worked-Out Example**

We shall now refer to Fig. 11 in order to provide a complete example. Suppose once again

$$\text{CP} = \{(\text{audio service}, \text{video service})\}.$$  

Here the user is supplying the information that audio and video services, when taken together, are considered more valuable than either of them alone. Computing (7) gives the results in Fig. 12.

Fig. 12 shows that object 1 is ranked above object 2 by synergy, as one would expect. We observe that, had the user requested all three features together (i.e., had the query contained a single block), he would have got for object 1

$$f(\{(\text{audio service}, \text{video service}), \text{audio service}, \text{video service}\}) = 2$$

and

$$f(\{(\text{whiteboard}, \text{audio service}, \text{video service}), \text{audio service}, \text{video service}, \text{whiteboard}\}) = 1$$

giving a total SV of $1/2$ (remember that the query contains a single block), which is lower than $2/3$. Indeed, one would rather expect so, since in this case the user wants three features together and object 1 only supplies two of them together, while in the previous case the user asked for two features together and object 1 provided them as requested. For object 2, we would get the same result of $1/2$, as expected.

**V. USING SYNERGY FOR SERVER SELECTION**

In this section we outline the integration of the synergy measure presented in Section IV in the framework of the fuzzy trader system described in Section III. Our approach is based on the idea of adding feature subsets denoting synergy as additional features (henceforth, called synergy features) to the fuzzy descriptors $D(s)$. Synergy feature names are given by (encoding of) the elements of the subsets they correspond to. The rationale for this choice is that it does not need to provide a separate status for synergy features. This greatly facilitates the implementation of the model (Section VII). The computation of the synergy features’ membership values will be described shortly.

**A. Preliminary Discussion**

Let us now consider a server offering audio and video sessions over the network at a given level of performance. While not absolutely requiring both services to be provided simultaneously, the organization managing the server considers providing the two services together to be, to some degree, a better option than providing them separately to different clients. This kind of preferences about granularity can be dealt with by extending the model of the descriptor base $\{D(s)\}$ so that it can hold feature subsets. Rather than providing a formal extension to the data model, we shall first appeal to intuition to show how granularity control is performed. Suppose that, to express its preferences about granularity, the server adds to the trader descriptor base an additional synergy feature, namely the subset $\{\text{video service, audio service}\}$. This feature expresses the server’s preferred granularity.

When a client queries the trader, it can itself specify one or more synergy features (i.e., subsets) as an addition to its usual request made of singletons.

These subsets express the client’s requested granularity. Intuitively, it is reasonable that if the synergy features specified by the client correspond somehow to the synergy features specified by the server. Indeed, the server should get an additional chance of being retrieved, as the client and the server are implicitly agreeing not only on individual features, but also on the granularity with which services will be provided. While the division-based framework described in Section 3 can in principle accommodate synergy-type features both in the trader descriptor base and in fuzzy requests, deciding the membership values that should be given to synergy features is not straightforward.

Synergy membership values clearly have a different semantics (namely, one of importance [2]) to the ones computed by the trader for standard features and need to be initialized separately. Classically, importance weights initialization involves polling the user; but a basic requirement of our system is membership values computation, and in general the fuzzy execution engine, to be hidden from servers and clients alike. Thus, we proceed as follows:

1) As far as clients’ requests are concerned, we initialize synergy features’ membership values by taking the minimum of the membership values of their elements. In other words, the added value for the user of getting some features together from the same server is estimated not be higher than the minimum membership value specified for one of them.

2) The weights of synergy features provided by servers in the descriptor base are obtained on-the-fly, using the fuzzy measure of Section IV. Namely, each synergy feature gets, as its membership value, its contribution in (7) to the total SV of the candidate object $s$.

3) Finally, fuzzy relational division is computed as described in Section III.

