

# Characterizing Web Service Substitutivity with Combined Deductive and Inductive Engines

Marcelo A.T. Aragão and Alvaro A.A. Fernandes

Department of Computer Science, University of Manchester  
Oxford Road, Manchester M13 9PL, UK  
{m.aragao|a.fernandes}@cs.man.ac.uk

**Abstract.** This paper proposes a lightweight mechanism to combine deductive and inductive inference engines and describes an application in which the resulting functionality is shown to add significant value. Within the unified setting of deductive database technology and inductive logic programming, the combination of inference engines is much easier and the resulting systems much better integrated. This combination is important, e.g., if software agents are indeed to become widespread in the technical context of pervasive distributed computing that the web promises. Software agents will need to automate more decision-making tasks, for some of which they may need to learn new knowledge on the fly. The dynamic, volatile nature of the environment in which they are expected to operate make this need all the more acute. A recent, challenging example is the web services approach to distributed business computing. This paper describes an application of a combined inference engine built by the authors in which the problem of characterizing whether one web service is likely to be a substitute for another is solved in a fluent, disencumbered manner such as will be needed if it is to be deployed by software agents.

## 1 Introduction

The application scenario is the web services [13] approach to distributed business computing. A web service deploys functionality defined using the Web Service Description Language (WSDL) [9]. WSDL enables vendor-independence, platform neutrality and opens the way for the orchestration of available web services so as to deliver complex, combined functionality. For instance, a `BuyStock` web service can be assembled by sequencing the web services `getStockQuote`, `checkCredit` and `purchaseStockBlock`, with each component being possibly provided by different organizations (e.g., a market data provider, a brokerage firm and a stock exchange clearing system, respectively). There already exist proposals for languages that enable web service orchestration into workflows that deliver the desired combination of functionalities. Examples are IBM's Web Service Flow Language (WSFL) [17] and Microsoft's XLANG [22].

In order to make web services available, providers will publish web service descriptions in distributed directories [11] conforming to the Universal Description, Discovery, and Integration (UDDI) standard interface [2]. In order to make use of web services available, requestors find them in UDDI directories and using the information contained in the WSDL documents thus fetched, requestors can bind to the web services needed and invoke them using, e.g., Simple Object Access Protocol (SOAP) [5].

While UDDI directories, by their very nature, provide the basic mechanism with which to find alternative services (in the sense that they act as a shop window), a more discriminatory requirement (e.g., one targeting a substitute that is not only similar in non-apparent ways, but also type safe and of good quality) goes beyond basic UDDI capabilities. Thereby, a great deal of human intervention is likely to be required to find substitutes for a given web service. In other words, given a target web service  $w$  one needs to track down, and remove inappropriate substitutes for  $w$  from the set of candidates for binding, while, as far as possible, ensuring that appropriate substitutes for  $w$  are indeed included among the candidates for binding.

Despite the formulation above, the issue is not necessarily an information retrieval one. Even if  $w$  is known in advance (which means that no search is involved), rather the requestor moves straight on to an attempt to bind to  $w$ , such an attempt is not guaranteed to succeed (e.g.,  $w$  may be down) or to have a desirable outcome from the viewpoint of the requestor even if the attempt does succeed (e.g., the quality of service that  $w$  is contracted to may have declined, or its cost may have climbed up).

This leads to the following additional observations. Since the run time environment for a web service orchestration cannot be guaranteed to be static or stable, a requirement that was stipulated (at workflow specification time) to a web service  $w$  may fail to be satisfied if  $w$  cannot be bound to (either at all or not appropriately) unless there is human intervention to re-route the request (by suggesting an alternative binding, i.e., an appropriate substitute for  $w$ ). Moreover, the dynamic, volatile nature of the web means that conditions and contexts are constantly changing, and hence that human intervention is likely to be required since adaptation and learning are crucial.

