Accepted at the International Logic Programming Symposium 1997

A Procedure for Mediation of Queries to Sources in Disparate Contexts

S. Bressan, C.H. Goh, T. Lee,
S. Madnick, & M. Siegel

Sloan WP# 3964 CISL WP# 97-09
June 1997

The Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02142

A Procedure for Mediation of Queries to Sources in Disparate Contexts

S. Bressan, C. H. Goh*, T. Lee, S. Madnick, and M. Siegel
MIT Sloan School of Management
50 Memorial Drive, E53-320
Cambridge, MA, 02139
email: {sbressan,chgoh,tlee,smadnick,msiegel}@MIT.EDU

Abstract

This paper discusses the algorithm we are using for the *mediation* of queries to disparate information sources in a *Context Interchange* system, where information sources may have different interpretations arising from their respective *context*. Queries are assumed to be formulated without regard for semantic heterogeneity, and are rewritten to corresponding *mediated queries* by taking into account the semantics of data codified in axioms associated with sources and receivers (the corresponding *context theories*). Our approach draws upon recent advances in abductive logic programming and presents an integration of techniques for query rewriting and semantic query optimization. We also demonstrate how this can be efficiently implemented using the constraint logic programming system ECLiPSe.

1 Introduction

Context Interchange [GBMS96a] is a novel approach towards the achievement of *semantic interoperability* of *heterogeneous information systems* [SL90, BHP92]. Using this strategy, queries to disparate systems can be constructed without regard for (potentially) conflicting representations or interpretations of data across different systems: for example, when comparing the room rates of two hotels on different sides of the US-Canadian border, the user asking the query need not be concerned with whether or not prices are reported using the same currency, or whether the prices reported are inclusive of applicable taxes. Loosely speaking, query mediation can be simplified to the following scheme: for a query expressed in the terms of a *receiver*[1], i.e under the assumptions and knowledge of the user or required by the application issuing the query, an equivalent query, in the terms of the component systems providing the data, must be composed, and a plan for the resulting query must be constructed, optimized and executed.

Our goal of this paper is to provide a logical interpretation of the query mediation step and to demonstrate how this is realized in a prototype implementation [BFG+97a] using the constraint logic programming system ECLiPSe. The inferences underlying query mediation can be characterized using an *abductive framework* [KKT93a] and points to some interesting connection between integrity constraint checking and classical work in *semantic query optimization* [CGM90].

*Current address: Department of Information Systems & Computer Sc, National University of Singapore, Kent Ridge, Singapore 119260. Email: gohch@iscs.nus.sg

[1] which may be a user or an application

The remainder of this paper is organized as follows. In section 2, we summarize the novelty of our approach and describe the translation between a COIN (COntext INterchange) framework and a program (or, logical theory) written in COINL (the COIN language). In section 3, we present and discuss the Abductive Logic Programming framework and some aspects of the duality between abduction and deduction. We set the requirements for our procedure and discuss the possible interpretations of its results. In section 4, we outline the algorithm and discuss its implementation in the Logic Programming environment ECLiPSe [ECL96]. In particular, we discuss the implementation of the consistency checking phase of the abductive procedure as *Constraint Logic Programming* propagation [JM96] using *Constraints Handling Rules* [FH93]. We also discuss the relationship between consistency checking with integrity constraints and *Semantic Query Optimization* [CGM90]. Finally, we conclude in section 5 on some more general aspects of our project and on future work.

We assume that the reader is familiar with notations of first order logic as defined, for instance, in [Llo87]. We refer the reader to [KLW95a] and [GBMS96a] respectively for formal definitions of the F-Logic and the syntax and semantics of our logical language COINL. Where appropriate, we will explain those constructs used in the examples which are necessary for the understanding of the discussion. Finally, we use the symbol $\models$ to represent logical consequence in the model theory and the symbol $\vdash$ for the application of an inference rule (the acronym of the rule is subscripted when ambiguous).