Step 3 needs some additional clarifications. In Section III, all we needed to do was compute implications between the membership values of features appearing both in the descriptor $D(s)$ and in the query. There was obviously no difficulty with the notion of two singleton features being the same, as string equality was used to compare singletons, seen as strings of the feature language FL. When dealing with synergy features, however, a problem arises, as we need to process the membership values of
each subset in the descriptor with respect to the “corresponding” one in the query. It is easy to see that requiring strict equality of subsets would make the whole computation of synergy membership values rather pointless, as synergy features would be systematically excluded from the division computation. Therefore, when computing the division involving synergy features we use a straightforward weaker notion of feature equality, containing the strict equality used for singletons as a special case. To safeguard efficiency and the use of our algorithm each descriptor synergy feature is considered “equal” to the query synergy feature when . Thus, when checking if a synergy feature in the query is also present in a descriptor, all our naïve algorithm has to do is to use subset inclusion instead of string equality. If features are ordered lexicographically inside subsets, this is simply substring checking and no additional computation is required.

B. A Worked-Out Example

Let us now describe a detailed example. In this example, we set the functional features of object 1 and object 2 to be identical and have the same membership values, so that the different ranking of the objects in the query result is completely due to the synergy features. Fig. 13 shows the tuples of \( D(s) \) associated with object 1 and object 2, augmented by their synergy features (belonging to type \( s \)). Note that the membership values for the synergy features are not specified in Fig. 13. Indeed, they will be computed on the fly, since they change depending on the user query.

It is important to remark again that, if the user chooses not to specify any synergy feature in the query, membership values of synergy features in the descriptor base remain unspecified. In this case synergy is not taken into account in the computation, and we are smoothly back to the scenario of Section III. Fig. 14 shows the positive part of a user query, augmented in the same way as \( D(s) \).

When an augmented user query is submitted to the trader, the synergy measure defined in Section 4 is used to compute on-the-fly the membership values of the synergy features in the descriptor base.

Each synergy feature \( ft \) in \( D(s) \) gets, as its membership value, its contribution to the total SV of the candidate object \( s \), namely

\[
\mu = \frac{f(\mu)}{|CD(s)| + |CP|}
\]

For instance, the membership value \( \mu \) of the \{audio service, video service\} synergy feature of object 1 is computed according to (7) as follows: \( f1 = f(\{audio service, video service\}, \{audio service, video service\}) = 2 \times 2 = 4; f2 = f(\{audio service, video service\}, \{whiteboard\}) = 0 \). Therefore, for \{audio service, video service\} we get

\[
\mu = \frac{f1}{|CD(s)| + |CP|} = \frac{4}{3 + 2} = \frac{4}{5} = 0.8.
\]

On the other hand, the contribution of the synergy feature \{whiteboard\} to the total SV of object 1 (and, therefore, the corresponding membership value) is 0. Once again, we remark that membership values of synergy features are computed on the fly depending on the user query and have therefore to be recalculated for each query submitted to the system. Fig. 15 shows the complete descriptor base for the query in Fig. 14.

Once the synergy features’ membership values have been computed, the query can be processed using the query execution engine described in Section III.

Fig. 16 summarizes the results obtained when executing the query of Fig. 14 on the descriptor base of Fig. 15 for the various semantics, taking into account synergy features.
Inspecting Fig. 16, we see that whenever absorption is required, only object 1 is retrieved. This is due to the synergy feature, since in \( D(\text{object2}) \) there is no \( ft \) such that \( pft \subseteq ft \). We remark that when specifying synergy features, asking for absorption means that a server that did not declare the specified features to be coupled will be excluded from the results.

When absorption is not required, the requested synergy feature, namely \( pft = \{\text{audio service}, \text{video service}\} \) appears with membership zero in the object 2 descriptor, and therefore in the computation of Lukasiewicz implication. As a general comment, we see that while object 1 scores consistently better than object 2, the difference in ranking between the two objects is lower for price without absorption than for performance semantics, as one would expect.

VI. DESIGN AND IMPLEMENTATION GUIDELINES

Up to now, we described a general framework for distributed server selection at a high level of abstraction, without dealing with software design issues. In this section we shall provide some design and implementation guidelines for a trader system based on our approach. Though our techniques do not depend on a specific execution environment, here we shall discuss system design and implementation using the well-known CORBA standard architecture for distributed object-oriented systems [30].

