Abstract

Intellectual Property (IP) reuse is one of the most promising techniques addressing the design complexity problem [1]. IP reuse assumes that pre-designed components can be integrated into the design under development, thereby reducing design complexity and time. On the other hand, as the number of IP providers increases, the selection of the best IP block for a given design becomes more challenging and time-consuming. In this paper, we present an IP component matching system targeting automatic component searching and matching across the Internet. The system is based on Extensible Markup Language (XML) specification both for IP libraries (a repository of pre-designed IP components indexed by their corresponding specifications) and an IP user queries (specifications with incomplete/uncertain attributes). An IP query is parsed into a document object model (DOM) and the DOM is transformed to an internal tree-structured model. Fuzzy logic scoring and aggregation algorithms are applied to the internal tree structure to provide a set of candidate approximate matches ranked by proximity between the query and IP specification.

Component Selection and Matching for IP-Based Design

1 Introduction

Current silicon manufacturing technology allows entire systems to be built on a single chip (SoC) as a more cost-effective solution than multi-chip systems. However, the increase of design engineers' productivity hardly keeps pace with the increase of complexity. Intellectual Property (IP) reuse is seen as a viable approach to close the "design productivity gap". IP reuse assumes that pre-designed, pre-synthesized and pre-verified components can easily integrated into a design, hence reducing design complexity and time. A prerequisite for successful integration is IP *component selection*: the IP-based design flow needs tools to facilitate the process of selecting the component with both functionality and cost that optimally match the designer's needs. Current IP publication and search techniques remain at the informal level of Internet-based message boards ([2], [3]). Some IP reuse tools have been proposed ([4], [5], [7]); however, they do not address the component selection problem.

In this paper, we present a component matching system, targeting automatic component search via the Internet. The system is part of the JavaCAD framework, an Internet-based EDA tool aimed at providing IP protection in simulation and cost estimation involving IP components ([6]). Our contributions are:

1. *IP component specification in XML.*
The system employs the Extensible Markup Language (XML) as a representation both for IP providers building a database (a repository of pre-designed IP components indexed by their specifications) and IP users expressing a query (a particular specification with incomplete/uncertain attributes).

2. *IP component selection based on fuzzy logic.*
IP query and specification are parsed to the Document Object Model (DOM), which in turn is transformed into an internal tree structure. Fuzzy queries are represented as *fuzzy trees* (i.e., trees with fuzzy objects as nodes) and specifications are represented as *crisp tree* (i.e., trees with crisp objects as nodes). A fuzzy tree is essentially a fuzzy set, and matching computes membership degree (proximity) of a crisp tree in this set. The computation is split into two stages: atomic scoring and bottom-up conjunction aggregation which is the combination of weighted and unweighted aggregation.

The reasons for the adoption of a fuzzy search engine can be summarized as follows. First, fuzzy logic allows designers to specify selection criterias (queries) that are partially defined and subject to later refinement. For instance, the designer may not entirely sure on what tradeoff between speed and power will suit her/his needs. Additionally, initially vague specifications may be iteratively refined (i.e., made crisper) as the design space becomes increasingly constrained. Finally, fuzzy matching is applicable to queries that are inherently ambiguous (such as informal functional requirements).

Section 2 introduces how IP component matching should apply in the design practice. We view matching as a two-step process: selection on design metrics and functionality checking. Section 3 presents the software architecture of the component matching system, including workflow and implementation overview. Eight main interfaces and related methods are given. Section 4 introduces XML as an IP content representation language between IP providers and users. Advantages of this type of representation are discussed in detail. Section 5 describes how fuzzy matching algorithms are used to rank crisp component specifications according to a fuzzy query specification. For each step, the requirement of corresponding algorithms is discussed and algorithms are given. Section 6 presents experimental results demonstrating that our system can clearly reflect the intuition of "proximity matching".

To give the reader concrete examples of use of our system, we carry out a case study on two DSP components, Discrete Cosine Transform and its inverse (DCT/IDCT). Our choice is based on the fact that DCT/IDCT have broad applications in audio/video/image compression and have many alternative implementations.

2 Methodology

IP component matching is an iterative process that can be split in two phases: selection on design metrics and functionality checking.

1. Selection on Design Metrics.
At this step, a designer is not concerned with rigid matching. In fact, a precise specification has not been finalized. Similarly to library searching in software development, the designer wants to find some component that fulfills the target functionality. The goal is to narrow down the search scope first. The designer is willing to accept some tune-up of the design to accommodate IP components. In particular, the designer tries to collect a set of component candidates which:

- belong to the category of interest and nominally implement the required functionality,
- have a compatible interface with the design under development such as a port map, etc.
- satisfy requirements on design metrics such as area, power, throughput, timing constraints and testability.

Property (3) is obviously fuzzy in nature. A designer usually has room to make tradeoffs between the component and its design environment. Properties (1) and (2) are crisper. Nevertheless, some latitude is allowed in these two conditions.