2 Context Mediation

Before describing what context mediation entails, it is necessary to provide a summary of the motivation behind the architecture of a Context Interchange system, presented in the form of a COIN *framework*. Following which, we introduce an example which illustrates what context mediation entails. Finally, we show how the different components of a Context Interchange system can be used in the construction of a logical theory which may take the form of a *normal (Horn) program* [Llo87].

2.1 The Context Interchange Framework

Traditionally, two different approaches have been adopted for providing integrated access to disparate information sources. The *tight-coupling* approaches to semantic interoperability rely on the *a priori* creation of federated views on heterogeneous information sources. Although they provide better support for data access, they do not scale-up efficiently given the complexity inherent in the construction and maintenance of a shared schema for a large number of autonomously administered sources. *Loose-coupling* approaches rely on the user's intimate knowledge of the semantic conflicts between the sources and the conflict resolution procedures. This flexibility becomes a drawback for scalability since the size of this knowledge increases exponentially with the number of sources, and may require frequent revisions as the semantics and structure of underlying sources undergo changes.

The Context Interchange approach takes a middle ground between these two approaches. Unlike the loose-coupling approaches, queries in such a system need not be concerned with differences in data representation or interpretation across sites; i.e., it allows queries to be formulated on multiple sources *as if* these were fragments of a homogeneous distributed database. Although it requires a common lexicon (called a *domain model*) for disambiguating types and role names, it is no longer mandatory for all conflicts to be resolved a priori in one place (e.g., as in the tight-coupling approaches). Instead, sources and receivers need only provide a declarative specification of the semantics of data pertaining to itself, while deferring conflict detection and resolution to the time when a query is actually submitted (when the sites involved in data exchange is identified). The *Context Mediator* takes on

the role of conflict detection and resolution: this process is referred to as *context mediation.* A more detailed comparison of the loose- and tight-coupling approaches, and the relative advantages of Context Interchange can be found in [GMS94].

To support the functionalities described above, information about data sources and the semantics of data therein are captured in a Context Interchange system in a slightly more complex way. Loosely speaking, a Context Interchange system, as characterized by a COIN framework, comprises of the following:

- a *domain model*, which provides a lexicon of types and *modifiers* corresponding to each type;
- a collection of *sources* corresponding to heterogeneous extensional databases (which we assume to be relational without any loss of generality);
- a collection of *elevation theories*, each comprising of a collection of elevation axioms which define the types corresponding to the data domains in a source. Elevation axioms are source-specific: i.e., each source is bound to one and only one elevation theory;
- a collection of *context theories*, each of which is a collection of declarative statements (in COINL) which either provide for the assignment of a value to a modifier, or identify a *conversion function* which can be used as the basis for converting the values of objects across different contexts; and finally,
- a mapping function `mu` which maps sources to contexts: in essence, this means that several different sources can share the same context. This feature not only provides for greater economy of expression but also allow context theories to be nested in a hierarchy, facilitating the reuse of context axioms[2].

Our primary motivation behind this structuring of a Context Interchange system is to provide the transparency of tight-coupling systems without the burden of reconciling all conflicts in one or more federated views. We argue that by allowing data semantics to be declaratively and independently captured in the form of context theories, changes in a local site can be better contained: in most instances, these changes require only modification of the context theory pertaining to the given site and have no repercussions on the global system. An interesting "side-effect" is that receivers too can have context: by associating a query with a context theory, we can request for answers to be returned in a form that is meaningful with respect to the stated context. Finally, notice that evolution in membership of sources have no effect the system: the addition or retraction of a source only involve the introduction or retraction of the corresponding elevation theory (and possibly the introduction of a new context theory). This compares favorably to current tight-coupling systems where the view definition needs to be modified corresponding to every of these changes.