Besides being a nonproprietary and mature technology, CORBA natively supports the concept of distributed services' trading and is therefore well suited to our application. The framework described in this paper is at the basis of the design and implementation of a software prototype [18]. Our current prototype is a trader agent developed (in Java 1.2.2) for the free ORB Jacorb 1.1 [6], available on the Gnu/Linux Red Hat 5.2 platform.

Our trader fully complies with the CORBA 2.0 standard for distributed object interoperability [27]. We are now fine tuning the design of the prototype to support scalability to large scale network environments, while refining the user control on query semantics.

A. CORBA Conceptual Model

Since the early 1990s, the object management group (OMG) has developed a conceptual model for distributed application development, known as the Core Object Model, and a reference architecture, called the object management architecture (OMA) upon which distributed object-oriented applications can be constructed. OMA attempts to define at a high level of abstraction all facilities necessary for distributed object-oriented computing. It consists of four components: 1) a set of object services (OS); 2) the common facilities (CF); 3) the object request broker (ORB); and, 4) the application objects (AO). OS specifications define a set of objects that all CORBA environments should provide. Such objects perform basic functions like naming (used to translate remote objects’ names into valid references), lifecycle management, transactions and trader services. Generally speaking, OS augment and complement the functionality of the ORB, whereas CORBA CF provide other services of direct use to application objects.

The core of the OMA is the ORB component, which is a transparent communication bus for application objects that let them transparently make requests and receive responses from other objects located locally or remotely.

In other words, the ORB allows client and server objects, that can be written in different languages, to interoperate. It intercepts method calls and is responsible for finding an object that can execute them, pass it the parameters, invoke its methods and return the results. Invocations can be done either statically at compile time or dynamically at run time with a late binding of servers.

B. Structure of a CORBA Application

The client side of a CORBA application is composed of IDL stubs (i.e., interface modules generated from standard interface definition language declarations), a dynamic invocation interface (DII), an interface repository, and an ORB interface (left-hand side of Fig. 17). The client-side IDL stubs provide the static interfaces to available object services and define how clients must invoke them. In turn, the DII allows clients to construct and issue the invocation of a method whose signature is unknown until runtime, using information from the interface repository. Clearly, the DII provides a very dynamic environment that allows distributed systems to remain flexible and extensible. The ORB interface (the only component of the architecture shared by both sides), allows ORB functions to be accessed directly by the client code.

The server side implementation of a CORBA application (right-hand side of Fig. 17) consists of IDL skeletons that provide static interfaces to each service exported by the server, a dynamic skeleton interface (DSI), an object adapter, an implementation repository and the ORB interface. The DSI (the server-side equivalent to the DII) looks at parameters values in an incoming message to determine a target object and method.

The object adapter is on top of the ORB’s core communication services and accepts requests on behalf of server objects. It provides the run time environment for creating instances of server objects, passing requests to them and registering their classes in the Implementation Repository.

C. CORBA Trading Service Specification

CORBA dynamic distributed invocation mechanism naturally leads to the problem of dynamic selection of distributed services. How will a CORBA client choose a specific remote
service among those available? The OMG has dealt with this problem by defining a standard interface for a Trading Service. Using CORBA’s Dynamic Invocation Interface, Naming Services, trader Services (together with the Interface Repository), a client application can discover new objects at run time and dynamically invoke their methods, with a late binding of servers. The dynamic identification and invocation of a CORBA object is made in 5 steps:

1) The client communicates to the trader a list of desired properties and associated values. Properties are provided by the CORBA standard as a policy-free mechanism: both their syntax and semantics are left to the implementation.

2) The trader identifies an object offering the service requested by the user on the basis of its properties, and returns a reference to it (called an IOR or Interoperable Object Reference in CORBA terminology) to the client. The identification is usually made by the trader by finding the server object name in its internal data structure, and then using the Naming service to collect the IOR to be passed to the client.

3) Using the Interface Repository, the client retrieves the object interface.

4) According to the interface methods’ signature (i.e., the number and types of arguments), the client constructs the invocation.

5) Finally, the client invokes the server object’s method with adequate parameters and receives the results.