For instance, if a designer wants to select a DCT hardware, he/she may be adamant in requiring more than 12 bits data precision, but may not care if precision is 16 or 18 bits. Functionality redundancy is tolerated as long as the design cost increase is restricted within a reasonable scope. Similar situations exist for port interface. For example, the port map of any DCT component candidates should contain a DataIn and DataOut vector with desired length. However, it may not matter how many other control signals should be provided and how they are arranged. As long as the component behaves consistently with specification, the designer will tune up the development environment to embed the component. In summary, in this stage, all matching criteria have uncertain properties which are ideally suited to the application of fuzzy logic in the search process. Section 5 will address the problem in detail.

2. Functionality Checking. After a set of potential candidates are selected, it is time for the designer to carry out precise functionality matching. This step usually relies on simulation. The designer feeds the component some input bit patterns and observes outputs to see whether there is a discrepancy with the desired functionality. Take DCT functionality checking as an example. The designer feeds each candidate component a matrix of image pixels (e.g. 8x8) and tests if the output matrix contains the expected values. Although this process does not complete the validation, the designer minimizes a lot of "false positives", which avoids time-consuming manual testingwith simulation within the design.

In this paper, we concentrate only on the first phase of component matching. Our approach helps speeding up the initial selection process. The matching systems is divided in two parts: (1) XML representation of design specification (described in Section 4) and (2) fuzzy matching between tree structures (see Section 5).

3 Software Architecture

Our matching system is contained in the JavaCAD framework, an Internet-based EDA tool with a secure client/server architecture, providing IP verification and estimation via the Internet without revealing IP details before purchase ([6]). The communication between IP provider and user is through the Remote Method Interface (RMI) and a web server using HTTP.The overall system architecture is shown in Figure 1.

Figure 1 focuses the part of the JavaCAD infrastructure which is most closely related with component matching. This part is seamlessly integrated with the simulation process by allowing Virtual Components (VC) in the simulation setup. When the simulation enters a VC, the component matching is

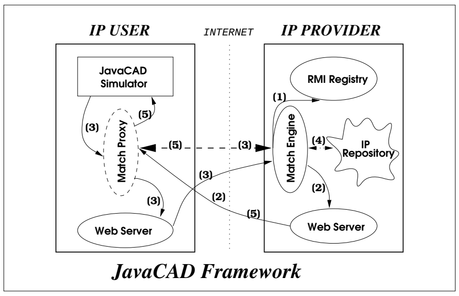

Figure 1: The IP Component Matching Architecture

triggered to search databases of the IP provider via the Internet. After searching, the candidates with the nominal functionality are ranked according to design metrics. The behavioral specification (in Java) of these components will be shipped back to the IP user. Then the JavaCAD simulator will automatically instantiate the VC with the behavioral codes and resume the simulation process. More precisely, matching is a five-stage process.

• *Match Engine* is the set of matching logics in a server side (an IP provider) that interacts with its IP database. It registers itself with the RMI Registry server which provides the naming service for remote proxy objects (1).

• After the RMI naming service is set up, the client (the IP user) has obtained the virtual proxy of *Match Engine* shipped to its location using HTTP through the web server (2). Once the proxy, (the *Match Engine*), is setup at the client side, it accepts client queries which are specifications with uncertain attributes.

• The specifications are either pre-compiled Java objects loaded from files, or URL links to XML documents which are parsed to the Document Object Model(DOM) (see Section 4). During this phase, specific fuzzy scoring methods and aggregation algorithms are plugged into the DOM tree, forming the final query (3) (see Section 5).

• Transparently to the end user, fuzzy matching in *Match Engine* is run at the server side. Matching retrieves the desired IP components from the repository (IP database) and ranks them according to the design metrics of the fuzzy query (4).

• Once the matching engine selects a set of candidates, the designer instantiates the component in the design. The matched component are shipped to the client side and instantiated (5).

As is noted in Section 2, the above process is incomplete due to lack of functionality checking. Once the component candidates are shipped to the client side, the user can either choose formal methods (like model checking) or simulations in order to determine which component should be incorporated into the design under development.

Our design principle aims at providing matching algorithm independence and extensibility. To fulfill this requirement, we represented every key concept of IP reuse (e.g., *specification, component, implementation* etc.) in a Java interface. In particular, the two basic algorithmic components of the matching engine, *atomic scoring* and *conjunction aggregation*, are defined by the interfaces *Scorer* and *Aggregator* respectively (see below and Section 5). Users are allowed the maximum freedom to choose a suitable domain-dependent algorithm as long as these algorithms satisfy the two interfaces.

In the following paragraph, we briefly review the eight main interfaces of the component matching system, and outline their relationship. Also, representative methods for each interface are listed.

• *DOMTransformable*: the interface for all XML representable objects in the system, including *Specification*, *Implementation*, *Component*, and *Repository*. These types of objects are constructed in three steps: (1) representing an IP as XML document, (2) parsing the XML document to a DOM tree (syntax tree), (3) transforming the DOM tree into an internal tree structure (semantics tree).