The contributions of a lightweight mechanism that combines deductive and inductive inference engines are, thus, motivated by the desirability of attenuating the need for human intervention in contexts such as above, insofar as human intervention is likely to be costly and prone to inconsistency. One such mechanism is shown in this paper to contribute a solution to the following problem: given a WSDL-specified web service  $w$ , find a set  $W$  of substitutes for  $w$ , where a web service  $w' \in W$  is a substitute for  $w$  if  $w'$  is *similar* to  $w$ , it is (type) *safe* to invoke  $w'$  in lieu of  $w$ , and  $w'$  is of *good quality*, where the italicized words denote properties and relationships in need of formalization (provided below). In doing so, the purpose of this paper is to provide evidence that such mechanism can viably endow, e.g., software agents, with the capabilities to automate

more decision-making tasks, for some of which it may be necessary to learn new knowledge on the fly at low cost with appropriate guarantees of accuracy.

The remainder of the paper is structured as follows. Section 2 defines inferential task flows. Section 3 shows how to instantiate the basic framework with concrete inferences engines and auxiliary algorithms. Then, in Section 4, a solution to the problem is expressed as an inferential task flow, thereby providing evidence for the utility of the concept. Section 5 briefly considers work that is, to varying degrees, related to the contributions reported. Finally, Section 6 discusses some of the implications of these contributions.

## 2 Defining Inferential Task Flows

This section introduces the concept of inferential task flows that chain deductive and inductive tasks.

*Preliminaries.* The underlying logical machinery is that of Datalog theories [18], especially as understood in the fields of deductive databases [8] and inductive logic programming [20]. The reader is assumed to be broadly familiar with the basic technical knowledge contained in the above references. Usage that is particularly relevant to this paper is now described.

In deductive tasks, a set  $K$  of rules characterizes an intensional database and a set  $D$  of facts characterizes an extensional database. Under the usual constraints [8], the set  $K \cup D$  characterizes a deductive database. In inductive tasks, a set  $B$  of rules is used to represent the background knowledge with which the inductive task can be underpinned, and a set  $H$  of rules is used to represent the hypothesis output by the inductive algorithm. In inductive tasks that model supervised learning [19] (e.g., concept learning), a set  $E^+$  (respectively,  $E^-$ ) of facts is used to represent the set of positive (respectively, negative) examples. In the case of concept learning, the hypothesis  $H$  is an operational definition of the target concept. Given an instance,  $H$  decides whether it belongs to the extension of the target concept. In inductive tasks that model unsupervised learning [19] (e.g., conceptual clustering), a set of facts is used to represent each case to be considered by the algorithm in the inductive process. In the case of conceptual clustering, the hypothesis  $H$  is an operational definition of a hierarchy of clusters. Given an instance,  $H$  assigns the instance to the extension of the cluster in  $H$  that the instance most resembles.

In this paper, as is customary in the deductive database area [8], deductive tasks are constrained to query answering against a deductive database. Also, inductive tasks are constrained to concept learning in the case of supervised learning and to conceptual clustering in the case of unsupervised learning. For both concept learning and clustering, there is a need to specify a **language bias**  $L$  [19], and this is done either indirectly, via simplified grammar formalisms (which characterize the subset of clauses constituting the language of interest), or directly, via a given set of clauses taken as the language of interest.

*Inferential Task Flows.* Let **Rules**, **Facts**, and **Clauses**, respectively denote the set of all rules, facts, and clauses. Let **Bias** denote the set of all language biases over Datalog clauses. The following definitions are used in this paper.

A **preparation policy** is a mapping with the signature **Rules**  $\times$  **Facts**  $\rightarrow$  **Rules**  $\times$  **Facts**. Informally, a preparation policy defines a computation that precedes the invocation of an inference engine. The primary purpose of a preparation policy is to select which rules and facts constitute the set of premises for the invocation of an inference engine that immediately follows. As examples of preparation policies, first consider a predicate that filters facts for recency. If such a predicate is used to constrain the active domain of a deductive database before an inference engine (as defined below) is invoked on the latter, then the predicate implements a preparation policy. A similar predicate may, in addition, filter out rules with very high selectivity, thereby implementing another preparation policy.