Following current CORBA terminology, in the remainder of this section we shall use the term exporters to designate (potential) callees, i.e., remote objects that implement distributed services and make their interfaces available over the network. The term importers will be used to designate caller objects.

D. Design Problems

A main problem that arises when designing and implementing our service selection framework as a CORBA trader is how to obtain exporters’ functional and nonfunctional features, to be used by the trader as properties. Following [Dam99], we chose to exploit the published IDL interfaces themselves as skeletal descriptions of exporters’ functional properties. In order to support nonfunctional features, we rely on the fact that CORBA exporters may also provide dynamic properties, i.e., policy-free text descriptions whose values are made available to importers invoking their methods.

Our trader straightforwardly uses CORBAs naming service to keep track of all the exporters registered in a given namespace (called Service) and exploits their IDL interfaces and dynamic properties to obtain the functional and nonfunctional properties needed to process the importers’ requests. Another classic design problem for CORBA traders is how to provide importers with the system’s vocabulary, i.e., with the lexicon of the feature language FL, as shown in Fig. 18. In our prototype, this language is simply the set of strings used for the services’ IDL interfaces and dynamic properties definition. Without such a provision, a run-time Thesaurus [Dam95] would be needed to deal with synonymy in feature names. In our design, importers may query the trader for the whole FL vocabulary, i.e., for the list of the available services’ feature names. While our current approach has the merit of being simple to implement, it assumes a system-wide naming discipline that may well not be realistic in all cases.

E. Trader Operation

We are now ready to outline our prototype’s operation. Our trader’s initialization and startup consists of three steps.

1) Exporters register their services with our trader, communicating their functional and nonfunctional properties as specified in their IDL interfaces and dynamic properties. This step strictly complies with CORBA Trading specification, as exporters register properties’ names and the associated crisp values. Fuzzification and update of the trader’s fuzzy table are dealt with internally by the trader. Dynamic properties’ values are to be updated periodically.

2) Importers get a reference to our trader from the ORB. Note that, according to the OMG standard, the trader’s OID is defined by default by the ORB and is therefore accessible to all importers.

3) Importers are now enabled to query the trader. After initialization, our trader table has the following structure.

The dynamic label in the above table is simply a reminder of the fact that while the first two properties are functional ones, extracted once and for all by the server IDL interface, the third property is dynamic and may therefore vary. Our trader periodically checks in with the exporters in order to update values associated with their dynamic features. There are two possible mechanisms for doing this, according to the applications’ requirements: by periodical polling (as in our current prototype) and by leaving it to the exporters to use CORBA asynchronous event channel interface to communicate with the trader. This second solution, though it provides better performance, would require exporters internal logic to be aware of the trader operation and was, therefore, discarded.

When a query is presented to the trader, it consists of a “special” property-value pair specifying the desired semantics (e.g., semantics = price) followed by a list of the properties the server should possess, together with the associated crisp values. The “special” property-value pair triggers a unknown property Java exception that converts the query in a fuzzy request (using the fuzzy predicates stored inside the trader) and executes it according to our model. Note that this technique allowed us to retain CORBA standard trading interface; importers unaware of the trader “intelligent” capabilities simply do not prefix their query with the “special” pair, the Java exception is not raised and our prototype will behave as Jacob’s standard trader, based on crisp keyword matching. Finally, we observe

<table>
<thead>
<tr>
<th>IOR</th>
<th>Method</th>
<th>Param</th>
<th>Type</th>
<th>μ</th>
</tr>
</thead>
<tbody>
<tr>
<td>4349</td>
<td>Read</td>
<td>int</td>
<td>F</td>
<td>0.53</td>
</tr>
<tr>
<td>4349</td>
<td>Write</td>
<td>int</td>
<td>F</td>
<td>0.2</td>
</tr>
<tr>
<td>4349</td>
<td>Lag</td>
<td>int</td>
<td>NF</td>
<td>dynamic</td>
</tr>
</tbody>
</table>

Fig. 18. A sample trader table in our prototype.
that, since our model deals with synergy features without giving them a separate status in query formulation and execution, synergy-based computations are straightforwardly executed by our trader internal logic.