• *Specification*: defining all standard behaviors of an IP specification. The key method is

$$public\ abstract\ double\ match(Specification\ spec);$$

which gives a matching score between *this* (querying specification) and *spec* (queried specification). Another important method is

$$public\ abstract\ void\ setScorer(Scorer\ scorer);$$

which makes it possible to plug different scorers into the specification tree, providing algorithm extensibility.

• *Implementation*: representing module implementations. Behavioral code in JavaCAD is represented by its corresponding top level *Class* object.

• *Component*: gluing together an IP specification and its corresponding implementations.

• *Repository*: organizing an IP database as a *Map* from index *Specification* to *Component*.

• *Scorer*: describing abstract behavior for all scoring algorithms including the atomic scoring and aggregation. The key method is

$$double\ score(Specification\ spec1, Specification\ spec2);$$

which takes two specifications (*spec1* and *spec2*) and returns their proximity. Any algorithm satisfying this interface can be plugged into the query using *setScorer()*, providing algorithm independence. During matching, *score()* method will be called inside *match(spec)* method with parameters *this* and *spec*.

• *Aggregator*: sub-interface of *Scorer*, containing the method of merging a set of pre-computed scores to a final one.

$$public\ abstract\ double\ aggregate(List\ scorers);$$

• *Query*: wrapping up a particular IP specification (fuzzy query) which contains incomplete/uncertain attributes.

The architecture is open in the sense that the IP user freely chooses scoring functions and aggregation algorithms, and plugs them into the fuzzy query tree. These searching functions and algorithms are shipped to the server side with the query itself. When a query is carried out, the search engine performs general computation on the IP database using those functions and algorithms. Therefore, IP users have the flexibility to adjust strategy according to previous search performances. If a user does not want to customize query algorithms, default ones are used.

4 XML Representation of IP Specification

IP Specification is a set of heterogeneous information varying drastically from component to component. It is not feasible to use fixed schema to describe these loosely structured (or semi-structured) data. In our system, we adopt XML as an IP content representation interface between IP providers and users. XML is a metalanguage for creating domain-dependent markup languages. In contrast to other markup languages (e.g., HTML), XML does not target visual presentation of data. Rather, it describes the logic structure of data using a BNF-like grammar called *Document Type Definition* (DTD). For example, Table 1 shows a DTD description of a component specification. The description is an excerpt from a DTD of component repository whose corresponding DOM tree is shown in Figure 2.

The following are three important syntax definitions in DTD.

• $<!\ ENTITY\ \mathbf{E} \ldots >$ denotes **E** an individual item which is included by referencing its name, similar to the macro definition in programming languages. Since every tag has an ID, we define a *global-id* entity and reference it in all other definitions.

• $<!\ ELEMENT\ \mathbf{T} \ldots >$ defines the syntax of tag **T**. For example,

```
<!-- DTD for ip_specification version 1.0 -->
<!-- parameter entities: -->
<!ENTITY % global_id ''ID ID #IMPLIED''>
<!-- matching scorer declaration:-->
<!ELEMENT ip_scorer (parameter)*>
<!ATTLIST ip_scorer name  CDATA  #REQUIRED>
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
  type  CDATA  #REQUIRED
  value  CDATA  #REQUIRED>
<!-- value declaration:-->
<!ELEMENT value (#PCDATA)>
<!-- ip specification declaration:-->
<!ELEMENT ip_specification
  (ip_scorer?, area, power, clock,
   throughput, portmap, timing, description)>
<!ATTLIST ip_specification %global_id;>
<!ELEMENT area (ip_scorer?, value)>
<!ELEMENT clock (ip_scorer?, value)>
<!ELEMENT power (ip_scorer?, value)>
<!ELEMENT throughput (ip_scorer?, value)>
<!ELEMENT portmap (ip_scorer?, port+)>
<!ELEMENT port (ip_scorer?)>
<!ATTLIST port %global_id;
  name  CDATA  #REQUIRED
  mode  (in | out | inout | clock)  #REQUIRED
  size  CDATA  #REQUIRED>
<!ELEMENT timing (ip_scorer?, port_timing+)>
<!ELEMENT port_timing (ip_scorer?)>
<!ATTLIST port_timing %global_id;
  name  CDATA  #REQUIRED
  setup CDATA  #IMPLIED
  hold  CDATA  #IMPLIED>
<!ELEMENT description (ip_scorer?, value)>
```

Table 1: The Data Type Definition (DTD) of a Generic IP Specification

```
<!ELEMENT ip_specification
  (ip_scorer?, area, power, clock,
   throughput, portmap, timing, description)>
```

defines a tag *ip-specification* whose corresponding object is composed from sub-objects tagged with *area*, *power*, etc. Note that *ip-scorer?* means the tag *ip-scorer* is optional.

