Interoperability among Heterogeneous Software Agents on the Internet

Katia Sycara, Jianguo Lu, and Matthias Klusch

CMU-RI-TR-98-22

The Robotics Institute
Carnegie Mellon University
Pittsburgh, USA.

October 1998

This research has been sponsored in part by Office of Naval Research grant N-00014-96-16-1-1222.

Contents

List of Figures

1 Introduction

Due to the exponential increase of offered services in the most famous offspring of the Internet, the World Wide Web, searching and selecting relevant services is essential for users. Various search engines and software agents providing various different services are already deployed on the Web. However, novice users of the Web may have no idea where to start their search, where to find what they really want, and what agents are available for doing their job. Even experienced users may not be aware of every change in the Web, e.g., relevant web pages might not exist or their content be valid anymore, and agents may appear and disappear over time. The user is simply overtaxed by manually searching in the Web for information or appropriate agents.

On the other hand, as the number and sophistication of agents on the Web that may have been developed by different designers increases, there is an obvious need for a standardized, meaningful communication among agents to enable them to perform collaborative task execution. We distinguish two general agent categories, service providers and service requester agents. Service providers provide some type of service, such as finding information, or performing some particular domain specific problem solving (e.g. number sorting). Requester agents need provider agents to perform some service for them. Since the Internet is an open environment, where information sources, communication links and agents themselves may appear and disappear unpredictably, there is a need for some means to help requester agents find providers. Agents that help locate others are called *middle agents*.

We have identified different types of middle agents in the Internet, such as matchmakers (yellow page services), brokers, billboards, etc. [3], and experimentally evaluated different protocols for interoperation between providers, requesters and various types of middle agents. Figure 1 shows the protocol for two different types of middle agents: brokers and matchmakers. We have also developed protocols for distributed matchmaking [12]. The process of finding an appropriate provider through a middle agent is called *matchmaking*. It has the following general form:

- Provider agents advertise their capabilities such as know-how, expertise, and so on, to middle agents.
- Middle agents store these advertisements.
- A requester asks some middle agent whether it knows of providers with desired capabilities.
- The middle agent matches the request against the stored advertisements and returns the result.

While this process at first glance seems very simple, it is complicated by the fact that providers and requesters are usually heterogeneous and incapable in general of understanding each other. This difficulty gives rise to the need for a common language for describing the capabilities and requests of software agents in a convenient way. In addition, one has to devise a mechanism for matching descriptions in that language. This mechanism can then be used by middle agents to efficiently select relevant agents for some given tasks.

In the following, we first elaborate the desiderata of an agent capability description language (ACDL), and propose such an ACDL, called LARKS, in detail. Then we will discuss the matchmaking process using LARKS and give a complete working scenario with some examples. We have implemented LARKS and the associated powerful matchmaking process, and are currently incorporating it within our RETSINA multi-agent infrastructure framework [22]. The paper concludes with comparing our language and the matchmaking process with related works.

2 Matchmaking Among Heterogeneous Agents

In the process of matchmaking (see Fig. 1) are three different kinds of collaborating agents involved:

1. **Provider** agents provide their capabilities, e.g., information search services, retail electronic commerce for special products, etc., to their users and other agents.

2. **Requester** agents consume informations and services offered by provider agents in the system. Requests for any provider agent capabilities have to be sent to a matchmaker agent.

3. **Matchmaker** agents mediate among both, requesters and providers, for some mutually beneficial cooperation. Each provider must first register himself with a matchmaker. Provider agents advertise their capabilities (advertisements) by sending some appropriate messages describing the kind of service they offer.

   Every request a matchmaker receives will be matched with his actual set of advertisements. If the match is successful the matchmaker returns a ranked set of appropriate provider agents and the relevant advertisements to the requester.

In contrast to a broker agent, a matchmaker does not deal with the task of contacting the relevant providers, transmitting the service request to the service provider and communicate the results to the requester. This avoids data transmission bottlenecks, but it might increase the amount of interactions among agents.

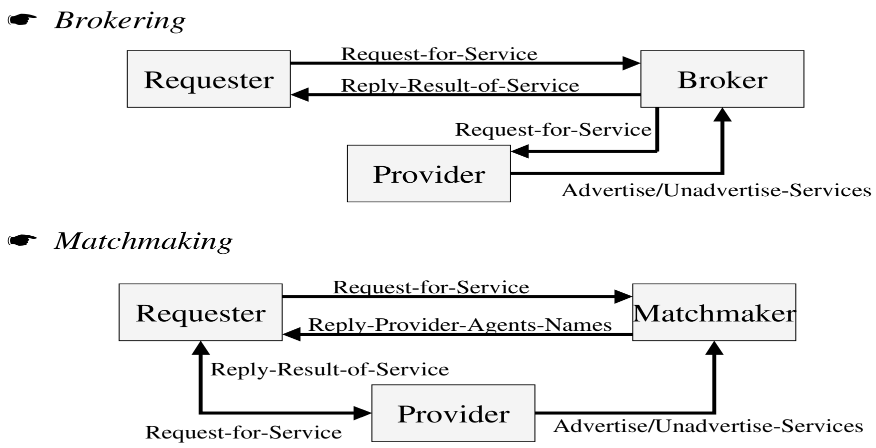

Figure 1: Service Brokering vs. Matchmaking

2.1 Desiderata for an Agent Capability Description Language

There is an obvious need to describe agent capabilities in a common language before any advertisement, request or even matchmaking among the agents can take place. In fact, the formal description of capabilities is one of the major problems in the area of software engineering and AI. Some of the main desired features of such a agent capability description language are the following.

- **Expressiveness**.
  The language is expressive enough to represent not only data and knowledge, but also to describe the meaning of program code. Agent capabilities are described at an abstract rather than implementation level. Most of existing agents can be distinguished by describing their capabilities in this language.

- **Inferences**.
  Inferences on descriptions written in this language are supported. A user can read any statement in the language, and software agents are able to process, especially to compare any pair of statements automatically.

- **Ease of Use**.
  Every description should not only be easy to read and understand, but also easy to write by the user. The language supports the use of domain or common ontologies for specifying agents capabilities. It avoids redundant work for the user and improves the readability of specifications.

- **Application in the Web.**
  One of the main application domains for the language is the specification of advertisements and requests of agents in the Web. The language allows for automated exchange and processing of information among these agents.

In addition, the matchmatching process on a given set of capability descriptions and a request, both written in the chosen ACDL, should be efficient, most accurate, not only rely on keyword extraction and comparison, and fully automated.

3 The Agent Capability Description Language LARKS

Representing capabilities is a difficult problem that has been one of the major concerns in the areas of software engineering, AI, and more recently, in the area of internet computing. There are many program description languages, like VDM or Z[28], to describe the functionalities of programs. These languages concern too much detail to be useful for the searching purpose. Also, reading and writing specifications in these languages require sophisticated training. On the other hand, the interface definition languages, like IDL, WIDL, go to the other extreme by omitting the functional descriptions of the services at all. Only the input and output information are provided.

In AI, knowledge description languages, like KL-ONE, or KIF are meant to describe the knowledge instead of the actions of a service. The action representation formalisms like STRIPS are too restrictive to represent complicated service. Some agent communication languages like KQML and FIPA concentrate on the communication protocals (message types) between agents but leave the content part of the language unspecified.

In internet computing, various description format are being proposed, notably the WIDL and the Resource Description Framework(RDF)[27]. Although the RDF also aims at the interoperablity between web applications, it is rather intended to be a basis for describing metadata. RDF allowes different vendors to describe the properties and relations between resources on the Web. That enables other programs, like Web robots, to easily extract relevant information, and to build a graph structure of the resources available on the Web, without the need to give any specific information. However, the description does not describe the functionalities of the Web *services*.

Since none of those languages satisfies our requirements, we propose an ACDL, called LARKS (**L**anguage for **A**dvertisement and **R**equest for **K**nowledge **S**haring) that enables for advertising, requesting and matching agent capabilities. It satisfies the desiderata given in the former section.

3.1 Specification in LARKS

A specification in LARKS is a frame with the following slot structure.

`Context`	Context of specification
`Types`	Declaration of used variable types
`Input`	Declaration of input variables
`Output`	Declaration of output variables
`InConstraints`	Constraints on input variables
`OutConstraints`	Constraints on output variables
`ConcDescriptions`	Ontological descriptions of used words

The frame slot types have the following meaning.

- `Context`.
  The context of the specification in the local domain of the agent.

- `Types`.
  Optional definition of the used data types. If not used, all data types are assumed to be defined in the following slots for input and output variables.

- `Input` and `Output`.
  Input/output variables for required input/output knowledge to describe a capability of an agent: if the input given to an agent fits with the specified input declaration part, then the agent is able to process an output as specified in the output declaration part. Processing takes all specified constraints on the input and output variables into consideration.

- `InConstraints` and `OutConstraints`.
  Logical constraints on input/output variables in the input/output declaration part. The constraints are specified as Horn clauses.

- `ConcDesriptions`.
  Optional description of the meaning of words used in the specification. The description relies on concepts in a given local domain ontology. Attachement of a concept C to a word w in any of the slots above is done in the form: w*C. That means that the concept C is the ontological description of the word w. The concept C is included in the slot `ConcDescription`.