2.2 Example

We consider a simple example where a user poses a query to a source (`security`), which provides historical financial data about a stock exchange. The user and the source have different assumptions regarding the interpretation of the data. These assumptions are captured in their respective contexts $c1$ and $C2$. The Domain Model defines the semantic types `moneyAmount`, `date`, `currencyType`, and `companyName`. The following query requests the price of the IBM security on March, 12^{th} 1995:

```
Q1: select security.Price
    from security
    where security.Company = "International Business Machines"
    and security.Date = "12/03/95";
```

[2] A more detailed discussion can be found in [Goh97]

Suppose the user's context $c1$ indicates that money amounts are assumed to be in French Francs, dates are to be reported in the European format, and that currency conversions should be based on the date for corresponding to that for which the price is reported for. Notice that this information is at least needed to avoid the confusion between March, 12^{th} and December, 3^{rd} 1995.

Let us assume, on the other hand, that the source context $C2$ indicates that money amounts in the source are in local currencies of the country-of-incorporation of each company, and dates are reported in the American format. Under these circumstances, the Context Mediator will rewrite the query to incorporate the proper currency conversion (as of March, 12^{th} 1995, using ancillary source (`cc`) for the conversion rates), and also make the appropriate transformations on dates to ensure that the query is correctly interpreted. In addition, if both contexts use different naming conventions for companies involved, then appropriate mapping between two naming conventions will be needed. For example, $c1$ may assume the full company name ("International Business Machines") while $C2$ uses company ticker symbol ("IBM"). Under the above circumstances, the mediated query corresponding to Q1 will be given by Q2 as shown below:

```
select security.Price * cc.Rate
       from security, cc
       where security.Company "IBM"
       and security.Date = "03/12/95"
       and cc.source = "USD"
       and cc.target = "FRF"
       and cc.date = security.Date;
```

A primary goal of this paper is to demonstrate that the kind of transformations which we have exemplified can be understood as a special type of logical inference, which is sometimes characterized as *abduction* [KKT93a]. The next section describes the representations in a Context Interchange framework in somewhat more detail, and illustrate how this can be transformed to a logical theory (comprising only of normal Horn clauses), thereby setting the stage for the subsequent sections of this paper.

2.3 A Logical Interpretation of Context Mediation

The axioms in a Context Interchange framework are represented using a deductive object-oriented formalism called COINL, which is a variant of F-logic [KLW95b]. The collection of axioms present in an instance of this framework constituted a COINL program. The mediation of queries submitted to a Context Interchange system proceeds as follows. First, the user query and the COINL program are compiled into a goal (a negative Horn clause) and a normal logic program respectively. Following this, the goal is evaluated against the logic program using th the *mediation procedure* (which we will describe shortly) which returns a set of conjunctive clauses and a set of variable substitutions associated with each such clause. These are then translated back to SQL, which can then be evaluated using a distributed database engine. In the remainder of this discussion, we will focus on the query mediation procedure. An in-depth description of the query evaluation framework can be found in [GBMS96b].

Figure 1 illustrates the approach taken towards the representation of contextual information. Each data element, be it stored in a source database, or expected or expressed by a receiver, corresponds to a *semantic-object* or equivalently, an instance of a *semantic-type* defined in the domain model. In the above figure, we show only a portion of the domain model with the semantic-types **`moneyAmount`** and **`curType`** (for currency type). The semantic types may be arranged in a type hierarchy. The information on what types exists, their relationship to one another, and the signature of methods (called *modifiers*) defined on each

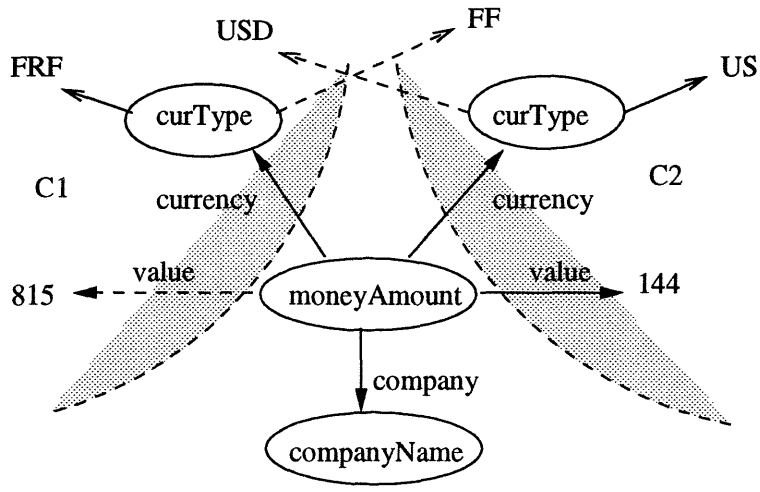

Figure 1: A summary of the COIN Framework.