• <! *ATTLIST* **T** . . . > defines the attribute list of the tag **T**. Each attribute is defined by a tuple $< N, T, S >$ where N is the name of the attribute, T is the data type, and S is the source of the attribute value, i.e., if the value is given explicitly or implied from the context. The following example gives the attribute list definition of element *port-timing*.

```
<!ATTLIST port_timing %global_id;
  name  CDATA  #REQUIRED
  setup CDATA  #IMPLIED
  hold  CDATA  #IMPLIED>
```

The above definition says that the tag *port-timing* has four sub-attributes. The first one is an external reference to the pre-defined entity *global-id*. The symbol % is used for de-referencing an entity. The second attribute is *name* whose type is *CDATA*, requiring an explicitly supplied value. *CDATA* is an acronym of "Character DATA", which means "interpreting these characters literally". The third and fourth attributes are *setup* and *hold* whose values are implied from the context, and are usually optional.

DTD defines the syntax tree of an instance of XML language. In XML vocabulary, this syntax tree is called a DOM tree. Figure 2 shows a DOM tree of a generic IP repository.

XML has several advantages as a representation language:

1. XML keeps SGML expressiveness power on the structured data without the SGML complexity. The hierarchical topology of data organization reflects the logic structure of data and reduces access steps to sub-elements. For example, consider a specification of DCT/IDCT. It contains area, power, clock and throughput as numeric metrics. However, the port map is more complicated, because each port may have different lengths, directions and timing constraints with respect to other ports. Moreover, even a seemingly numeric metric is not necessarily atomic. For example, the throughput can be measured in various ways, namely by clock cycles per pixel block/frame, or by processing time per frame. Hence, it is necessary for description language to express a hierarchical structure.

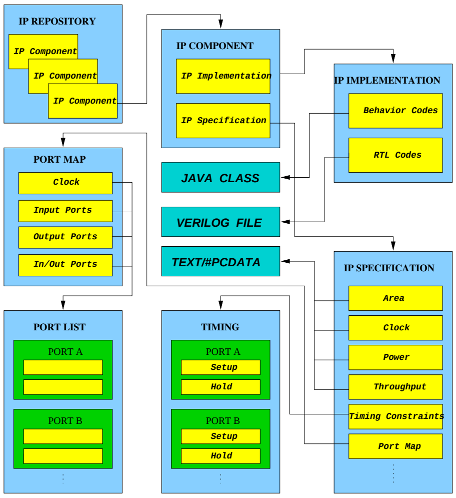

Figure 2: The DOM Tree of a Generic IP Repository

2. The tagged data embeds the structure information within the data which makes the processing easier. In particular, inline DTD makes the universal syntax parser applicable, giving flexibility to the document creator and interpreters (applications) on data organization and visual representation. As the semantics model of XML, DOM is a tree-structured object graph (with no links) which is easily extracted and updated by application, avoiding ad hoc processing codes.

3. XML provides the ability to stylistically customize the document, which makes it an excellent candidate for the next generation of web languages. Both the Extensible Stylesheet Language(XSL) and the Extensible HTML (XHTML), as a recast style sheet and markup language (respectively) in XML vocabulary, are becoming standard recommendations. In addition, stylistic customization provides a pleasing type-setting ability of IP specification publications on the web as well as on paper. Actually, the Apache Software Foundation is developing an XSL Formatting Object Processor (FOP), which reads a formatting object tree and then turns it into a PDF document.

4. XML is becoming a new data organization model (vs the traditional relational one) for semistructured data. For example, the path length between two objects in a DOM tree are constructed to be a metric of some relationship such as similarity. Consider a IP repository of DSP components. The component DCT/IDCT, FIR/IIR (Finite/Infinite Impulse Response Filter) is organized within a subtree such that the distances between these components reflect their closed relationship in the application domain. A user who is searching DCT may be interested in similar components as well. In fact, this property has been explored in building proximity query techniques in semi-structure based lightweight DBMS Lore ([8]).

In our system, XML is used in building an IP database for a provider as well as in constructing an IP query for a user. There are two steps in processing XML representation.

1. An XML document is parsed to generate a DOM tree, viewed as the corresponding syntax tree.

2. The DOM tree is further transformed to an internal tree structure in JavaCAD with roughly the same topology, viewed as the corresponding semantics tree.

After the semantic tree construction, the next step is to apply fuzzy logic to compute the proximity between semantic trees(see Section 5).

5 Fuzzy Matching of IP Specification

IP matching is intrinsically different from the traditional database searches in the query semantics. In the traditional relational database system, the expected query result is a crisp set. Although the result may be sorted according to some fields, it is essentially unordered. However, in the settings of IP component matching, the IP specification is hardly finalized at this design phase. Variation and tradeoff should be allowed in a reasonable scope. Due to the fuzzy nature of query specifications, it is desirable to have the search result ranked according to the query. Moreover, users should be able to express special interests on some particular attributes. For example, if a user is designing a portable still camera, he/she would be stringent on power consumption while relaxing performance requirements since it is unlikely to have a sustained input stream.