An **inference engine** is a mapping with the signature **Bias**  $\times$  **Rules**  $\times$  **Facts**  $\rightarrow$  **Rules**  $\times$  **Facts**. A **deductive inference engine** is an inference engine in which the input bias is a query and the set of rules returned is the empty set. An example of a deductive engine is a deductive database (e.g., the one in [10], pp. 373–398). An **inductive inference engine** is an inference engine in which the input bias constrains the syntactic form of the set of rules returned and the set of facts returned is the empty set. An example of an inductive engine is a relational concept learner (e.g., as implemented by mFOIL [15]). Another example is a conceptual clustering algorithm (e.g., as implemented by COBWEB [12]).

An **assimilation policy** is a mapping with the signature **Clauses**  $\times$  **Rules**  $\times$  **Facts**  $\rightarrow$  **Rules**  $\times$  **Facts**. Informally, an assimilation policy defines a computation that follows the invocation of an inference engine. The primary purpose of an assimilation policy is to select which clauses that now follow (i.e., that result from the immediately preceding invocation of an inference engine) are to be chained forward to subsequent inferential tasks by being incorporated into the currently available rules and facts. As examples of assimilation policies, first consider a predicate that filters classification rules for minimum accuracy results. If such a predicate is used to avoid incorporating high-accuracy rules into the currently available rules, then the predicate implements an assimilation policy. Another example of assimilation policy, this time in the deductive setting, is the materialization of query results.

An **inferential task** is a triple  $\langle \text{Prepare}, \text{Infer}, \text{Assimilate} \rangle$ , where **Prepare**, **Infer**, and **Assimilate** are, respectively, a preparation policy, an inference engine, and an assimilation policy, as defined above. An **inferential task flow** is a (possibly empty) sequence of inferential tasks. A formal semantics for inferential task flows is given in the full version of the paper [1].

The next section shows how the authors have generated or reused mappings that instantiate **Prepare**, **Infer**, and **Assimilate** to build a prototype and gauge its applicability in modern, challenging contexts.

### 3 Instantiating Inferential Task Flows

This section describes some instantiations for the components of inferential tasks that have been implemented by the authors and are later used in the example application reported in the paper.

For inferential task flows to be deployed, one must provide instantiations for the mappings that inferential tasks combine at each step of the flow.

*Example Inference Engines.* The benefits of inferential task flows are manifested more clearly if one can appeal to a variety of inferential engines since it is the chaining of results which gives rise to added value. Thus, one would like to have a set of inference engines at one's disposal when composing inferential task flows.

The contributions of this paper are based on exactly such an implemented collection of inference engines. The authors have built a prototype implementing the semantics given in the full version of the paper [1]. There, the following inference engines can be used to draw bindings for *Infer*: the deductive database engine described in [10], pp. 373–398, referred to in the rest of the paper as *deduce*; the relational concept learner *mFOIL* [15], referred to in the rest of the paper as *learn\_concept*; the COBWEB conceptual clustering algorithm [12], referred to in the rest of the paper as *cluster*.

The primary goal in building such a prototype with such a pool of inference engines from which to draw bindings for *Infer* was to investigate concrete situations in relevant application scenarios where a gap in the performance of a complex sequence of tasks may need to (or may better) be bridged by a combination of deductive and inductive inference. Once these mappings have been implemented, they can be invoked in an inferential task. It is then the role of the corresponding preparation and assimilation policies to mediate the use that each inference engine makes of the available stock of rules and facts.

*Example Preparation and Assimilation Policies.* A working prototype also requires bindings for preparation and assimilation policies. There is a natural split in the kind of policies one would use before and after an inference step based on whether the inference is deductive or inductive in kind.