type are defined in the domain model using the COIN language (COINL, examples of which are shown below:

```
moneyAmt :: number.
curType :: string.
moneyAmt[currency(ctx) => curType].
```

The above assertions state that **moneyAmt** is a subtype of **number**, whereas **curType** is a subtype of **string**. In addition, the semantic-type **moneyAmt** is modified by a method **currency** which returns an instance of the type **curType**

Semantic-objects are instantiated through the use of elevation axioms. For each relation R exported by a source, a *semantic relation* **R'** is defined. The semantic relation **R'** is *isomorphic* to the extensional relation R in the sense that it has the same *arity* m (i.e., number of arguments) and for every tuple $r(x_1, \ldots, x_m) \in R$, there exists a tuple **r'**$(o_1, \ldots, o_m) \in$ **R'** such that the value of o_i in the context corresponding to R is given by x_i. For instance, for the relation **security(company, price)**, the corresponding semantic relation is given by **security'(companyName, moneyAmt)**, where **companyName** and **moneyAmt** are semantic-types. Semantic-objects are syntactic constructs (Skolem functions) of the relation, the attribute, and the tuple. For instance:

O_1 = **f(security, company, ["IBM", 144, "03/12/95"])**.

is a semantic-object. The value of this object in the source context (of the relation **security**) is **"IBM"**. This is expressed in the elevation by the following axioms. The first axiom defines the semantic relation and the semantic objects for **security**. The three other axioms define the respective values and attributes of the semantic objects. Notice that the value is a function of the context. The context associated with **security** is given by the **mu** function. **mu** is defined for each source when the source joins the mediation system.

```
security'( f(security, company, [N,P,D]),
           f(security, price, [N,P,D]),
           f(security, date, [N,P,D]) ) <- security(N,P,D).
f(security, company, [N,P,D]):companyName
           [value(C)->N]<- mu(security, C).
```

```
f(security, price, [N,P,D]):moneyAmount
          [value(C)->P] <- mu(security, C).
f(security, date, [N,P,D]):date
          [value(C)->D]<- mu(security, C).
```

Notice that semantic-objects are "virtual" in the sense that they are never actually instantiated but are present solely for the purpose of allowing us to reason with the different meanings of syntactic tokens which may appear identical.

In addition to the above, integrity constraints on sources can also be introduced to facilitate their use in semantic optimization of the mediated queries. For example, we may introduce the functional dependency `company` $\rightarrow$ `price` via the following assertion:

```
security(N, P1, D), security(N, P2, D) -> P1 = P2.
```

As we will see later, such constraints allow superfluous references to extensional data sources to be pruned and can result in significant savings.

Intuitively, a semantic-object is a syntactic construction which allows information in a source to be abstracted the peculiarities of its representation and from the assumptions underlying its interpretation. For example, the same semantic-object may have different "values" in different context because different currencies are used for their reporting. In general the assumptions can be characterized by a number of orthogonal concepts (e.g. unit, scalefactor, format, rounding etc). To each semantic-type, and therefore to each semantic object, we associate a *modifier* corresponding to each of the relevant notion defining the interpretation of the data. On Figure 1, the semantic-object o of type **moneyAmt** has a modifier currency which may take on different values (e.g., "French Francs" or "US Dollars") in different contexts.