In our current implementation we assume each local domain ontology to be written in the concept language ITL (Information Terminological Language). the syntax and semantics of the Itl are given in the appendix. Section 3.3 gives an example for how to attach concepts in a Larks specification, and also shows an example domain ontology in ITL. A generic interface for using ontologies in Larks expressed in languages other than Itl will be implemented in near future.

Every specification in Larks can be interpreted as an advertisement as well as a request; this depends on the purpose for which an agent sends a specification to some matchmaker agent(s). Every Larks specification must be wrapped up in an appropriate KQML message by the sending agent indicating if the message content is to be treated as a request or an advertisement.

3.2 Examples of Specifications in LARKS

The following two examples show how to describe in LARKS the capability to sort a given list of items, and return the sorted list. Example 3.1 is the the specification of the capability to sort a list of at most 100 integer numbers, whereas in example 3.2 a more generic kind of sorting real numbers or strings is specified in LARKS. Note that the `ConcDescriptions` slot is empty, i.e. the semantics of the words in the specification are assumed to be known to the matchmaker

Example 3.1: *Sorting integer numbers*

IntegerSort	
`Context`	Sort
`Types`	
`Input`	xs: `ListOf Integer`;
`Output`	ys: `ListOf Integer`;
`InConstraints`	le(length(xs),100);
`OutConstraints`	before(x,y,ys) $<-$ ge(x,y); in(x,ys) $<-$ in(x,xs);
`ConcDescriptions`	

∘

Example 3.2: *Generic sort of real numbers or strings*

GenericSort	
`Context`	Sorting
`Types`	
`Input`	xs: `ListOf` Real | String;
`Output`	ys: `ListOf` Real | String;
`InConstraints`	

`OutConstraints`	before(x,y,ys) $<-$ ge(x,y);
	before(x,y,ys) $<-$ preceeds(x,y);
	in(x,ys) $<-$ in(x,xs);

`ConcDescriptions`	

∘

The next example is a specification of an agent's capability to buy stocks at a stock market. Given the name of the stock, the amount of money available for buying stocks and the shares for one stock, the agent is able to order stocks at the stock market. The constraints on the order are that the amount for buying stocks given by the user covers the shares times the current price for one stock. After performing the order the agent will inform the user about the stock, the shares, and the gained benefit.

Example 3.3: *Selling stocks by a portfolio agent*

sellStock	
`Context`	Stock, StockMarket;
`Types`	
`Input`	symbol: StockSymbols; yourMoney: Money; shares: Money;
`Output`	yourStock: StockSymbols; yourShares: Money; yourChange: Money;
`InConstraints`	yourMoney >= shares*currentPrice(symb);
`OutConstraints`	yourChange = yourMoney - shares*currentPrice(symb); yourShares = shares; yourStock = symbol;
`ConcDescriptions`	

$\circ$

3.3 Using Domain Knowledge in Larks

As mentioned before, Larks offers the option to use application domain knowledge in any advertisement or request. This is done by using a local ontology for describing the meaning of a word in a Larks specification. Local ontologies can be formally defined using, e.g., concept languages such as Itl (see Appendix), BACK, LOOM, CLASSIC or KRIS, a full-fledged first order predicate logic, such as the knowledge interchange format (KIF), or even the unified modeling language (UML).

The main benefit of that option is twofold: (1) the user can specify in more detail what he is requesting or advertising, and (2) the matchmaker agent is able to make automated inferences on such kind of additional semantic descriptions while matching Larks specifications, thereby improving the overall quality of matching.

Example 3.4: *Finding informations on computers*

Suppose that a provider agent such as, e.g., HotBot, Excite, or even a meta-searchbot, like SavvySearch or MetaCrawler, advertises the capability to find informations about any type of computers. The administrator of the agent may specify that capability in Larks as follows.

FindComputerInfo	
Context	Computer*Computer;
Types	InfoList = ListOf(model: Model*ComputerModel, brand: Brand*Brand, price: Price*Money, color: Color*Colors);
Input	brands: SetOf Brand*Brand; areas: SetOf State; processor: SetOf CPU*CPU; priceLow*LowPrice: Integer; priceHigh*HighPrice: Integer;
Output	Info: InfoList;
InConstraints	
OutConstraints	sorted(Info).
ConcDescriptions	Computer = (and Product (exists has-processor CPU) (all has-memory Memory) (all is-model ComputerModel)); LowPrice = (and Price (ge 1800)(exists in-currency aset(USD))); HighPrice = (and Price (le 50000)(exists in-currency aset(USD))); ComputerModel = aset(HP-Vectra,PowerPC-G3,Thinkpad770,Satellite315); CPU = aset(Pentium,K6,PentiumII,G3,Merced) [Product, Colors, Brand, Money]

Most words in this specification have been attached with a name of some concept out of a given ontology. The definitions of these concepts are included in the slot `ConcDescriptions`. Concept definitions which were already sent to the matchmaker are enclosed in brackets. In this example we assume the underlying ontology to be written in the concept language Itl. An example for such an ontology is given in the next section.

Suppose that an agent registers himself at some matchmaker agent and sends the above specifications as advertisements. The matchmaker will then treat that agent as a provider agent, i.e., an agent who is capable to provide all these kinds of services.

3.3.1 Example for a Domain Ontology in the Concept Language Itl

As mentioned before, our current implementation of Larks assumes the domain ontology to be written in the concept language ITL.

The research area on concept languages (or description logics) in AI has its origins in the theoretical deficiencies of semantic networks in the late 70's. KL-ONE was the first concept language providing a well-founded semantic for a more native language-based description of knowledge. Since then different concept languages are intensively investigated; they are almost decidable fragments of first-order predicate logic. Several knowledge representation and inference systems, such as CLASSIC, BACK, KRIS, or CRACK, based on such languages are available.

Conceptual knowledge about a given application domain, or even common-sense, is defined by a set of concepts and roles as terms in the given concept language; each term as a definition of some concept C is a conjunction of logical constraints which are necessary for any object to be an instance of C. The set of terminological definitions forms a *terminology*. Any canonical definition of concepts relies in particular on a given basic vocabulary of words (primitive components) which are not defined in the terminology, i.e., their semantic is assumed to be known and consistently used across boundaries.

The following terminology, is written in the concept language ITL and defines concepts in the computer application domain. It is in particular used in the example 3.4 in the former section.

Product	=	(and (`all` is-manufactured-by Brand) (`atleast` 1 is-manufactured-by) (`all` has-price Price))
Computer	=	(and Product (`exists` has-processor CPU) (`all` has-memory Memory) (`all` is-model ComputerModel))
Notebook	=	(and Computer (`all` has-price (and (and (`ge` 1000) (`le` 2999)) (`all` in-currency `aset`(USD))) (`all` has-weight (and kg (le 5)) (all is-manufactured-by Company)) (`all` is-model aset(Thinkpad380,Thinkpad770,Satellite315))))
Brand	=	(and Company (`all` is-located-in State))
State	=	(and (`all` part-of Country) `aset`(VA,PA,TX,OH,NY))
Company	=	`aset`(IBM,Toshiba,HP,Apple,DEC,Dell,Gateway)
Colors	=	`aset`(Blue,Green,Yellow,Red)
Money	=	(and Real (`all` in-currency `aset`(USD,DM,FF,Y,P)))
Price	=	Money
LowPrice	=	(and Price (`ge` 1800)(`exists` in-currency `aset`(USD))),
HighPrice	=	(and Price (`le` 50000)(`exists` in-currency `aset`(USD)))
ComputerModel	=	`aset`(HP-Vectra,PowerPC-G3,Thinkpad380,Thinkpad770,Satellite315)
CPU	=	`aset`(Pentium,K6,PentiumII,G3,Merced)

∘

3.3.2 Subsumption Relationships Among Concepts

One of the main inferences on ontologies written in concept languages is the computation of the *subsumption* relation among two concepts: A concept C subsumes another concept C' if the extension of C' is a subset of that of C. This means, that the logical constraints defined in the term of the concept C' logically imply those of the more general concept C.

Any concept language is decidable if it is for concept subsumption among two concepts defined in that language. The concept language ITL we use is NP-complete decidable. The well-known trade-off between expressiveness and tractability of concept languages in practice is surrounded almost by subsumption algorithms which are correct but incomplete. We use an incomplete inference algorithm for computing subsumption relations among concepts in ITL.