Several approaches were evaluated before we adopted fuzzy logic. One is the neural net which is widely applied to pattern matching and classification. However, neural nets needs a moderately large and typical dataset for training while IP specifications are loosely structured with broad variation spectrum. Moreover, dataset training is computation intensive even for the obvious pattern classifications. Therefore, the neural net is not well-suited in the domain of IP component matching. On the contrary, fuzzy logic is widely used in the domain of "approximate matching" ([12])

Recall from Section 4 that an XML specification is parsed to generate a semantic tree. In the tree, nodes correspond to attributes in an IP specification. In our fuzzy logic solution, every attribute (the node in a query semantic tree) is viewed as a fuzzy mathematical object. In contrast, attributes of the component specifications in IP providers' database are crisp. We call *fuzzy tree* any semantic tree containing fuzzy objects as nodes and refer similarly to a *crisp tree*. Figure 3 shows a simple DCT specification tree and a fuzzy query tree.

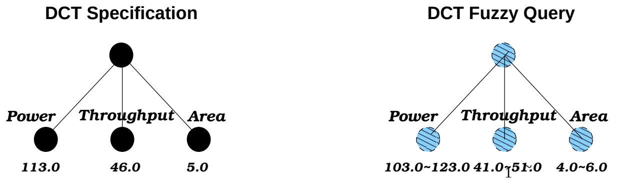

Figure 3: A Specification Tree vs a Fuzzy Query Tree

The fuzzy tree is essentially a fuzzy set, and matching computes the membership degree of a crisp tree in this set. Our computation divides into two stages: (1) an atomic scoring leaf nodes (in Figure 4), (2) a bottom-up conjunction aggregation (in Figure 5). More in detail, conjunction aggregation is a combination of two distinct steps, an unweighted conjunction aggregation and a weighted conjunction aggregation.

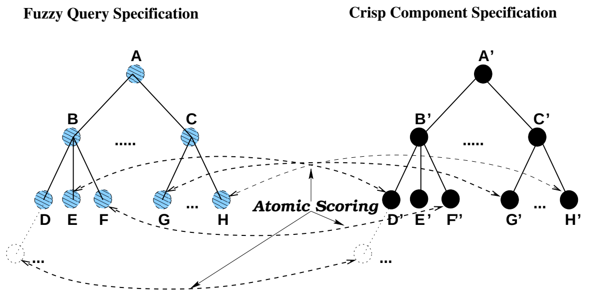

Figure 4: The Atomic Scoring Between a Query Tree and a Specification Tree

5.1 Atomic Scoring

Each of the atomic attributes in a query specification (i.e., leaf node in a semantic tree) is viewed as a fuzzy object. For example, numeric values of area, power, and throughput are modeled as fuzzy numbers. Figure 6 illustrates the geometric representation of a triangular fuzzy number.

This triangular fuzzy number is a fuzzy set over universe $\mathcal{R}$ with only one crisp point as its core. Equation 1 gives its corresponding membership function.

$$\mathcal{A}(t) = \begin{cases} 1-(a-t)/u & \text{if } a-u \le t \le a \\ 1-(t-a)/v & \text{if } a \le t \le a+v \\ 0 & otherwise \end{cases} \quad (1)$$

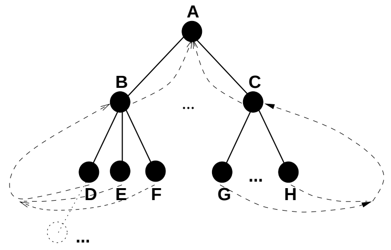

Figure 5: The Conjunction Aggregation on a Fuzzily Scored Specification Tree

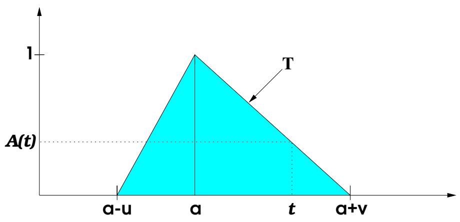

Figure 6: A Triangular Fuzzy Number

Given a corresponding crisp number t_0, the membership function value $\mathcal{A}(t_0)$ is a confidence degree (or membership degree) with respect to the fuzzy number.

For example, assume we have a DCT query specification in which the throughput is characterized by the fuzzy number (above) with the parameters

$$a = 46, u = 5, v = 10$$

The following list gives three crisp values and their respective membership degree to the above fuzzy number.

Value 47.0	Score $= 0.9$
Value 44.0	Score $= 0.6$
Value 52.0	Score $= 0.4$

There are many other possible choices of membership functions. Figure 7 shows a trapezoidal fuzzy number whose membership function is summarized in Equation 2.

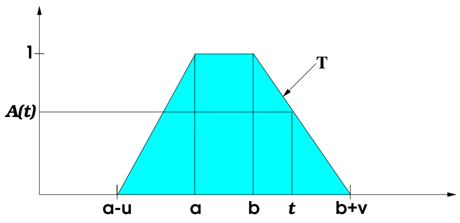

Figure 7: A Trapezoidal Fuzzy Number

$$\mathcal{A}(t) = \begin{cases} 1-(a-t)/u & \text{if } a-u \le t \le a \\ 1 & \text{if } a \le t \le b \\ 1-(t-b)/v & \text{if } a \le t \le b+v \\ 0 & otherwise \end{cases} \quad (2)$$

In general, any real continuous function whose value domain is $[0, 1]$ could be a membership function, although continuity is usually required. Otherwise, membership degenerates to a characteristic function in traditional (crisp) set theory, which is viewed as a special fuzzy membership function that maps the core to 1 and others to 0. There is not a rigorous mathematical reason for the choice in a fuzzy membership function, the choice depends on the application domain. Suppose we search for a DCT component and requires a clock frequency within the range of $60-100$ MHZ. The perfect matching may be $70-90$ MHZ within which we have no preference at all. In this case, it is more appropriate to use the trapezoidal fuzzy number defined in Equation 2 with the parameters

$$a = 70, b = 90, u = 10, v = 10$$

In practice, it is enough for a numeric value to satisfy either lower bound or upper bound, but not necessarily both. Hence, we usually set $u = a$ or $v = \infty$.