A *context theory* is the set of definitions for the modifiers corresponding to each semantic type in the domain model (context inheritance allows the reuse and specialization of context definitions). In our example, the context `c1` and `c2` define the *currency* modifiers as returning `string` (also a semantic-type) whose value will be respectively "United States Dollar" and "Japanese Yen". Notice that the modifier are in turn assigned semantic-objects whose values are to be interpreted in a context. Indeed, different contexts may represent the data element for United States Dollar as, for instance, "USD", or "$", or "US", etc. In complex situations we may use *modifiers of modifiers*. Let us consider a simple situation and give the axioms for the modifier *currency* in the context `c1`.

```
X : moneyAmount[currency(c1)->currency(c1,X)].
currency(c1,X):string[value(c1)->"FRF"].
```

Notice that the modifier is a function of the context. The semantic-object `currency(c1,`X`)` (assigned to the modifier `currency` of X) is created by the first axiom. The second axiom assigns the value of this object which must is a printable string.

Conversion functions define the the mapping of a value to its corresponding counterpart under different assumptions, i.e for different values of the modifiers. Typically, a conversion function for the currency conversion is multiplying the money amount by the currency exchange rate. Administrators and users contribute to a library of conversion functions.

The process of mediation consists, for each data element encountered in mediating a query, in the following steps. The source and associated context of the corresponding semantic object are identified. They are retrieved from the information in the skolem function identifying the semantic object. The target context, i.e. the context in which the data element needs to be interpreted is identified. It is primarily given by the context in which the query is asked. Then the modifier values of the semantic objects are compared. For each mismatch in the modifier values, the corresponding conversion function is introduced. This mechanism is expressed by built-in axioms.

The user queries the relations exported by the component sources. However the query is expressed under the assumptions of the receivers, i.e. in her context. The query (Q1 in the example from the introduction section) needs to be rewritten to take these assumptions into account. The mediated query Q1' is:

```
answer(P) <- security'(O1, O2, O3),
             O1[value(c1)->"International Business Machines"],
             O3[value(c1)->"12/03/95"],
             O2[value(c1)->P].
```

Q1' expresses the user's intention to query about the values of the semantic objects in the relation *security'* in her context.

The query constitutes a goal for the program composed of all the basic axioms, elevation axioms and context axioms. It is evaluated by the mediation procedure. The answers are the mediated queries which only contain atoms from the component databases and the conversion functions as we have shown in the introduction.

We have chosen to implement the mediation process by separating it into a general purpose procedure and the explicit set of axioms. The generic mediation axioms and the application contexts, conversions, and elevation axioms are expressed in COINL and compiled into Datalog [CGT90]. The alternative of specializing the procedure by hard coding the generic axioms is attractive. However by maintaining the separation clear we achieve another objective: we provide a uniform query mechanism for both data level query mediation, i.e. queries to the sources as illustrated in the previous examples, and knowledge level queries, i.e queries involving elements from the domain model and contexts. For instance we can ask the following query: "what are the names of the companies reported in the security relation and in which currency are their stocks reported in context c1?".

3 Abduction

3.1 Abduction

Abduction [KKT93b] is a form of reasoning initially defined by C.S. Pierce as the inference of the case from the rule and the result. For example, from the observation of G and the knowledge of the rule $D \rightarrow G$, one can infer D by abduction.

$$G \wedge (D \rightarrow G) \vdash_{abd} D$$

Let us, more generally, define the abduction inference in terms of the model semantics, where T is a set of sentences constituting our theory, G is the sentence describing our observation, and D is the

$$(T \wedge G) \vdash_{abd} D$$

if and only if

$$(T \cup D) \models G$$

Informally, we could say that D is a *"logical antecedent"* of G under T.

In order to make a practical use of abduction as a reasoning mechanism, we also want to avoid non-constructive inferences such as $G \vdash_{abd} D$ where D contains literals which are not "interesting" for the application. A trivial example could be a situation where D is G itself. We therefore add to our definition the condition that only certain literals are acceptable in the sentence D. Such literals are called *abducible literals*. They are identified as such from their predicate name which are explicitly defined as *abducible predicates* and be declared in a set called P_{abd}.