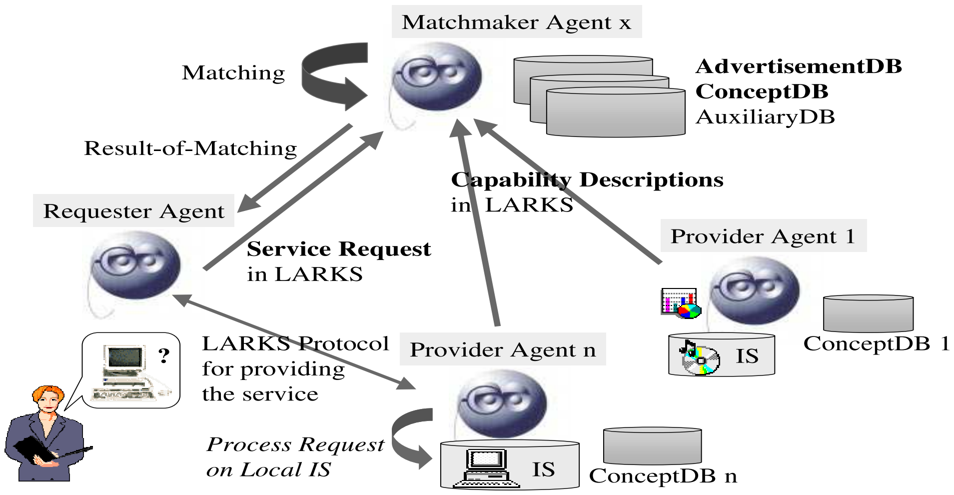

Figure 2: Matchmaking using LARKS: An Overview

For the mechanism of subsumption computation we refer the reader to, e.g., [19, 14, 20, 21].

The computation of subsumption relationships among all concepts in a ontology yields a so-called concept hierarchy. Both, the subsumption computation and the concept hierarchy are used in the matchmaking process (see section 4.1.2).

4 The Matchmaking Process Using LARKS

As mentioned before, we differentiate between three different kinds of collaborating information agents: provider, requester and matchmaker agents. The following figure shows an overview of the matchmaking process using LARKS.

The matchmaker agent process a received request in the following main steps:

- Compare the request with all advertisements in the advertisement database.
- Determine the provider agents whose capabilities match best with the request. Every pair of request and advertisement has to go through several different filtering during the matchmaking process.
- Inform the requesting agent by sending them the contact addresses and related capability descriptions of the relevant provider agents.

For being able to perform a steady, just-in-time matchmaking process the information model of the matchmaker agent comprises the following components.

1. *Advertisement database* (ADB).
   This database contains all advertisements written in LARKS the matchmaker receives from provider agents.

2. *Partial global ontology.*
   The ontology of the matchmaker consists of all ontological descriptions of words in advertisements stored in the ADB. Such a description is included in the slot `ConcDescriptions` and sent to the matchmaker with any advertisement.

3. *Auxiliary database.*
   The auxiliary data for the matchmaker comprise a database for word pairs and word distances, basic type hierarchy, and internal data.

Please note that the ontology of a matchmaker agent is not necessarily equal to the union of local domain ontologies of all provider agents who are actually registered at the matchmaker. This also holds for the advertisement database. Thus, a matchmaker agent has only partial global knowledge on available information in the overall multi-agent system; this partial knowledge might also be not up-to-date concerning the actual time of processing incoming requests. This is due to the fact that for efficiency reasons changes in the local ontology of an provider agent will not be propagated immediately to all matchmaker agents he is registered at. In the following we will describe the matchmaking process using LARKS in a more detail.

4.1 The Filtering Stages of the Matchmaking Process

The matching process of the matchmaker is designed with respect to the following criteria:

- The matching should *not be based on keyword retrieval only.* Instead, unlike the usual free text search engines, the semantics of requests and advertisements should be taken into consideration.

- The matching process should be *automated.* A vast amount of agents appear and disappear in the Internet. It is nearly impossible for a user to manually search or browse all agents capabilities.

- The matching process should be *accurate.* For example, if the matches returned by the match engine are claimed to be exact match or the plug-in match, those matches should satisfy the definitions of exact matching and plug-in matching.

- The matching process should be *efficient*, i.e., it should be fast.

- The matching process should be *effective*, i.e., the set of matches should not be too large. For the user, typing in a request and receiving hundreds of matches is not necessarily very useful. Instead, we prefer a small set of highly rated matches to a given request.

To fulfill the matching criteria listed in the above section, the matching process is organized as a series of increasingly stringent filters on candidate agents. That means that matching a given request into a set of advertisements consists of the following five filters that we organize in three consecutive filtering stages:

1. *Context Matching*

   Select those advertisements in the ADB which can be compared with the request in the same or similar context.

2. *Syntactical Matching*

   This filter compares the request with any advertisement selected by the context matching in three steps:

   (a) Comparison of profiles.

   (b) Similarity matching.

   (c) Signature matching.

   The request and advertisement profile comparison uses a weighted keyword representation for the specifications and a given term frequency based similarity measure (Salton, 1989). The last two steps focus on the (input/output) constraints and declaration parts of the specifications.

3. *Semantical Matching*

   This final filter checks if the input/output constraints of any pair of request and advertisement logically match (see section 4.1.5).

For reasons of efficiency the context filter roughly prunes off advertisements which are not relevant for a given request. In the following two filtering stages, syntactical and semantical matching, the remaining advertisements in the ADB of the matchmaker are checked in a more detail. All filters are independent from each other; each of them narrows the set of matching candidates with respect to a given filter criteria.

In our current implementation the matchmaker offers different types and modes of matching a request to a given set of advertisements.

4.1.1 Different Types of Matching in LARKS

Agent capability matching is the process of determining whether an advertisement registered in the matchmaker matches a request. But when can we say two descriptions *match* against each other? Does it mean that they have the same text? Or the occurrence of words in one discription sufficiently overlap with those of another discription? When both descriptions are totally different in text, is it still possible for them to match? Even if they match in a given sense, what can we then say about the matched advertisements? Before we go into the details of the matchmaking process, we should clarify the various notions of matches of two specifications.

4.1.1.1 Exact Match

Of course, the most accurate match is when both descriptions are equivalent, either equal literally, or equal by renaming the variables, or equal logically obtained by logical inference. This type of matching is the most restrictive one.

4.1.1.2 Plug-In Match

A less accurate but more useful match is the so-called $plug - in$ match. Roughly speaking, plug-in matching means that the agent which capability description matches a given request can be "plugged into the place" where that request was raised. Any pair of request and advertisement can differ in the signatures of their input/output declarations, the number of constraints, and the constraints themselves. As we can see, exact match is a special case of plug-in match, i.e., wherever two descriptions are exact match, they are also plug-in match.

A simple example of a plug-in match is that of the match between a request to sort a list of integers and an advertisement of an agent that can sort both list of integers and list of strings. This example is elaborated in section 5. Another example of plug-in match is between the request to find some computer information without any constraint on the output and the advertisement of an agent that can provide these informations and sorts the respective output.

4.1.1.3 Relaxed Match

The least accurate but most useful match is the so-called $relaxed$ match. A relaxed match has a much more weaker semantic interpretation than a exact match and plug-in match. In fact, relaxed match will not tell whether two descriptions semantically match or not. Instead it determines how close the two descriptions are by returning just a numerical distance value. Two descriptions match if the distance value is smaller than a preset threshold value. Normally the plug-in match and the exact match will be a special case of the relaxed match if the threshold value is not too small.

An example of a relaxed match is that of the request to find the place (or address) where to buy a Compaq Pentium233 computer and the capability description of an agent that may provide the price and contact phone number for that computer dealer.

Different users in different situation may want to have different types of matches. Although people usually may prefer to have plug-in matches, such a kind of match does not exist in many cases. Thus, people may try to see the result of a relaxed match first. If there is a sufficient number of relaxed matches returned a refined search may be performed to locate plug-in matching advertisements. Even when people are interested in a plug-in match for their requests only, the computational costs for this type of matching might outweigh its benefits.

As mentioned above we have five different matching filters:

1. context matching
2. profile comparison

3. similarity matching
4. signature matching
5. semantical matching

The first three filters are meant for relaxed matching, and the signature and semantical matching filter are meant for plug-in matching. Please note, that the computational costs of these filters are in increasing order. Users may select any combinations of these filters according their demand. Since the similarity filter also performs intensive computation one may just select the context filter and the profile filter if efficiency is of major concern.

Based on the given notions of matching we did implement four different modes of matching for the matchmaker:

1. **Complete Matching Mode**. All filtering stages are considered.
2. **Relaxed Matching Mode**. The first two filtering stages are considered except signature matching, i.e., the context, profile and similarity filter only.
3. **Profile Matching Mode**. Only the context matching and comparison of profiles is done.
4. **Plug-In Matching Mode**. In this mode, the matchmaker performs the signature and semantical matching.

As said above, the matching process proceeds in different filtering stages. If the considered advertisement and request contain conceptual attachments (ontological description of used words), then in most of the filtering stages (except for the comparison of profiles) we need a way to determine the semantic distance between the defined concepts. For that we use the computation of subsumption relationships and a weighted associative network.

4.1.2 Computation of Semantic Distances Among Concepts

We have presented the notion of concept subsumption in section 3.3.2. But the concept subsumption gives only a generalization/specialization relation based on the definition of the concepts via roles and attribute sets. In particular for matchmaking the identification of additional relations among concepts is very useful because it leads to a deeper semantic understanding. Moreover, since the expressivity of the concept language ITL is restrictive so that performance can be enhanced, we need some way to express additional associations among concepts.