The fuzzy object concept is not restricted to a number. It can be generalized to traditional mathematical objects such as vectors, geometric regions, functions, and predicates ([13]). In fact, we also view functionality as a fuzzy object with the membership function based on keyword occurrence probabilities. In particular, keywords are assumed to appear independently in the description text. The individual keyword membership degree d is a function of the occurrence probability p and occurrence time n,

$$d = f(p, n) = 1 - p^n \tag{3}$$

For example, consider the following functionality description provided by an IP provider.

```
<description>
    S_DCT_IDCT performs the two dimensional Discrete
Cosine Transform (DCT) and its inverse (IDCT) on an 8x8
block of image samples ...  DCT ...  IDCT ...
</description>
```

Suppose an IP user uses the keyword "DCT" in a query and assumes the occurrence probability of "DCT" p_{dct} is 0.5. From the above description text, we have $n_{dct} = 2$, and hence

$$d_{dct} = 1 - 0.5^2 = 0.75$$

In the implementation, a scoring function (membership function) is associated with each leaf node, either by default or the user assignment. The function scores each crisp atomic counterpart in the component specification, as is shown in dash lines in Figure 4.

Tree matching does not necessarily require two trees with the same topology. Usually the query specification is more abstract than the component specification which may have collected many more details during the synthesis phase. Hence, usually the fuzzy tree is topologically embedded in the crisp one. However, the fuzzy tree could have some branches missing within the crisp one, as IP users may be concerned with some features not conceived by IP providers. In this case, atomic scoring returns 0.

5.2 Conjunction Aggregation

After having the atomic attributes (in leaf nodes) scored, the next step is to compute the scores of the intermediate nodes. This step is called conjunction aggregation. The term "conjunction" reflects the aggregation semantics, roughly a complex attribute with a "high" score if all the sub-attributes score well. The semantics is expressed through the folowing properties:

$$A(x, y) = A(y, x) \tag{4}$$
$$A(A(x, y), z) = A(x, A(y, z)) \tag{5}$$
$$A(x, y) \leq A(u, v) \text{ given } x \leq u \text{ and } y \leq v \tag{6}$$
$$A(x, 1) = x and A(x, 0) = 0 \tag{7}$$

The first three properties are the commutativity, associativity and mononicity properties respectively. The last property guarantees the existence of identity and zero elements (1 and 0 respectively), which is essential to accommodate strict semantics in the weighted aggregation (see Section 5.3). The *and* operator is $\mathcal{MIN}(x, y) = \min\{x, y\}$. Any aggregation function that satisfies the four properties is called a t-norm([13]). Similarly, there are several t-normal aggregation functions whose fitness is domain-dependent.

Example. Suppose the DCT query specification is comprised of three attributes, namely power (P), throughput (T) and area (A). All three attributes are numeric, represented by the triangular fuzzy number in Figure 6. The parameters of the membership functions are listed below the corresponding nodes in the top tree shown in Figure 8. Note that **C** denotes the crisp core value. Also in Figure 8, the two bottom trees stand for two crisp specifications, S_1 and S_2 respectively. Their attribute values and scores are listed below the corresponding nodes. $\mathcal{MIN}$ is the unweighted aggregation function. The final aggregation score is marked just beside the root nodes of S_1 and S_2. The result shows that S_1 scores lower than S_2, since one of its sub-attributes (the throughput) receives the lowest score.

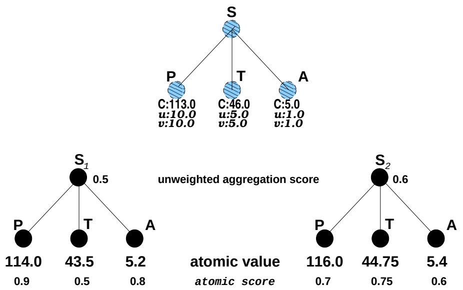

Figure 8: An Example of the Unweighted Aggregation

5.3 Weighted Aggregation

Conjunction aggregation treats every sub-attribute equally without taking into account the user's preference. However, a user often cares about some features more than others. As previously mentioned, a portable still camera designer may care, for instance more about power consumption than throughput. To be able to incorporate preference factor into a query, weights are assigned to each attribute according to their importance. Unweighted aggregation functions are subject to adaptations in order to reflect the new semantics. We call this process weighted conjunction aggregation. We now formalize the semantics of weighted aggregation. Let $\vec{W} = (w_1, w_2, \cdots, w_n)$ be a weight vector, $\vec{X} = (x_1, x_2, \cdots, x_n)$ be its corresponding score vector, and $A(\vec{X})$ be an unweighted aggregation function. A weighted aggregation function $A_{\vec{W}}(\vec{X})$ should satisfy:

$$A_{\vec{W}}(\vec{X}) = A(\vec{X}) \text{ given } w_1 = \cdots = w_n \tag{8}$$
$$A_{\vec{W}}(\vec{X}) = A_{\vec{W} \setminus (x_i = 0)}(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n) \tag{9}$$
$$A_{\vec{W}}(\vec{X}) \text{ is continuous with respect to } \vec{W} \tag{10}$$