Furthermore we notice that a formula F such that $T \cup F$ is inconsistent ($T \cup F \models \square$) is a logical antecedent of anything. Indeed, for all formulae G: $\square \models G$ and therefore $T \cup F \models G$.

We add to our definition the requirement that $T \cup F$ is consistent. Thus, we can enrich this framework by considering a set of integrity constraints IC, i.e a set of formulae that are statements about the universe of discourse. We assume that these statements are consistent with our theory T. We also require in our definition that $T \cup IC \cup D$ is consistent.

We can now give the complete definition for the abduction framework, which is consistent with that given elsewhere (e.g., [KKT93a]): Given a set of sentences T called a theory, a set of sentences IC called integrity constraints, a sentence G called the observation, and a set of predicates P_{abd}, given that $T \cup IC$ is consistent, we say infer D by abduction from T and G under IC:

$$(T \cup G) \vdash_{abd}^{(IC)} D$$

if and only if

- $T \cup D \models G$;
- $T \cup IC \cup D$ is consistent; and
- all predicates used to form literals in D belong to P_{abd}.

3.2 Applications of Abduction and Abductive Procedures

In [KKT93b], Kakas and Kowaslki present the abductive framework and a survey of recent work on abduction. Abduction has been applied to many categories of problems such as *design synthesis* [FG85], *planning* [CP86, Esh88], or *database updates* (view and intensional updates) [KM90, Bry90, Dec]. In fact, as recalled in [KKT93b], the *Abductive Logic Programming* framework combines many aspects of deduction, truth maintenance, non monotonic reasoning, and default reasoning and can serve as a general problem solving programming environment. Many of these work observe the duality between deduction and abduction [Sha89].

An important aspect of the abductive framework is the notion of integrity constraints and integrity checking. It offers the opportunity to include semantic query optimization and constraint logic programming features into the inference mechanism.

Wetzel, Kowalski, and Toni [WKT95] have presented a *Theorem Proving* approach to *Constraint Logic Programming* which attempt to unify abductive logic programming, constraint logic programming, and semantic query optimization. One implementation of this framework is the *Procalog* [WKT96] programming language. This work as been developed in parallel with our effort. It focuses on a general programming environment while we have concentrated on mediation.

Several procedures have been developed for the abductive framework.

The residue procedure [FG85] is a resolution based procedure that operates on clausal programs. The procedure includes inference steps for both abduction and constraint propagation. We borrow and adapt an example from [FG85] in appendix A.

In [Esh93], Eshghi defines a suitable notion of minimality for abductive answers and prove that there exists a polynomial algorithm for computing them when programs are acyclic propositional horn theories. His algorithm is based on unit resolution which has been proven equivalent to input resolution [Cha70] (For a general discussion and references on resolution see [BJ87]).

An issue which we do not address in this paper but is useful to consider for the application of abduction to non monotonic reasoning such as planning and database updates, is the management of negation [Bid91]. Several semantics and procedures have been developed to handle this problem. We refer the reader to [EK89], [Dec], and [DS92] for instance.

3.3 An Abductive Framework for Context Mediation

The COINL program resulting from the definition of a domain model, elevation axioms and contexts for the integration of a set of disparate information sources can be equivalently transformed to a normal Horn program (equivalently, a Datalog^{neg} program). Similar transformations of object-oriented logical formalisms to predicate calculus based languages have been described in several places (for example, [ALUW93, McC92]). Let us call this program T.

In this paper, we consider an integrity theory composed of integrity constraint statements expressed in *Datalog*^{neg}[3] on the exported schemas of the sources. We restrict ourselves to constraints of the form $l_1 \wedge ... \wedge l_n \rightarrow l_0$ where the l_i are atoms and l_0 is a constraint literal. This form is equivalent to $l_1 \wedge ... \wedge l_n \wedge \overline{l_0} \rightarrow$ where $\overline{l_0}$ is $\neg l_0$. Integrity constraints are Horn clauses without positive literals, i.e. denials. Let us call the set of integrity constraints IC.