For this purpose we use a so-called weighted associative network, that is a semantic network with directed edges between concepts as nodes. Any edge denotes the kind of a binary relation among two concepts, and is labeled in addition with a numerical weight (interpreted as a fuzzy number). The weight indicates the strength of belief in that relation, since its real world semantics may vary[1]. We assume that the semantic network consists of three kinds of binary, weighted relationships: (1) generalization, (2) specialization (as inverse of generalization), and (3) positive association among concepts (Fankhauser et al., 1991). The *positive association* is the most general relationship among concepts in the network indicating them as synonyms in some context. Such a semantic network is called an *associative network* (AN).

In our implementation we create an associative network by using the concept hierarchy of a given terminology defined in the concept language ITL. All subsumption relations in this concept hierarchy are used for setting the generalization and specialization relations among concepts in the corresponding associative network. Positive associations may be set by the administrator or user. Positive association, generalization and specialization are transitive.

As mentioned above, every edge in the associative network is labeled with a fuzzy weight. These weights are set by the user or automatically by default. The distance between two concepts in an associative network is then computed as the strength of the shortest path among them. Combining the strength of each relation in this path is done by using the following triangular norms for fuzzy set intersections (Kruse et al., 1991):

$$\begin{aligned} \tau_1(\alpha,\beta) &= max\{0, \alpha+\beta-1\} & n &= -1 \\ \tau_2(\alpha,\beta) &= \alpha\cdot\beta & n &= 0 \\ \tau_3(\alpha,\beta) &= min\{\alpha,\beta\} & n &= \infty \end{aligned}$$

Since we have three different kinds of relationships among two concepts in an AN the kind and strength of a path among two arbitrary concepts in the network is determined as shown in the following tables. For a formal discussion of that issue we refer to the work of Fankhauser et al. (1991), Kracker (1992), and Fankhauser and Neuhold (1992).

	g	s	p
g	g	p	p
s	p	s	p
p	p	p	p

Table 1: Kind of paths in an AN.

	g	s	p
g	τ_3	τ_1	τ_2
s	τ_1	τ_3	τ_2
p	τ_2	τ_2	τ_2

Table 2: Strength of paths in an AN.

For all $0 \leq \alpha, \beta \leq 1$ holds that $\tau_1(\alpha,\beta) \leq \tau_2(\alpha,\beta) \leq \tau_3(\alpha,\beta)$. Each triangular norm is monotonic, commutative and associative, and can be used as axiomatic sceletons for fuzzy set intersection. We restrict ourselves to a pessimistic, neutral, and optimistic t-norm τ_1, τ_2 and τ_3, respectively.

Since these triangular norms are not mutually associative the strength of a path in an associative network depends on the direction of strength composition. This asymmetry in turn might lead to unintuitive derived results: Consider, e.g., a path consisting of just three relations among four concepts C_1, C_2, C_3, C_4 with

[1]The relationships are fuzzy, and one cannot possibly associate all concepts with each other.

$C_1 \Rightarrow_{g,0.6} C_2 \Rightarrow_{g,0.8} C_3 \Rightarrow_{p,0.9} C_4$. It holds that $\tau_2(\tau_3(0.6, 0.8), 0.9) = 0.54$, but the strength of the same path in opposite direction is $\tau_2(\tau_2(0.9, 0.8), 0.6) = 0.43$. According to Fankhauser and Neuhold (1992) we can avoid this asymmetry by imposing a precedence relation ($3 > 2 > 1$) for strength combination (see Table 3).

	g	s	p
g	2	3	1
s	1	2	1
p	1	1	3

Table 3: Computational precedence for the strength of a path.

The computation of semantic distances among concepts is used in most of the filtering stages of the matching process. We will now describe each of the filters in detail.

4.1.3 Context Matching

It is obvious that any matching of two specifications has to be in an appropriate context. Suppose a provider agent advertises to sell several different types of products, like cars, computers, shoes, etc. Further assume that all his advertisements include the only input variable declaration: brand: `SetOf` Brand; But what is meant by the type 'Brand' in the context of any specification of a capability of finding a *particular* item? Without any additional knowledge about the particular context, a request to find information about a particular item, like computers, would match with *all* product advertisements.

In LARKS there are two possibilities to deal with this problem which is connected to the well-known ontological mismatch problem. First, the `Context` slot in a specification S contains a (list of) words denoting the domain of discourse for matching S with any other specification. When comparing two specifications it is assumed that their domains, means their context, are the same (or atleast sufficiently similar) as long as the real-valued distances between these words do not exceed a given threshold[2]. The matching process only proceeds if that is true.

Second, every word in a LARKS specification may be associated with a concept in a given domain ontology. Again, if the context of both specifications turned out to be sufficiently similar in the step before then the concept definitions describe the meaning of the words they are attached to in a more detail in the same domain. In this case, two concepts with same name but different definitions will be stored separately by extending each concept name by the identifier of the agent who did send this concept.

To summarize, the context matching consists of two consecutive steps:

[2]Any distance between two words is computed by an appropriate word distance function using the auxiliary database of the matchmaker.

1. For every pair of words u, v given in the `context` slots compute the real-valued word distances $d_w(u, v) \in[0,1]$. Determine the most similar matches for any word u by selecting words v with the minimum distance value $d_w(u, v)$. These distances must not exceed a given threshold.

2. For every pair of most similar matching words, check that the semantic distance among the attached concepts does not exceed a given threshold.

4.1.4 Syntactical Matching

4.1.4.1 Comparison of Profiles The comparison of two profiles relies on a standard technique from the Information Retrieval area, called term frequency-inverse document frequency weighting (TF-IDF) (see Salton, 1989). According to that, any specification in LARKS is treated as a document.

Each word w in a document Req is weighted for that document in the following way. The number of times w occurs throughout all documents is called the document frequency $df(w)$ of w. The used collection of documents is not unlimited, such as the advertisement database of the matchmaker.

Thus, for a given document d, the relevance of d based on a word w is proportional to the number $wf(w, d)$ of times the word w occurs in d and inverse proportional to $df(w)$. A weight $h(w, d)$ for a word in a document d out of a set D of documents denotes the significance of the classification of w for d, and is defined as follows:

$$h(w, d) = wf(w, d) \cdot log(\frac{|D|}{df(w)}).$$

The weighted keyword representation $wkv(d, V)$ of a document d contains for every word w in a given dictionary V the weight $h(w, d)$ as an element. Since most dictionaries provide a huge vocabulary we cut down the dimension of the vector by using a fixed set of appropriate keywords determined by heuristics and the set of keywords in LARKS itself.

The similarity $dps(Req, Ad)$ of a request Req and an advertisement Ad under consideration is then calculated by :

$$dps(Req, Ad) = \frac{Req \bullet Ad}{|Req| \cdot |Ad|}$$

where $Req \bullet Ad$ denotes the inner product of the weighted keyword vectors. If the value $dps(Req, Ad)$ does exceed a given threshold $\beta \in \mathbf{R}$ the matching process continues with the following steps.

The matchmaker then checks if the declarations and constraints of both specifications for a request and advertisement are sufficiently similar. This is done by a pairwise comparison of declarations and constraints in two steps:

1. *Similarity matching* and

2. *Signature matching*

4.1.4.2 Similarity Matching

Let E_i, E_j be variable declarations or constraints, and $S(E)$ the set of words in E. The similarity among two expressions E_i and E_j is determined by pairwise computation of word distances as follows:

$$Sim(E_i, E_j) = 1 - ((\sum_{(u,v) \in S(E_i) \times S(E_j)} d_w(u,v)) / |S(E_i) \times S(E_j)|))$$

The similarity value $Sim(S_a, S_b)$ among two specifications S_a and S_b in LARKS is computed as the average of the sum of similarity computations among all pairs of declarations and constraints:

$$Sim(S_a, S_b) = \sum_{(E_i,E_j) \in (D(S_a) \times D(S_b)) U (C(S_a) \times C(S_b))} Sim(E_i, E_j) / |(D(S_a) \times D(S_b)) U (C(S_a) \times C(S_b))|$$

with $D(S)$ and $C(S)$ denoting the input/output declaration and input/output constraint part of a specification S in LARKS, respectively.

4.1.4.3 Signature Matching

Consider the declaration parts of the request and the advertisement, and determine pairwise if their signatures of the (input or output) variable types match following the type inference rules given below.

Definition 4.1: *Subtype Inference Rules*

Consider two types t_1 and t_2 as part of an input or output variable declaration part (in the form `Input` $v : t_1$; or `Output` $v : t_2$;) in a LARKS specification.