The properties (8) and (9) are necessary. Otherwise, inconsistency would occur between the weighted and unweighted aggregations. Property (10) guarantees that a small weight change won't cause a drastic score variation. Once again, there are many aggregation algorithms satisfying the above three properties. However, given an additional property called local linearity ([14]), there leaves a unique candidate, namely the Fagin aggregation function.

$$A_{\vec{W}}(\vec{X_n}) = \sum_{i=1}^{n} (i \times (w_i - w_{i+1}) \times A(\vec{X_i})) \tag{11}$$

where $0 = w_{n+1} \leq w_n \leq w_{n-1} \leq \cdots \leq w_1$ and $\vec{X_i} = (x_1, x_2, \cdots, x_i)$.

Example. In Figure 9, we revisit the previous example (in Figure 8) with weight associated with each attribute and marked beside each node. We adopt the Fagin aggregation algorithm with $\mathcal{MIN}$ as the unweighted aggregation function. As before, the final aggregation scores are indicated beside the roots. It is noted that in Figure 8, S_1 gets a lower ranking due to a large deviation in its throughput from the query specification. However, if we assign a big weight on power attribute, its final aggregation score exceeds that of S_2. It is worth noting that in the above three steps, the last two treat their corresponding previous phases as black boxes. A weighted aggregation algorithm does not need to know the unweighted aggregation as long as it can call upon it when necessary. So does unweighted aggregation to atomic scoring. Therefore, the scoring mechanisms are mutually independent and the user has the freedom to replace any of them individually.

In our current implementation, all matching operations are shipped to the server side and executed by the match engine which performs a general-purpose computation. Of course, the client also can relinquish its right and let the server decide which matching policy to use. Fagin's algorithm with $\mathcal{MIN}$ is the default choice since this combination can accommodate strict semantic. More precisely, we can express "must have" semantics by grouping crisp attributes and assigning them the biggest weight. It is easy to verify by formula (11) and (7) that if the crisp attributes are not satisfied, the final score is 0; otherwise, the final score is the same as that computed without considering the crisp attributes.

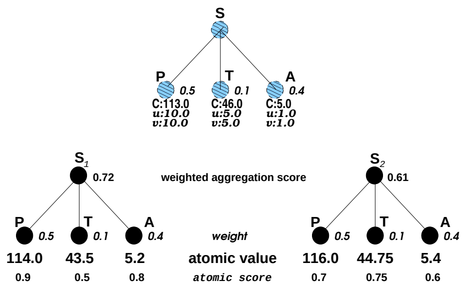

Figure 9: An Example of the Fagin (Weighted) Aggregation

6 Implementation, Experiments and Results

The matching system is developed completely in Java 1.2 on a Solaris 2.6 platform. All programs are organized in the root package *javacad.matching* and its four sub-packages (*engine, client, classserver, protocol*). We use *XML4J3.0.0 Early Release* (developed at IBM) as the XML parser. The core programs are around 3500 lines in 65 class files.

A concern with fuzzy matching algorithms with respect to traditional, exact matching approaches, is the computational cost of matching and ranking solutions. To demonstrate the efficiency of our implementation (i.e., the default matching and aggregation algorithms described in the previous section), we report results on an experiment designed as a stress test for the fuzzy matching engine. Our experiment is to select a DCT/IDCT component from a large IP database. The experiment scheme is:

1. choose a DCT specification S,

2. form a fuzzy query Q by associating each crisp attribute in S with an atomic scoring function,

3. generate a specification set $\mathcal{S}$ of 1000 elements by randomly tuning the crisp attributes in S,

4. run the matching algorithms with the query Q on $\mathcal{S}$.

As previously mentioned in Section 2, the first functionality selection will eliminate a large portion of IP components not related to the DCT, either by the external category index or by the fuzzy method described in Section 5.1. Assume only DSP IP components remain. Further selection on crisp criteria like the port map filters out other IP components such as FIR/IIR filters. The selection on the crisp criteria is actually integrated with fuzzy matching algorithms (see Section 5.3). Now the search scope is restricted to the IP components with nominally correct functionality and the same port map (or other crisp attributes). We claim the number of these final candidates can hadly exceeds 1000, which justifies our assumption that the generated 1000 IP specifications can simulate a very large IP database of heterogeneous components.