A query to be mediated is transformed into a COINL and further into a Datalog query. This transformation is not a logical transformation. The query is interpreted with respect to the user contexts. We refer the reader to the example in the previous section. Let us call the Datalog rule Q. Let assume that Q is of the form $G \rightarrow answer(\vec{X})$ where $\vec{X}$ is the vector of variables projected out.

Let us assume that the set of data in the disparate information sources is a single database DB. G is a query on the deductive database $< P, DB >$ where P is the intensional database and DB the extensional database. $DB \cup IC$ is consistent by definition of IC. An answer to the query G is a substitution θ such that $T \cup DB \models \theta(G)$. Such an answer is usually found by refutation: one looks for θ such that $T \cup DB \cup \neg\theta(G) \models \square$. For Horn clause programs, resolution is a complete inference rule for refutation [Llo87, BJ87]. For non recursive Datalog programs, SLD-resolution is a complete inference rule for refutation [GM78, CGT90]. A SLD-resolution proof will also construct the substitution θ:

$$T \cup DB \cup \neg\theta(G) \vdash_{SLD} \square$$

Let us assume that we want to separate the resolution with clauses from T from the resolution with facts in the database DB. In other words, let us assume that we want to rewrite the query according to the intensional database before we access the extensional database. Since T is a non recursive program, we should be able to achieve this objective by performing some kind of unfolding of the query against the program only keeping certain literals. We need to slightly modify the selection function of the SLD-resolution. We need to decide to postpone indefinitely the resolution of literals in the query which correspond to literals of the extensional database DB and delaying the resolution of the constraints unless they are ground (when they can be evaluated e.g. $2 > 1$). Once all the possible resolutions have been attempted, we remain with a resolvant containing extensional database literals and constraints literals that could not be evaluated because of they are non ground.

For the program $T = \{p(X,Y) \wedge r(X,Z) \rightarrow q(X,Y,Z), p(a,Y) \wedge Y > 10 \rightarrow r(a,Y)\}$ and the extensional database relation $p(X,Y,Z)$ and the query $q(U,9,V)$, the modified SLD-resolution we are discussing stops with the resolvant

$$p(a,9), p(a,Z), Z > 10$$

and the substitution $\theta = \{U/a, V/Z\}$. A resolvant $\neg G'$ obtained with that procedure is such that: $T \cup \neg\theta(G) \vdash_{SLD} \neg G'$ and therefore: $T \cup \neg\theta(G) \models \neg G'$ or $T \cup G' \models \theta(G)$.

In other words, if we describe the literals of the extensional database as abducible, and consider G as an observation for an abductive proof, G' is an abductive answer for G against T. we have:

$$T \models G' \rightarrow answer(\theta(\vec{X}))$$

[3] Datalog^{neg} is a subset of COINL. (For brevity, we will refer to the target language as simply Datalog.)

The union of all the formulae $G' \rightarrow answer(\theta(\vec{X}))$ (the mediated queries) for G' and θ corresponding to each successful branch of the modified SLD-resolution is a program which is equivalent to T for the processing of Q against the extensional database. The completeness of the result is given by the completeness of SLD-resolution for refutation of the type of programs we consider.

If now we restrict the answers G' to those consistent with the integrity constraints, given a sound consistency procedure, we are filtering out some mediated queries which would result in an empty answer if evaluated against the extensional database. It must be clear that in the context of mediation of queries to disparate information sources, where the sources are remote, such an elimination of useless network access is a crucial optimization. In addition, any propagation of the constraints that can be performed in the process of consistency checking can also increase the performance of the mediation service by pushing more selections to the remote sources and potentially leading to smaller amounts of data transported over the network.

We are indeed talking about a logical optimization. If the consistency test is not performed, the subsequent query evaluation would still provide sound and complete answers. For this reason we can satisfy ourselves with a sound test as opposed to a sound and complete test. We accept a looser third condition in the definition of the abductive framework.