1. Type t_1 is a subtype of type t_2 (denoted as $t_1 \preceq_{st} t_2$) if this can be deduced by the following subtype inference rules.
2. Two types t_1, t_2 are equal ($t_1 =_{st} t_2$) if $t_1 \preceq_{st} t_2$ and $t_2 \preceq_{st} t_1$ with
   (a) $t_1 =_{st} t_2$ if they are identical $t_1 = t_2$
   (b) $t_1 \mid t_2 =_{st} t_2 \mid t_1$ (commutative)
   (c) $(t_1 \mid t_2) \mid t_3 = t_1 \mid (t_2 \mid t_3)$ (associative)

Subtype Inference Rules:

1) $t_1 \preceq_{st} t_2$ if t_2 is a type variable

2) $\dfrac{t_1 =_{st} t_2}{t_1 \preceq_{st} t_2}$

3) t_1, t_2 are sets,
$\dfrac{t_1 \subset t_2}{t_1 \preceq_{st} t_2}$

4) $t_1 \preceq_{st} t_1 \mid t_2$

5) $t_2 \preceq_{st} t_1 \mid t_2$

6) $\frac{t_1 \preceq_{st} t_2, \; s_1 \preceq_{st} s_2}{(t_1,s_1) \preceq_{st} (t_2,s_2)}$

7) $\frac{t_1 \preceq_{st} t_2, \; s_1 \preceq_{st} s_2}{t_1|s_1 \preceq_{st} t_2|s_2}$

8) $\frac{t_1 \preceq_{st} t_2}{\texttt{SetOf}(t_1) \preceq_{st} \texttt{SetOf}(t_2)}$

9) $\frac{t_1 \preceq_{st} t_2}{\texttt{ListOf}(t_1) \preceq_{st} \texttt{ListOf}(t_2)}$

•

Matching of two signatures sig and sig' is done by a binary string-valued function fsm on signatures with

$$fsm(sig, sig') = \begin{cases} sub & sig' \preceq_{st} sig \\ Sub & sig \preceq_{st} sig' \\ eq & sig =_{st} sig' \\ disj & \text{else} \end{cases}$$

Having described both filters of the syntactical matching we now define the meaning of syntactical matching of two specifications written in LARKS.

__Definition__ **4.2:** *Syntactical matching of specifications in* LARKS

Consider two specifications S_a and S_b in LARKS with n_k input declarations, m_k output declarations, and v_k constraints $n_k, m_k \in \mathbf{N}, k \in \{a, b\}$, two declarations D_i, D_j, and constraints C_i, C_j in these specifications, and V a given dictionary for the computation of weighted keyword vectors. Let β, γ, θ be real threshold values for profile comparison and similarity matching.

- The **declarations** D_i **and** D_i **syntactically match** if they are sufficiently similar:

  $$Sim(D_i, D_j) \geq \gamma \wedge fsm(D_i, D_j) \neq disj.$$

  The **constraints** C_i **and** C_j **syntactically match** if they are sufficiently similar:

  $$Sim(C_i, C_j) \geq \gamma.$$

  If both words in every pair $(u, v) \in S(E_i) \times S(E_j)$ of most similar words are associated with a concept C and C', respectively, then the distance among C and C' in the so-called associative network of the matchmaker must not exceed a given threshold value θ.

  The syntactical match of two declarations or constraints is denoted by a boolean predicate $Synt$.

- The **specifications S_a and S_b syntactically match** if
  1. their profiles match, i.e., $dps(S_a, S_b) \geq \beta$, and
  2. for each declaration or constraint E_i, $i \in \{1, ...n_a\}$ in the declaration or constraint part of S_a there exists a most similar matching declaration or constraint E_j, $j \in \{1, ..., n_b\}$ in the declaration or constraint part of S_b such that

     $$Synt(E_i, E_j) \ \wedge \ Sim(E_i, E_j) = max\{Sim(E_i, E_y), y \in \{1, .., n_b\}\}$$

     (Analogous for each declaration or constraint in S_b.)
  3. for each pair of declarations determined in (1.) the matching of their signatures is of the same type, i.e., for each (D_i, D_j) in (1.) it holds that the value $fsm(D_i, D_j)$ is the same, and
  4. the similarity value $Sim(S_a, S_b)$ exceeds a given threshold.

•

4.1.5 Semantical Matching

By using the syntactical filter many matches might be found in a large agent society. Hence, it is important to use some kind of semantic information to narrow the search, and to pin down more precise matches.

The most common and natural interpretation for a specification (even for a software program) is using sets of pre- and post-conditions, denoted as Pre_S and $Post_S$, respectively. In a simplified notation, any specification S can be represented by the pair $(Pre_S, Post_S)$.

Definition 4.3: *Semantical matching of two specifications*

Consider two specifications $S(Pre_S, Post_S)$ and $T(Pre_T, Post_T)$.

The specification S **semantically matches** the specification T if

$$(Pre_S \Rightarrow Pre_T) \ \wedge \ (Post_T \Rightarrow Post_S)$$

That means, the set of pre-conditions of S logically implies that of T, and the set of post-conditions of S is logically implied by that of T.

•

The problem in performing the semantical matching is that the logical implication is not decidable for first order predicate logic, and even not for a set of Horn clauses. To make the matching process tractable and feasible, we have to decide on the expressiveness of the language used to represent the pre- and post- conditions, and to choose a relation that is weaker than logical implication. The θ-subsumption relation among two constraints C, C' (denoted as $C \preceq_\theta C'$) appears to be a suitable choice for semantical matching, because it is computationally tractable and semantically sound.

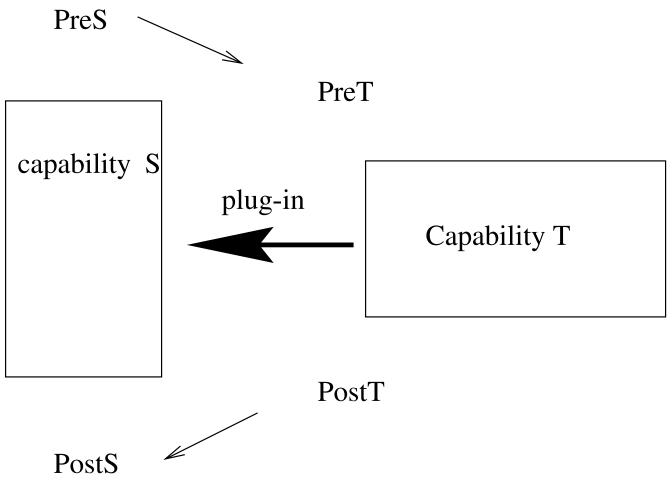

Figure 3: Plug-In Match of Specifications: T plugs into S.

4.1.5.1 Plug-in Semantical Matching in LARKS It is proven in the software engineering area that if the condition of semantical matching in definition 4.3 holds, and the signatures of both specifications match, then T can be directly used in the place of S, i.e., T plugs in S (see figure 4.1.5).

Definition 4.4: *Plug-In semantical matching of two specifications*

Given two specifications $Spec1$ and $Spec2$ in LARKS then $Spec1$ **plug-in matches** $Spec2$ if

- Their signatures matches (see section 4.1.4.2).
- For every clause $C1$ in the set of input constraints of $Spec1$ there is a clause $C2$ in the set of input constraint of $Spec2$ such that $C1 \preceq_\theta C2$.
- For every clause $C2$ in the set of output constraints of $Spec2$ there is a clause $C1$ in the set of output constraints of $Spec1$ such that $C2 \preceq_\theta C1$.

where $\preceq_\theta$ denotes the θ-subsumption relation between constraints.

•

4.1.5.2 θ-Subsumption between Constraints One suitable selection of the language and the relation is the (definite program) clause and the the so-called θ-subsumption relation between clauses, respectively.[3] In the following we will only consider Horn clauses. A general form of Horn clause is

[3] A *clause* is a finite set of *literals*, which is treated as the universally quantified disjunction of those *literals*. A *literal* may be positive or negative. A positive *literal* is an *atom*, a negtive literal is the negation of an *atom*. A *definite program clause* is a clause with one positive

$a_0 \lor (\neg a_1) \lor ... \lor (\neg a_n)$, where each $a_i, i \in \{1, ..., n\}$ is an atom. This is equivalent to $a_0 \lor \neg(a_1 \land ... \land a_n)$, which in turn is equivalent to $(a_1 \land ... \land a_n) \Rightarrow a_0)$.[4] We adopt the standard notation for that clause as $a_0 \leftarrow a_1, \ldots, a_n$; in PROLOG the same clause is written as a_0 :- $a_1, \ldots, a_n$.

Examples of definite program clauses are

- $Date.year > 1995$, $sorted(computerInfo)$,
- $before(x, y, ys) \leftarrow ge(x, y)$, and
- $scheduleMeeting(group1, group2, interval, meetingDuration, meetTime) \leftarrow belongs(p1, group1), belongs(p2, group2), subset(meetTime, interval), length(meetTime) = meetingDuration, available(p1, meetTime), available(p2, meetTime)$.

We say that a clause C θ-**subsumes** another clause D (denoted as $C \succeq_\theta D$) if there is a substitution θ such that $C\theta \subseteq D$. C and D are θ-equivalent if $C \preceq_\theta D$ and D
$preceq_\theta C$.

Examples of θ-subsumption between clauses are

- $P(a) \leftarrow Q(a) \ \preceq_\theta \ P(X) \leftarrow Q(X)$
- $P(X) \leftarrow Q(X), R(X) \ \preceq_\theta \ P(X) \leftarrow Q(X)$.

Since a single clause is not expressive enough, we need to use a set of clauses to express the pre and post conditions (i.e., the input and output constraints) of a specification in LARKS. A set of clauses is treated as a conjunction of those clauses.