F. Internal Architecture and Performance

Our trader implementation is made on top of Jacorb standard trader, which follows the OMG suggested design pattern [7]. A simplified version of the trader architecture is given in Fig. 19, also showing the five steps of the trader’s operation. Fig. 19 shows the tight architectural coupling between our trader and CORBA naming service, according to the OMG specification.

As far as the overall performance of our implementation is concerned, it should be noted that CORBA dynamic invocation is usually much slower than the static one. To a certain extent, this is an unavoidable part of the “speed versus flexibility” bargain in distributed systems. However, dynamic invocation is dealt with differently by individual CORBA implementations, and experiments have shown that the performance gap does vary a lot across implementations.

Also, total response time for remote method invocation is heavily affected by the parameters’ datatypes, which may require intensive marshalling.

Finally, being the trader a standard CORBA service whose IOR reference is known to importers, an invocation to the trader itself may be likened to a static call.

Thus, we did not try to assess the total time required to follow steps from 1 to 5 of Fig. 19. Rather, we tried to single out the performance one should expect from traders designed according to our model.

However, the CORBA standard mandates the trader’s interface and not its implementation; moreover, some free CORBA environments do not provide a trader service at all, while commercial ones may include alternative implementations. This makes it somewhat tricky to compare the actual performance of our intelligent trader prototype to any “reference” implementation.

Fig. 20 gives an idea of the very small overhead introduced by our fuzzy-matching trader prototype when queries are reasonably small.

Results are compared to the standard trader provided by the Jacob system. Query size is reported as a percentage of the maximum size used, i.e., 10 000 B.

In both cases, the trader internal data contained a single match for the query. Delay is measured in “tics” (1 tic = 10 μs). The minimum value (a little more than 10 ms for 1000 B) is more or less the “ping time” expected from a static call with practically no marshalling.

The trader response time is more or less linear with respect to the query size, and the matching technique influence becomes apparent only for query sizes greater than 900 B, where the standard technique very slightly outperforms fuzzy matching. The measurements were conducted under Gnu/Linux Red Hat 5.2 on a 500 MHz PC with 128 MB RAM.

VII. RELATED WORK

In the soft computing research community, many measures of comparison of descriptions of objects have been proposed and studied in several application domains.

Bouchon-Meunier, Rifqi, and Bothorel [5] proposed a general classification of measures of object comparison, depending on the purpose of their utilization. Namely, they defined measures of \textit{satisfiability}, \textit{resemblance} and \textit{comparison} that can be considered as measures of similarity and on the other hand, as measures of dissimilarity.

In the rough-set area, researchers defined similarity between objects (described as sets of attributes) by extending the \textit{indiscernibility} relation used for the definition of the rough set concept [31].

Interestingly, that approach involved separate definition and processing of \textit{concordance} and \textit{discordance} attributes.

Though profoundly different, all these approaches tried to provide a sound mathematical basis to the concept of similarity between abstract notions of objects.

At the same time, the software engineering community was independently pursuing a line of research aimed at computing similarity between object-oriented software components.

Software engineering researchers had to solve the two related but distinct problems of objects classification and selection to attain repository-based reuse of source-level components, itself a much investigated subject [1].

Software components’ repositories based on a fuzzy-relational data model were first proposed in [28], while program
databases in a conventional relational setting are described in detail in [29].

In [14], fuzzy similarity was used as the basis of the retrieval mechanism for a repository storing reusable components. A different technique, based on an analogy paradigm is presented in [23], while in more recent research [25] specifications were used to drive components' selection.

In [26] a complete survey of specification matching for software components is presented, describing the process of determining if two software components are interchangeable for a number of application development purposes, such as software reuse or system reengineering. Most software engineering approaches do not deal with feature interdependency in object selection. However, in [36] the features of software components are interrelated by exploiting various kinds of relationships, forming partially ordered sets.

In [13] a fuzzy query language the component query language (CQL) was used as the basis for an industrial-scale software development environment, using fuzzy linguistic variables instead of numerical weights in order to facilitate user interaction. A standard Database Management System (DBMS) is used to store descriptors of available components and integration with the development environment is achieved.