Before invoking a deductive engine, one needs to decide which theory (in the case of a deductive database, which facts and which rules) one wants to infer from. After invoking a deductive engine one needs to decide whether (and to what extent) to assimilate the derived information into which theory. Here, questions regarding the status of the inferred results do not arise because deduction is truth-preserving and, in the case of deductive databases, the underlying logic is monotonic.

Before invoking an inductive engine, one needs to decide which theory one wants to infer as output. This is typically specified as a language bias which in the case of concept learning, constrains the syntactic form of the inferred hypothesis, and in the case of conceptual clustering, specifies which features constitute the dimensions of the clusters to be returned. After invoking an inductive engine, one needs to decide whether (and to what extent) to assimilate the derived information into which theory. Here, questions regarding the status

of the inferred results are more relevant, insofar as induction is usually justifiable by probabilistic reasoning. One can, however, take the simplifying tack of delegating to the user the responsibility for deciding on the matter. Or, perhaps, be permissive, i.e., assimilate everything. Or, still, the opposite, viz., never to assimilate anything. This paper does not address these questions further but this is not to belittle their relevance.

The contributions of this paper are based on a simple set of policies that suffice, in this context, to demonstrate that there is added value in deploying inferential task flows. The policies used in the application described in Section 4 are as follows ([1] contains a formal definition of these policies):

- `use_available_stocks` is a preparation policy for deduction that is all inclusive, in that it does not filter out any of the available rules and facts.
- `use_examples` is a preparation policy for conceptual clustering that stipulates which examples are to be used in the inductive task, and which background knowledge can the learner rely on (in the case of COBWEB, background knowledge is not exploited).
- `use_examples_of_concept` is a preparation policy for concept learning that specializes `use_examples`, in the sense that it also stipulates the name and arity of the predicate to be learned (while allowing any predicate in the background knowledge to appear in the body of any clause in the hypothesis).
- `materialize_consequences` is an assimilation policy for deduction which (e.g., with efficiency concerns in mind) materializes the set of logical consequences computed and (because a well-formedness condition on deductive databases requires it) drops the rule that defined the view that was queried.
- `assimilate_concept` is an assimilation policy for concept learning that models the reverse of `materialize_consequences` insofar as it replaces all instances of the concept with the rules that allow them to be computed by the application of a deductive engine.
- `assimilate_clusters` is an assimilation policy for clustering that specializes `assimilate_concept` insofar as it appends to the current set of rules an axiomatization of the `is_a` (i.e., member-of) and `a_kind_of` (i.e., subset-of) relationships, in order to enrich the set of logical consequences derivable from the resulting cluster.

Using the above set of mappings for inference engines, and for preparation and assimilation policies, one can bind `Prepare`, `Infer`, and `Assimilate` in different ways in different inferential tasks to compose inferential task flows that compute results not easily achievable by other mechanisms. The next section provides some evidence for this claim.

## 4 Applying Inferential Task Flows

This section describes how the inferential tasks flows defined in Section 2 and instantiated as described in Section 3 can be deployed to solve the problem described in Section 1.

*Design.* Given a web service  $w$ , to identify substitutes for  $w$  one needs to state the constraints on the relationship between  $w$  and its potential substitutes.

In this application, a web service  $w'$  is a substitute for a web service  $w$  if  $w'$  is similar to  $w$ , it is type safe to replace  $w$  with  $w'$ , and  $w'$  is good (in some sense, in some context). A web service  $w'$  is similar to a web service  $w$  if in a taxonomy of web services, there is a non-root common ancestor to  $w$  and  $w'$ . A web service  $w'$  is type safe to replace  $w$  if it is the case that an input to  $w'$  is a subtype<sup>1</sup> of the corresponding input to  $w$  and the output of  $w$  is a subtype of the output of  $w'$  [7]. A web service  $w'$  is good in the context of, say, a user community if it is classifiable as such by some membership function that captures the aggregated opinion of the community.