Subsumption between two set of clauses is defined in terms of the subsumption between single clauses. More specifically, let S and T be such sets of clauses. Then, we define that S θ-subsumes T if every clause in T is θ-subsumed by a clause in S.

There is a complete algorithm to test the θ-subsumption relation, which is in general NP-complete but polynomial in certain cases. On the other hand, θ-subsumption is a weaker relation than logical implication, i.e., from $C \preceq_\theta D$ we can only infer that C logically implies D but not vice versa.[5]

5 Examples of Matchmaking using LARKS

Consider the specifications 'IntegerSort' and 'GenericSort' (see example 3.1, 3.2) as a request of sorting integer numbers and an advertisement for some agent's

literal and zero or more negative literals. A *definite goal* is a clause without positive literals. A *Horn clause* is either a definite program clause or a definite goal.

[4] The literal a_0 is called the head of the clause, and $(a_1 \land ... \land a_n)$ is called the body of the clause.

[5] Please also note that the θ-subsumption relation is similar to the query containment in database. When advertisements are database queries, specification matching is reduced to the problem of query containment testing.

capability of sorting real numbers and strings, respectively.

IntegerSort	
`Context`	Sort
`Types`	
`Input`	xs: `ListOf Integer`;
`Output`	ys: `ListOf Integer`;
`InConstraints`	le(length(xs),100);
`OutConstraints`	before(x,y,ys) $<-$ ge(x,y); in(x,ys) $<-$ in(x,xs);
`ConcDescriptions`	

GenericSort	
`Context`	Sorting
`Types`	
`Input`	xs: `ListOf` Real | String;
`Output`	ys: `ListOf` Real | String;
`InConstraints`	

`OutConstraints`	before(x,y,ys) $<-$ ge(x,y);
	before(x,y,ys) $<-$ preceeds(x,y);
	in(x,ys) $<-$ in(x,xs);

`ConcDescriptions`	

Assume that the requester and provider agent sends the request IntegerSort and advertisment GenericSort to the matchmaker, respectively. Figure 5 describes the overall matchmaking process for that request.

1. *Context Matching*
   Both words in the **`Context`** declaration parts are sufficiently similar. We have no referenced concepts to check for terminologically equity. Thus, the matching process proceeds with the following two filtering stages.

2. *Syntactical Matching*

   (a) *Comparison of Profiles*
   According to the result of TF-IDF method both specifications are sufficiently similar:

   (b) *Signature Matching*
   Consider the signatures t_1= (**`ListOf Integer`**) and t_2= (**`ListOf Real`**|**`String`**). Following the subtype inference rules 9., 4. and 1. it holds that $t_1 \preceq_{st} t_2$, but not vice versa, thus $fsm(D_{11}, D_{21}) =$ sub. Analogous for $fsm(D_{12}, D_{22}) =$ sub.

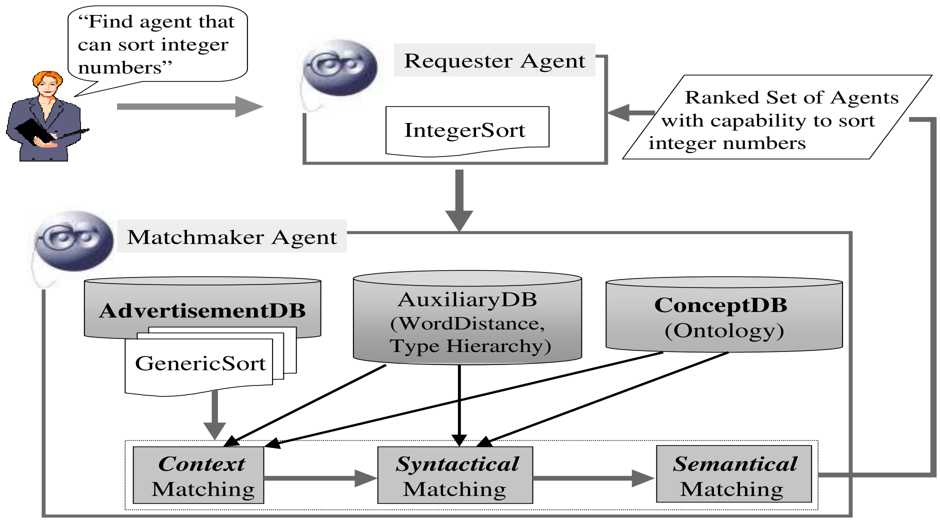

Figure 4: An Example of Matchmaking using LARKS

(c) *Similarity Matching*
Using the current auxiliary database for word distance values similarity matching of constraints yields:

le(length(xs),100))	null	= 1.0
before(x,y,ys) $<-$ ge(x,y)	in(x,ys) $<-$ in(x,xs)	= 0.5729
in(x,ys) $<-$ in(x,xs)	before(x,y,ys) $<-$ preceeds(x,y))	= 0.4375
before(x,y,ys)$<-$ ge(x,y))	before(x,y,ys) $<-$ preceeds(x,y))	= 0.28125

The similarity of both specifications is computed as:
$Sim(\texttt{IntegerSort}, \texttt{GenericSort}) = 0.64$.

3. *Semantical Matching*
The advertisement `GenericSort` also matches semantically with the request `IntegerSort`, because the set of input constraints of `IntegerSort` θ-subsumes that of `GenericSort`, and the output constraints of `GenericSort` θ-subsumes that of `IntegerSort`. Thus `GenericSort` plugs into `IntegerSort`. Please note that this does not hold vice versa.

6 Related works

Agent matchmaking has been actively studied since the inception of software agent research. The earlist matchmaker we are aware of is the ABSI facilitator, which is based on the KQML specification and uses the KIF as the content language. The KIF expression is basically treated like the Horn clauses. The matching between the advertisement and request expressed in KIF is the simple unification with the equality predicate. Matchmaking using LARKS performs better than ABSI in both, the language and the matching process. The plug-in matching in LARKS uses the θ-subsumption test, which select more matches that are also semantically matches.

The SHADE and COINS[17] are matchmakers based on KQML. The content language of COINS allowes for the free text and its matching algorithm utilizes the tf-idf. The contect language of SHADE matchmaker consists of two parts, one is a subset of KIF, another is a structured logic representation called MAX. MAX use logic frames to declaratively store the knowledge. SHADE uses a frame like representation and the matcher use the prolog like unifier.

A more recent service broker-based information system is InfoSleuth[10, 11]. The content language supported by InfoSleuth is KIF and the deductive database language LDL++, which has a semantics similar to Prolog. The constraints for both the user request and the resource data are specified in terms of some given central ontology. It is the use of this common vocabulary that enables the dynamic matching of requests to the available resources. The advertisements specify agents' capabilities in terms of one or more ontologies. The constraint matching is an intersection function between the user query and the data resource constraints. If the conjunction of all the user constraints with all the resource constraints is satisfiable, then the resource contains data which are relevant to the user request.

A somewhat related research area is the research on information mediators among heterogenous information systems[23][1]. Each local information system is wrapped by a so-called wrapper agent and their capabilities are described in two levels. One is what they can provide, usually described in the local data model and local database schema. Another is what kind of queries they can answer; usually it is a subset of the SQL language. The set of queries a service can accept is described using a grammar-like notation. The matching between the query and the service is simple: it just decides whether the query can be generated by this grammar. This area emphasizes the planning of database queries according to heterogeneous information systems not providing complete SQL sevices. Those systems are not supposed to be searched for among a vast number of resources on the Internet.

The desfription of capabilities and matching are not only studied in the agent community, but also in other related areas.

6.1 Works related with capability description

The problem of capability and service descriptions can be tackled at least from the following different approaches:

1. Software specification techniques.
   Agents are computer programs that have some specific characteristics. There are numerous work for software specifications in formal methods, like model-oriented VDM and Z[28], or algebraic-oriented Larch. Although these languages are good at describing computer programs in a precise

way, the specification usually contains too much details to be of interests to other agents. Besides, those existing languages are so complex that the semantic comparison between the specifications is impossible. The reading and writing of these specifications also require substantial training.

2. Action representation formalisms.
   Agent capability can be seen as the actions that the agents perform. There are a number of action representation formalisms in AI planning like the classical one the STRIPS. The action representation formalism are inadequate in our task in that they are propositional and not involving data types.

3. Concept languages for knowledge representation.
   There are various terminological knowledge representation languages. However, ontology itself does not describe capabilities. On the other hand, it provides auxiliary concepts to assist the specification of the capabilities of agents.

4. Database query capability description.
   The database query capability description technique is developed as an attempt to describe the information sources on the Internet, such that an automated integration of information is possible. In this approach the information source is modeled as a database with restricted quering capabilities.

6.2 Works related with service retrieval

There are three broad approaches to service retrieval. One is the information retrieval techniques to search for relevant information based on text, another is the software component retrieval techniques[26][8][13] to search for software components based on software specifications. The third one is to search for web resources that are typically described as database models[18][23].