In [15] a set of fuzzy techniques is described for classifying components according to their features. The approach involves a similarity measure based on the number of occurrences of a feature name in source code. This measure as an estimate of the feature relevance to the description of component behavior, and, therefore, of its membership function in a fuzzy descriptor of such behavior.

Still, until recently virtually no attempt was made by software engineers to deal with the problem of intelligent server selection at run-time.

In the last few years, several proposals were made for increasing transparency and efficiency of access to O-O servers using CORBA traders [9]; however, those proposals focused on query routing and trader federation rather than on selection techniques. In the active networks scenario [34], clients choose server components at execution time and nonfunctional features such as price may play an important role in component selection.

The techniques for online selection of services presented in this paper are the counterpart of the offline classification and retrieval methodology enabling reuse of source level O-O components presented in [14].

Our research started with a preliminary version of our fuzzy service selection framework [12], [17] where independent features were mapped into fuzzy linguistic variables rather than predicates. Such framework was instrumental to assess how well classical implications could match user-provided semantics. The prototype described in [17] focused on a user interface allowing humans, rather than client agents, to communicate with the trader and inspect query results. Experience with that prototype pointed out that synergy between related features must be taken into account.

The framework presented in this paper is specifically aimed at addressing this problem.

VIII. CONCLUSION

The present paper described a framework for run-time server selection, relying on a query model based on fuzzy sets theory. Our model allows for specifying the intended semantics of the application through different choices of the implication operator to be used in query computation.

In future work, other types of interdependencies between servers' features will also be taken into account. Finally, domain-specific analytical queries based on fuzzy integrals [21] on the partition lattice are under investigation.

ACKNOWLEDGMENT

The authors would like to thank D. Rine for many interesting conversations on fuzzy classifications of software components. Thanks are also due to L. Le Nedelec and O. Scalia for their competent work on the fuzzy trader implementation.

REFERENCES

He is also a member of the Editorial Boards of *Fuzzy Sets and Systems* and the *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*. 

Ernesto Damiani (M’95) received the Ph.D. degree from Universit di Milano, Polo di Crema, Italy, in computer science, in 1994.

Currently, he is an Associate Professor of Computer Science with the Universit di Milano, Italy. He has been a Visiting Professor with George Mason University, Fairfax, VA and with the LaTrobe University, Melbourne, Australia. He is author of more than 70 research papers in international journals and in proceedings of international conferences. He is coauthor of *Intelligent Multimedia Multi-Agent Systems* (Norwell, MA: Kluwer, 2000). His research interests include software engineering, distributed architectures, semi-structured information processing, and soft computing.

Dr. Damiani is Vice Chair of the ACM-SIGAPP and serves on program committees of several international conferences, including the ACM Symposium on Applied Computing. He is a member of the Academic Steering Committee of the BISC Special Interest Group on Fuzzy Logic and the Internet.

Mariagrazia Fugini is currently an Associate Professor of Computer Science with the Politecnico di Milano, Italy. From 1987 to 1993, she was the Italian Representative for the IFIP-TC 11 Committee on Database Security. She has participated in several EU-ESPRIT Projects on Information System Security, Software Reuse, Requirements Engineering, and Workflow Design. She has coordinated projects sponsored by the National Research Council on Information Systems and Active Networks and has been involved in Development and Training Projects regarding Information Systems and Data Security in private and public organizations. She has also been a Visiting Professor at the University of Maryland, in Vienna and Stuttgart, Germany. She is coauthor of *Database Security* (Reading, MA: Addison-Wesley, 1995). Research interests include information systems design and reengineering, information systems security, and document management and retrieval.

Patrick Bosc received the Ph.D. degree in computer science from the University of Rennes, France, in 1978.

He joined the Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA) and is currently a Professor at École Nationale Supérieure de Sciences Appliquées et de Technologies (ENSSAT). His main research interests concern the roles of fuzzy sets and possibility theory in database management systems. Prof. Bosc is a member of the EUSFLAT Association aimed at the promotion of fuzzy sets in Europe.

He is also a member of the Editorial Boards of *Fuzzy Sets and Systems* and the *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*. 


Patrick Bosc

Ernesto Damiani

Mariagrazia Fugini