Now, using a deductive engine, the above specification is almost directly executable (modulo a specific syntactic convention) provided that there are no gaps in one's knowledge of how the subclauses are to be defined. However, at the very next level down, one finds an obstacle.

One needs a taxonomy of web services that can be used to measure the similarity of each member of a collection of web services to the web service  $w$  that one wants a substitute for. This means that one needs a hierarchy of clusters based on features of interest. Notice that such a taxonomy is relative to  $w$  and that the set of potential substitutes is likely to be frequently and autonomously updated. This implies that predefining and hardwiring such taxonomies is both tedious and ultimately very costly due to their inherent volatility. In short, one would like to induce this hierarchy of clusters on the fly. This is possible if one can (in a separate data preparation task) generate from WSDL documents represented as XML files, a set of service descriptions in clausal form and then take these as cases for a clustering algorithm.

A similar argument to the above may be made regarding the characterization of a service as good. One needs a rule that classifies a service as good based on certain attributes, but it is neither possible nor desirable to define this rule and hardwire it, because the quality of a service is variable over time and also the desiderata of a user community also change over time. In short, here too, one would like to induce such a rule on the fly. This is possible if one (again, in a separate data preparation task) can label entries in a web services usage log as examples of good service, generate from these a set of facts denoting examples of good service and submit these to a concept learning algorithm.

Thus, in order for a deductive engine to be able to characterize substitutivity in a dynamic way, one needs to be able to induce on the fly and in clausal form, a hierarchy of clusters and a concept definition. Unless there is a fluent integration of inductive capabilities into the computational mechanism available, the two issues described above characterize gaps that cannot be easily bridged, as they require invoking other tools. Using the inferential task flows defined in Section 2, instantiated as described in Section 3, one can easily express the desired computation as now described.

<sup>1</sup> Since WSDL documents presuppose the XML Schema specification, the latter can be used to define which types stand in a subtype relationship to one another.

*Implementation.* The implementation of the design described in Section 4 consists of assembling a set of Datalog clauses that defines a (effectively computable) substitutivity relation in the context specified in Section 1 and satisfying the constraints stated there. This assembly is now described in a top-down manner.

---

<pre> substitutable(Service1,Service2):-   similar(Service1,Service2),   safe(Service1,Service2),   goodService(Service2).  similar(Service1,Service2):-   is_a(Service1,Cluster1),   is_a(Service2,Cluster2),   Service1 \= Service2,   a_kind_of(Cluster1,CommonAncestor),   a_kind_of(Cluster2,CommonAncestor),   CommonAncestor \= root. </pre>	<pre> subtype('xsd:byte','xsd:short'). subtype('xsd:short','xsd:integer'). subtype('xsd:integer','xsd:long'). ...  safe(Service1,Service2):-   service(Service1,...,InType1,OutType1,...),   service(Service2,...,InType2,OutType2,...),   Service1 \= Service2,   subtype(InType2,InType1),   subtype(OutType1,OutType2). </pre>
---	---

---

**Fig. 1.** Definition of Substitutable Web Service

The substitutivity relation is a direct statement of the problem expressed as the `substitutable/2` predicate in the upper-left corner of Figure 1. A web service  $w'$  is a substitute for a web service  $w$  if it is similar to  $w$ , type safe to invoke in lieu of  $w$  and of good quality. This implies the need for definitions for the relations similarity and safeness, and for the property of having good quality.

Similarity is defined by `similar/2` in the lower-left corner of Figure 1. Given a similarity-based taxonomy of web services (i.e., one which clusters together those web services that have more features in common with one another),  $w'$  is similar to  $w$  if they have a non-root common ancestor. This implies that this taxonomy is available. Using an inferential task, one can apply a conceptual clustering algorithm to generate on-the-fly such a taxonomy, given the web service one wants substitutes for. The cases for the clustering algorithm are generated by a (largely mechanizable) data preparation step that retrieves WSDL documents from UDDI directories and maps them to Datalog facts, which comprise the extension of a `service/n` predicate. This is simply a view on the facts that result from the conversion of WSDL documents from XML to Datalog syntax mentioned above. The  $n$  features picked by the view ultimately come from the set of tags used in WSDL. There is no fixed number because of varying arities, alternative denotations and so on.