In the software component search techniques, [26] defined several notions of matches, including the exact match and the plug-in match, and formally proved the relationship between those matches. [8] propsed to use a sequence of filters to search for software components, for the purpose to increase the efficiency of the search process. [13] computed the distance between similar specifications. All these work are based on the algebraic specification of computer programs. No concept description and concept hierarchy are considered in their work.

In Web resource search techniques, [18] proposed a method to look for better search engines that may provide more relevant data for the user concerns, and rank those search engines according to their relevance to user's query. They propose the directory of services to record descriptions of each information server, called a server description. A user sends his query to the directory of services, which determins and ranks the servers relevant to the user's request. Both the query and the server are described using boolean expression. The search method is based on the similarity measure between the two boolean expressions.

7 Conclusion

The Internet is an open system where heterogeneous agents can appear and disappear dynamically. As the number of agents on the Internet increases, there is a need to define middle agents to help agents locate others that provide requested services. In prior research, we have identified a variety of middle agent types, their protocols and their performance characteristics. Matchmaking is the process that brings requester and service provider agents together. A provider agent advertises its know-how, or capability to a middle agent that stores the advertisements. An agent that desires a particular service sends a middle agent a service request that is subsequently matched with the middle agent's stored advertisements. The middle agent communicates the results to the requester (the way this happens depends on the type of middle agent involved). We have also defined protocols that allow more than one middle agent to maintain consistency of their adevertisement databases. Since matchmaking is usually done dynamically and over large networks, it must be efficient. There is an obvious trade-off between the quality and efficiency of service matching in the Internet.

We have defined and implemented a language, called LARKS, for agent advertisement and request and a matchmaking process using LARKS. LARKS judiciously balances language expressivity and efficiency in matching. LARKS performs both syntactic and semantic matching, and in addition allows the specification of concepts (local ontologies) via ITL, a concept language.

The matching process uses five filters, namely context matching, comparison of profiles, similarity matching, signature matching and semantic matching. Different degrees of partial matching can result from utilizing different combinations of these filters. Selection of filters to apply is under the control of the user (or the requester agent).

Acknowledgements:
We would like to thank Davide Brugali for helpful discussions and Seth Widoff for help with the implementation. This research has been sponsored by ONR grant N-00014-96-16-1-1222.

A Syntax of Larks

Definition A.1: *Syntax of Larks*

The syntax of Larks is given by the following production system in EBNF-grammar:

$< specification >$	::=	$< Ident >\ [< CDeclaration >]\ [< TDeclarations >]$
		$[< Declarations >]\ [< Constraints >]$
$< CDeclaration >$	::=	$'\texttt{Context}'\ < CDec >$
$< CDec >$	::=	$< Ident >'\ *'\ < Termdefinition >';'$
$< TDeclarations >$	::=	$< TDec > \mid < TDec >';' < TDeclarations >$
$< Delarations >$	::=	$'\texttt{Input}'\ < OptDecList >\ '\texttt{Output}'\ < DecList >$
$< TDec >$	::=	$'\texttt{type}'\ < Ident > [':: '< TExp >]';' \mid '\texttt{basicType}'\ < IdentList >';'$
$< Dec >$	::=	$< Ident >':'< TExp > ['='< Exp >]';'$
$< DecList >$	::=	$< Dec > \mid < Dec >';'< DecList >$
$< OptDecList >$	::=	$< OptDec > \mid < OptDec >';'< OptDecList >$
$< OptDec >$	::=	$['\texttt{Optional}']\ < Dec >$
$< TExp >$	::=	$< TVar > \mid < BType > \mid < PType > \mid < CType >$
$< PType >$	::=	$\texttt{Bool}' \mid \texttt{Int}' \mid \texttt{Real}' \mid '\texttt{String}'$
$< CType >$	::=	$'('[< Ident >':']\ < TExp >','\ [< Ident >':']\ < TExp >')' \mid$
		$< TExp >\ '\mid'\ < TExp > \mid$
		$< TExp >\ '->'\ < TExp > \mid$
		$'\texttt{SetOf}'\ '('< TExp >')' \mid$
		$'\texttt{ListOf}\ '('TExp')' \mid$
		$'\{'< ExpList >'\}'$
$< Exp >$	::=	$< aExp > \mid '('< ExpList >')' \mid '\{'< ExpList >'\}' \mid$
		$< Exp >'\ ('< ExpList >')' \mid < Exp >'\ .' < Ident >$
$< ExpList >$	::=	$< Exp > \mid < Exp >','\ < ExpList >$
$< aExp >$	::=	$< sConst > \mid < var > \mid < const >$
$< IdentList >$	::=	$< Ident > \mid < Ident >','< IdentList >$
$< Constraints >$	::=	$['\texttt{InConstraints}\ < formulaList >]$
		$['\texttt{OutConstraints}'\ < formulaList >]$
$< fomulaList >$	::=	$< formula > \mid < formula >';'\ < formulaList >$
$< formula >$	::=	$< atomList >$
$< atomList >$	::=	$< atom > \mid < atom >','\ < atomList >$
$< atom >$	::=	$< predicate > \mid '\texttt{not}'\ < predicate >$
$< predicate >$	::=	$< Ident >$
$< var >$	::=	$< Ident >$
$< const >$	::=	$< Ident >$

with non-terminals $< Ident >, < var >$, and $< const >$ denoting an identifier, variable and constant, respectively. The non-terminal $< Termdefinition >$ refers to that in the concept language Itl (see below), thus denoting a kind of a so-called 'escape hatch' from Larks to Itl.

Convention:
In a capability description or request any term definition will be replaced by the name of the corresponding concept or role which is assumed to be available in the local knowledge base.

•

B The concept language ITL

<u>Definition</u> B.1: *Syntax of* ITL

The syntax of the concept language ITL is given by the following production system in EBNF-grammar:

$$
\begin{array}{lll}
<\mathit{Terminology}> & ::= & <\mathit{Termdefinition}>^{+} \\
<\mathit{Termdefinition}> & ::= & <\mathit{Conceptdefinition}> \mid \ <\mathit{Roledefinition}> \\
<\mathit{Conceptdefinition}> & ::= & <\mathit{atomicConcept}> \ '=' \ <\mathit{Concept}> \mid \\
 & & <\mathit{atomicConcept}> \ '=' \ <\mathit{Concept}> \\
<\mathit{Roledefinition}> & ::= & <\mathit{atomicRole}> \ '=' \ <\mathit{Role}> \mid \\
 & & <\mathit{atomicRole}> \ '=' \ <\mathit{Role}> \\
<\mathit{Concept}> & ::= & <\mathit{Conc}> \mid \ <\mathit{AttrConc}> \\
<\mathit{Conc}> & ::= & <\mathit{atomicConcept}> \mid \\
 & & <\mathit{primComponent}> \mid '(\underline{\texttt{not}}' \ <\mathit{primConcComponent}>) \mid \\
 & & '(\texttt{and}' \ <\mathit{Concept}>^{+} \ ')' \mid \\
 & & '(\texttt{atleast}' \ n \ <\mathit{Role}>')' \mid \\
 & & '(\texttt{atmost}' \ m \ <\mathit{Role}>')' \mid \\
 & & '(\texttt{exists}' \ <\mathit{Role}> \ <\mathit{Concept}>')' \mid \\
 & & '(\texttt{all}' \ <\mathit{Role}><\mathit{Concept}>')' \mid \\
 & & '(\texttt{le}' \ <\mathit{num}>')' \mid '(\texttt{ge}' \ <\mathit{num}>')' \mid \\
 & & '(\texttt{lt}' \ <\mathit{num}>')' \mid '(\texttt{gt}' \ <\mathit{num}>')' \\
<\mathit{AttrConc}> & ::= & '\texttt{aset(}'<\mathit{aval}>^{+} \ ')' \\
<\mathit{Role}> & ::= & '(\texttt{androle}' \ <\mathit{Role}>^{+} \ ')' \mid \\
 & & <\mathit{atomicRole}> \mid \ <\mathit{primRoleComponent}> \\
<\mathit{atomicConcept}> & ::= & <\mathit{identifier}> \mid '\mathit{nothing}' \\
<\mathit{atomicRole}> & ::= & <\mathit{identifier}> \\
<\mathit{primComponent}> & ::= & <\mathit{primConcComponent}> \mid \ <\mathit{primRoleComponent}> \\
<\mathit{primConcComponent}> & ::= & <\mathit{identifier}>'.' \\
<\mathit{primRoleComponent}> & ::= & <\mathit{identifier}>'.' \\
<\mathit{aval}> & ::= & <\mathit{identifier}> \\
<\mathit{Term}> & ::= & <\mathit{Concept}> \mid \ <\mathit{Role}> \\
<\mathit{ObjectSet}> & ::= & <\mathit{Instance}>^{*} \\
<\mathit{Instance}> & ::= & <\mathit{ConceptInstance}> \mid \ <\mathit{RoleInstance}> \\
<\mathit{ConceptInstance}> & ::= & '('<\mathit{Object}> \ <\mathit{atomicConcept}>')' \mid \\
 & & '(<\mathit{Object}> \ \texttt{not}' \ <\mathit{primConcComponent}>')' \\
<\mathit{RoleInstance}> & ::= & '('<\mathit{Object}> \ <\mathit{atomicRole}> \ <\mathit{Object}>')' \mid \\
 & & '('<\mathit{Object}> \ <\mathit{NumRestr}> \ <\mathit{atomicRole}>')' \\
<\mathit{NumRestr}> & ::= & '\texttt{atleast}' \ <\mathit{num}> \mid '\texttt{atmost}' \ <\mathit{num}> \\
<\mathit{Object}> & ::= & <\mathit{identifier}>
\end{array}
$$

The meaning of (atomic) concept or role, attribute concept, concept and role definition, term definition, term, terminology and object set is defined as the set of strings which can be reduced to the respective non-terminal symbols in the production system.