Table 2 details the DCT fuzzy query. The query has four sub-queries, namely power, throughput, area, and clock. Each of the sub-queries is a triangular fuzzy number with a corresponding core (crisp) value listed in the first row and with the parameters of the membership function listed in the second and third row. Table 3 lists the top 10 rankings from the 1000 artificially generated specifications using $\mathcal{MIN}$ as the unweighted aggregation function (marked in the fourth row of Table 2). Table 4 shows the top 10 rankings with weights using the Fagin algorithm with $\mathcal{MIN}$. In the table, we place importance on the power consumption attribute whose assigned weight is 10 times greater than the other three attributes (shown in the fifth row of Table 2). For comparison, both Table 3 and Table 4 list the unweighted and weighted rankings with the corresponding principal one in the leftmost column. Though it is hard to formalize the intuition of "proximity", we can see from Table 3 that the selected specifications are very close to S in whole. Also in Table 4, after assigning a large weight to the power attribute, the difference of selected specifications on this attribute is very small.

	power	throughput	area	clock
crisp value	113.0	49.0	5.0	133.0
triangular parameter u	10.0	5.0	1.0	10.0
triangular parameter v	10.0	5.0	1.0	10.0
weight	10	1	1	1

Table 2: A Fuzzy Query Specification

u/rank	w/rank	score	power	throughput	area	clock
1	1	0.9017	113.18	49.13	5.06	133.98
2	5	0.8799	114.20	49.41	4.95	132.66
3	2	0.8238	113.74	48.63	4.99	134.76
4	43	0.7517	115.42	50.24	4.93	130.74
5	14	0.7507	114.71	47.75	5.23	132.73
6	12	0.7335	111.44	47.67	5.23	134.40
7	35	0.7270	115.07	50.36	5.03	133.62
8	72	0.7266	115.73	50.37	5.21	134.27
9	81	0.7200	110.20	47.70	4.94	130.46
10	103	0.6867	110.19	47.43	4.91	131.44

Table 3: The Unweighted Query Result

7 Conclusion

In this paper, we present an IP component matching system in the JavaCAD framework. The system aims at automatic IP selection on design metrics. Our contribution is (1) to introduce XML as a representation tool between IP users/providers for organizing IP databases and forming IP queries, (2) to apply fuzzy logic to compute proximity between an IP query and a specification which are tree-structured models constructed from their respective XML representation. Experiment results show that our approach can capture well the intuition of "proximity matching".

References

[1] T. Thomas, *Technology for IP reuse and portability* IEEE Design & Test of Computers Volume: 16 4, Oct.-Dec. 1999, pp. 7 -13

[2] *Design and Reuse Organization* http://www.design-reuse.com/

[3] *Reusable Application-Specific Intellectual Property Developers* http://www.rapid.com/

[4] *Virtual Socket Interface Alliance (VSIA)* http://www.vis.org/

[5] H. Coors, N.M. Madrid, R. Seepold *Hardware/software co-design for IP objects based on CORBA* Fall VIUF Workshop, 1999, pp. 63-68

[6] M. Dalpasso, A. Bogliolo and L. Benini *Specification and validation of distributed IP-based designs with JavaCAD.* Proceedings of DATE 1999, Munich Germany pp. 684-688

[7] P. Schindler, K. Weidenbacher, T. Zimmermann *IP repository, a web-based IP reuse infrastructure* Custom Integrated Circuits, 1999 pp. 415-418

[8] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. *Proximity Search in Databases.* Proceedings of the 24th International Conference on Very Large Databases, New York, August 1998, pp. 26-37.

[9] M.Stefik, *Introduction to Knowledge Systems*, Morgan Kaufmann, 1995.

[10] L.Zadeh, *Probability Measures of Fuzzy events*, J. Math. Anal. and Appl., Vol. 23, pp.421-427, 1968.

[11] N. Sinha and M.Gupta,
textitSoft Computing and Intelligent Systems, Academic Press, 2000.

w/rank	u/rank	score	power	throughput	area	clock
1	1	0.9573	113.18	49.13	5.06	133.98
2	3	0.8942	113.74	48.63	4.99	134.76
3	19	0.8865	112.98	47.18	4.73	130.83
4	13	0.8826	113.19	50.69	5.23	132.51
5	2	0.8799	114.20	49.41	4.95	132.66
6	37	0.8305	112.43	51.12	4.86	135.19
7	18	0.8283	112.13	47.19	5.34	135.07
8	102	0.8278	112.97	51.77	5.15	129.62
9	92	0.8187	112.77	51.69	5.10	130.62
10	119	0.8180	113.02	46.21	4.41	133.42

Table 4: The Weighted Query Result

[12] R. Fagin *Fuzzy queries in multimedia database systems* Proc Seventeenth Symposium on Principles of Database Systems, Seattle, 1998, pp. 1-10.

[13] H. Bandemer, S. Gottwald *Fuzzy Sets, Fuzzy Logics and Fuzzy Methods with Applications* John Wiley & Sons Ltd.

[14] R. Fagin, E. L. Wimmers *Incorporating User Preferences in Multimedia Queries*, ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, pp. 247-261.