The inferential task that bridges the gap created by the need to induce a hierarchy of clusters on the fly is specified as follows. The preparation policy is `use_examples`. It selects all the `service/n` facts available. The inference engine used is `cluster`, i.e., a logic-based implementation of COBWEB [12]. The assimilation policy is `assimilate_clusters`. It incorporates as facts into the rule base the nodes in the hierarchy of clusters induced by `cluster`, as well as the edges that characterize the topology of the hierarchy. The edges of such a taxonomy map to `is_a/2` and `a_kind_of/2` edges and axiomatizing these as (respectively) the set membership and the set inclusion relationships, one can capture the entire information content of the cluster hierarchy.

Safety is defined by `safe/2` in the bottom-right corner of Figure 1. This models the idea of type safeness in the object-oriented literature (as described, e.g., in [7]). A web service  $w'$  is (type) safe with respect to a web service  $w$  if each input of  $w'$  is a supertype of the corresponding input in  $w$  and the output of  $w'$  is a subtype of that of  $w$ . The definition of `service/n` is as mentioned above. The `subtype/2` relation formalizes the specifications in XML Schema [4] which WSDL documents abide by. Some examples are shown in the top-right corner of Figure 1.

By assumption, a definition for `goodService/1` must be obtained at run time, so as to adjust to dynamic, volatile conditions. Using an inferential task flow, one can use a concept learning algorithm to induce on-the-fly such a definition.

The inferential task that bridges the gap created by the need to induce on the fly the concept of a good service is specified as follows. The preparation policy is `use_examples_of_concept`. It presupposes (and selects all of) the labeled examples of good services generated in a preliminary data preparation step, as one would retrieve by inspection and ranking of the data recorded in usage logs. The inference engine used is `learn_concept`, i.e., an implementation of the relational concept learner mFOIL [15]. The assimilation policy is `assimilate_concept`, which removes all the elements in the extension of the learned concept that have been provided as examples for the learner and, in their place, asserts the rule that is output by the inductive engine.

The complete application, i.e., the derivation of the extension of the substitutivity relation can be expressed as an inferential task flow in which all but the two inductive tasks described above are deductive. For all such deductive tasks, the preparation policy is `use_available_stocks`, i.e., no filtering is applied and all facts and rules in the rule base are available for the inference engine. The inference engine is `deduce`, i.e., the classic Datalog query-answering procedure as implemented in [10]. The assimilation policy is `materialize_consequences`, i.e., the consequences computed are asserted into the rule base and the rule that was used for computing them is removed. This task flow is given as Figure 4.5 in the full version of the paper [1].

## 5 Related Work

A few other proposals to extend database technology with inductive capabilities have shaped, or are related to, the contributed solution reported. [3] was the first proposal to integrate deductive and inductive inference in a deductive database context. In contrast to the contributions reported here, however, there is no indication in [3] of how the combined functionality surfaces at the level of users. Further, it is not clear how [3] would cope with multiple (especially inductive) inference engines. [21] is primarily a proposal to integrate data-mining tools and deductive query answering, with a view towards deploying them as comprehensive data management systems, but only a fixed and closed set of hardwired functionalities is on offer, and hence one expects flexibility to be hard to achieve. This proposal also purposely leads to heavyweight systems. Web ser-

vice substitutivity is one of a class of applications for which flexible, dynamic and lightweight use of inferential engines seems very desirable.

Likewise, workflow systems [16], even if extended with the capability of invoking inferential engines, are neither likely to result in lightweight production systems nor would one expect much flexibility.