It is assumed that in every terminology T (written in ITL) all used atomic concepts

and roles are unique identifiers and defined in T; the enumerable sets of identifiers for concepts and roles, attribute values and objects, as well as primitive concept and role components are assumed to be pairwise disjoint. In addition, every primitive component (undefined identifier) in a terminology is assigned a given, fixed meaning[6].
•

Definition B.2: *Semantic of* ITL

Let G be a grammar, $\mathcal{D}$ interpretation domain and D, D_a disjoint subsets with $\mathcal{D} = D \uplus D_a$. $\mathcal{P}(S)$ denotes the power set of any set S. The semantic of ITL terms is defined by the following interpretation function.

$$\epsilon : \begin{cases} Conc & \rightarrow & \mathcal{P}(\mathcal{D}) \\ Role & \rightarrow & \mathcal{P}(D \times \mathcal{D}) \\ Attr & \rightarrow & D_a \end{cases} \tag{1}$$

ϵ is a ITL-**interpretation** if it satisfies the following equations:

$$\begin{aligned}
\epsilon(\texttt{and } C_1 ... C_n)) &= \bigcap_{i=1}^{n} \epsilon(C_i) & (2)\\
\epsilon((\texttt{all } R\ C)) &= \{d \in \mathcal{D} : rg(d, \epsilon(R)) \subseteq \epsilon(C)\} & (3)\\
\epsilon((\texttt{exists } R\ C)) &= \{d \in \mathcal{D} : rg(d, \epsilon(R)) \cap \epsilon(C) \neq \emptyset\} & (4)\\
\epsilon((\texttt{atleast } n\ R)) &= \{d \in \mathcal{D} : \ \mid rg(d, \epsilon(R)) \mid \ \geq n\} & (5)\\
\epsilon((\texttt{atmost } n\ R)) &= \{d \in \mathcal{D} : \ \mid rg(d, \epsilon(R)) \mid \ \leq n\} & (6)\\
\epsilon(\texttt{aset}(a_1, ..., a_n)) &= \{\epsilon(a_1), ..., \epsilon(a_n)\} & (7)\\
\epsilon((\texttt{not } C^p)) &= D \setminus \epsilon(C^p) & (8)\\
\epsilon((\texttt{androle } R_1 ... R_n)) &= \bigcap_{i=1}^{n} \epsilon(R_i) & (9)\\
\epsilon(nothing) &= \emptyset & (10)
\end{aligned}$$

with

$$rg(d, \epsilon(R)) := \{y \in \mathcal{D} : (d, y) \in \epsilon(R)\} \tag{11}$$

$rg(d, \epsilon(R))$ denotes the set of *role fillers* of instance d for the role R.

All attributes $a_1, ..., a_n$ of the concept $\underline{\texttt{aset}}(a_1, ..., a_n)$ are interpreted as constants, i.e., for some $D_a \subseteq Attr$ we assign $\epsilon(a_i) = a_i, i \in \{1, ..n\}$. The interpretation of the operators (`le` n), (`ge` n), (`lt` n), and (`gt` n) for numerical comparison denotes the set of real numbers $x \in D$ with $x \leq n,\ x \geq n, x < n$, and $x > n$, respectively.

•

[6] Primitive components are elements of a minimal common vocabulary used by each agent provider/user for a construction of their local domain-dependent terminologies (and object sets).

References

[1] Jose' Luis Ambite and Craig A. Knoblock. Planning by Rewriting: Efficiently Generating High-Quality Plans. Proceedings of the Fourteenth National Conference on Artificial Intelligence, Providence, RI, 1997.

[2] J. E. Caplan, M. T. Harandi. A logical framework for software proof reuse. Proceedings of the ACM SIGSOFT Symposium on Software Reusability, April 1995. ACM Software Engineering Note, Aug. 1995.

[3] K. Decker, K. Sycara, M. Williamson. Middle-Agents for the Internet. Proc. 15th IJCAI, pages 578-583, Nagoya, Japan, August 1997.

[4] S. Cranefield, A. Diaz, M. Purvis. Planning and Matchmaking for the Interoperation of Information Processing Agents. The Information Science Discussion Paper Series No. 97/01, University of Otago.

[5] P. Fankhauser, M. Kracker, E.J. Neuhold. Semantic vs. Structural Resemblance of Classes. Special Issue: Semantic Issues in Multidatabase Systems, ACM SIGMOD RECORD, Vol. 20, No. 4, pp.59-63, 1991.

[6] P. Fankhauser, E.J. Neuhold. Knowledge based integration of heterogeneous databases. Proceedings of IFIP Conference DS-5 Semantics of Interoperable Database Systems, Lorne, Victoria, Australia, 1992.

[7] T. Finin, R. Fritzson, D. McKay, R. McEntire. KQML as an Agent Communication Language. Proc. 3rd International Conference on Information and Knowledge Management CIKM-94, ACM Press, 1994.

[8] J. Goguen, D. Nguyen, J. Meseguer, Luqi, D. Zhang, V. Berzins. Software component search. Journal of Systems Integration, 6, pp. 93-134, 1996.

[9] G. Huck, P. Fankhauser, K. Aberer, E.J. Neuhold. Jedi: Extracting and Synthesizing Information from the Web. Proceedings of International Conference on Cooperative Information Systems CoopIS'98, IEEE Computer Society Press, 1998.

[10] Jacobs,N., Shea,R., 1995, "Carnot and InfoSleuth - Database Technology and the WWW", ACM SIGMOD Intern. Conf. on Management of Data, May 1995

[11] Jacobs,N., Shea,R., 1996, "The role of Java in InfoSleuth: Agent-based exploitation of heterogeneous information ressources", Proc. of Intranet-96 Java Developers Conference, April 1996

[12] S. Jha, P. Chalasani, O. Shehory and K. Sycara. A Formal Treatment of Distributed Matchmaking. In Proceedings of the Second International conference on Autonomous Agents (Agents 98), Minneapolis, MN, May 1998.

[13] J-J. Jeng, B.H.C. Cheng. Specification matching for software reuse: a foundation. Proceedings of the ACM SIGSOFT Symposium on Software Reusability, ACM Software Engineering Note, Aug. 1995.

[14] M. Klusch. *Cooperative Information Agents on the Internet.* PhD Thesis, University of Kiel, December 1996 (in German) Kovac Verlag, Hamburg, 1998, ISBN 3-86064-746-6.

[15] M. Kracker. A fuzzy concept network. Proc. IEEE International Conf. on Fuzzy Systems, 1992.

[16] R.Kruse, E.Schwecke, J.Heinsohn. *Uncertainty and Vagueness in Knowledge Based Systems*, Springer, 1991.

[17] D. Kuokka, L. Harrada. On using KQML for Matchmaking. Proc. 3rd Intl. Conf. on Information and Knowledge Management CIKM-95, pp. 239-45, AAAI/MIT Press, 1995.

[18] S.-H. Li, P. B. Danzig. Boolean Similarity Measures for Resource Discovery. IEEE Transactions on Knowledge and Data Engineering, Vol.9, No. 6, November/December, 1997.

[19] B. Nebel. *Reasoning and revision in hybrid representation systems*, Lecture Notes in Artificial Intelligence LNAI Series, Vol. 422, Springer, 1990.

[20] G. Smolka, and B. Nebel. *Representation and Reasoning with attributive descriptions.* IWBS Report 81, IBM Deutschland Wissenschaftl. Zentrum, 1989.

[21] G. Smolka, and M. Schmidt-Schauss. Attributive concept description with complements, AI 48, 1991.

[22] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed Intelligent Agents. IEEE Expert, pp36-46, December 1996.

[23] V. Vassalos, Y. Yapakonstantinou. Expressive Capabilities Description Languages and Query Rewriting Algorithms. available at http://www-cse.ucsd.edu/ yannis/papers/vpcap2.ps

[24] G. Wickler. Using Expressive and Flexible Action Representations to Reason about Capabilities for Intelligent Agent Cooperation. http://www.dai.ed.ac.uk/students/gw/phd/story.html

[25] WordNet - a Lexical Database for English. http://www.cogsci.princeton.edu/ wn/

[26] A. M. Zaremski, J. M. Wing Specification matching of software components. Technical Report CMU-CS-95-127, 1995.

[27] Resource Description Framework (RDF) Schema Specification, http://www.w3.org/TR/WD-rdf-schema/.

[28] Ben Potter, Jane Sinclair, David Till, Introduction to Formal Specification and Z, Prentice-Hall International Series in Computer Science.