In terms of the application described, to the best of the author's knowledge, no other proposal exists to address the problem of characterizing web service substitutivity. There are, however, proposals in the literature for the engineering (e.g., [14]) and also for automating web service construction (e.g., [6]). Thus, the proposal here is best understood as complementary to those just cited.

## 6 Conclusions

This paper has proposed a lightweight mechanism to combine deductive and inductive inference engines and has described an application in which the resulting functionality is shown to add significant value. The value added by this functionality has been highlighted in a modern, challenging setting. To the best of the authors' knowledge, both the solution and the approach are novel. The concepts deployed in the solution offer similar application opportunities in many situations for which software agents are considered a useful implementation strategy.

**Acknowledgement.** Marcelo A. T. Aragão is on leave from the Central Bank of Brazil and gratefully acknowledges the scholarship awarded to him by the Department of Computer Science of the University of Manchester.

## References

1. M. A. T. Aragão and A. A. A. Fernandes. Applying combined inference engines. Available at <http://www.cs.man.ac.uk/~aragaom/>.
2. Ariba Inc., IBM Co. and Microsoft Co. UDDI technical white paper. Available at <http://www.uddi.org/whitepapers.html>, 2000.
3. F. Bergadano. Inductive Database Relations. *IEEE TKDE*, 5(6):969–971, 1993.
4. P. V. Biron and A. Malhotra. XML Schema part 2: Datatypes. Available at <http://www.w3.org/TR/xmlschema-2/>, 2001.
5. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. Available at <http://www.w3.org/TR/SOAP/>, 2000.
6. F. Casati, M. Sayal, and M.-C. Shan. Developing e-services for composing e-services. In K. R. Dittrich, A. Geppert, and M. C. Norrie, editors, *CAiSE 2001, Switzerland, 2001*, volume 2068 of *LNCS*. Springer, 2001.
7. G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM TOPLAS*, 17(3):431–447, 1995.
8. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming&Databases*. Springer, 1990.
9. E. Christensen, F. Curbera, G. Meredith, and S. Weerawa. Web services description language (WSDL)1.1. Available at <http://www.w3.org/TR/wsdl.html>, 2001.
10. S. K. Das. *Deductive Databases and Logic Programming*. Addison Wesley, 1992.

11. V. Draluk. Discovering web services: An overview. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *Proceedings of 27th International Conference on Very Large Data Bases (VLDB 2001), September 11-14, 2001, Roma, Italy*, pages 637–640. Morgan Kaufmann, 2001.
12. D. H. Fisher. Conceptual Clustering, Learning from Examples, and Inference. In *Proceedings of the 4th International Workshop on Machine Learning*, 1987.
13. IBM Web Services Team. Web services architecture overview. Available at <http://www-106.ibm.com/developerworks/library/w-ovr/>, 2000.
14. E. Kirda, M. Jazayeri, C. Kerer, and M. W. Schranz. Experiences in engineering flexible web services. *IEEE MultiMedia*, 8(1):58–65, 2001.
15. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
16. F. Leymann, D. Roller, and A. Reuter. *Production Workflow: Concepts and Techniques*. Prentice Hall, 1999.
17. F. Leymann. Web services flow language (WSFL)1.0. Available at <http://www4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>, May 2001.
18. J. Minker. Logic and databases: A 20 year retrospective. In *Workshop on Logic in Databases (LID'96)*, volume 1154 of *LNCS*, pages 3–57, Italy, July 1996. Springer.
19. T. Mitchell. *Machine Learning*. McGraw Hill, 1997.
20. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *LNCS*. Springer, Heidelberg, 1997.
21. E. Simoudis, B. Livezey, and R. Kerber. Integrating inductive and deductive reasoning for data mining. In U. Fayyad, G. P. Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.
22. S. Thatte. XLANG: Web services for business process design. Available at [http://www.gotdotnet.com/team/xml\\_wsspecs/xlang-c/](http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/), 2001.