In our example we now consider a functional dependency integrity constraint on p expressed in Datalog by the clause: $p(X,Y) \wedge p(X,Z) \wedge Y \neq Z \rightarrow$, meaning Y and Z cannot be different for the same value X of the first attribute of p. The resolvant $p(a,9), p(a,Z), Z > 10$ can be simplified in three stages. First the integrity constraint is used to determine that Z must be equal to 9: $p(a,9), p(a,9), 9 > 10$. Second, one of the two syntactically identical literals $p(a,9)$ can be eliminated: $p(a,9), 9 > 10$. Third, the constraint solver for inequalities on integers (e.g.) figures out that $9 > 10$ is inconsistent. The resolvant is inconsistent with the integrity constraints and can be rejected.

4 The Procedure

The procedure we propose is therefore a modified SLD-resolution where the literals corresponding to constraints or relations in the remote data sources are not evaluated. From the point of view of abduction, they are abducible (`abducible` boolean function below).

The definition of the abductive framework suggest an algorithm which generates the candidate abductive answers and subsequently tests the consistency against the integrity constraints. Following the Constraint Logic Programming framework [JM96, Wal96], we argue that, if the consistency testing can be done incrementally during the construction of the SLD-tree, we are likely to have an improvement of the performance of the algorithm. This is a heuristic which depends on the shape of the proof tree.

We replace the *generate and test* procedure by a *constraint and generate* procedure [JM96]. From such a point of view, the resolvant is a constraint store whose consistency is maintained by a propagation algorithm.

Figure 2 is a sketch in pseudo code of the main algorithm of the abduction procedure. `.tail`, `.head` respectively access the tail and the head of a list or the body and the head of a horn clause. `[]` is the empty list. `append` and `add` respectively append two list and add an element to list. We assume that lists can have a boolean value `False` different from `[]` or other values of lists with elements.

The input `Goal` is a list of atoms corresponding to the initial query (a conjunctive query). Initially, the store `Store` (the data structure used for propagation) is empty. `Rules` is a data structure containing the program. `Abducted` contains the result of the procedure. It is the list of lists that contains the conjunctive components of the mediated query. It is initialized to `[]`. The unification procedure `unify` return a substitution in the form

of a list of equalities. If the element do not unify it returns **False**. The propagation procedure, **propagation** tests the consistency of the store and returns **false** if it detects an inconsistency. Otherwise, it returns the store which may have been modified by the propagation.

The idea of the algorithm is to traverse the resolution tree. We opted for a depth first traversal. This basic component is implemented by the **if** and **elsif** clauses. When the goal is emptied (**if** clause), a leaf of the resolution tree is reached. One can collect the answer from the store. Indeed, all abducible atoms have been posted to the store in the **else** clause. Notice that algorithm assumes that abducibles can not be heads of rules.

```
Procedure abduct(Goal, Store){
   if (Goal eq [])
      {Abducted := append(Abducted, Store);}
   elsif (!(abducible(Goal.head))
      {foreach Rule (Rules))
      {Substitutions := unify(Rule.head, Goal.head);
      if (Substitutions)
         {NewGoal := Goal.tail;
         Store := add(Store, Goal.head);
         Store := propagation(Store);
          if (Store) {abduct(NewGoal, Store);}}}}
   else (abducible(Goal.head))
      {NewGoal := Goal.tail;
      Store := add(Store, Goal.head);
      Store := propagation(Store);
      if (Store) {abduct(NewGoal, Store);}}
}
```

Figure 2: Pseudo Code Procedure

Variant algorithms for this procedure correspond to variant strategies for the traversal of the proof tree. For a depth first strategy, they correspond to the various Prolog meta interpretors described in the literature [SS94]. As a matter of fact the procedure is straightforward in Prolog, since the basic control of Prolog is exactly what we are trying to realize here. Any of the classical vanilla interpreter can be used and extended as, for instance, the one of figure 3. We will see different implementations of the propagation techniques in the next subsection.

4.1 Implementation of Constraint Propagation and Consistency Checking

In both the algorithms we have presented, the propagation procedure implements the consistency test we have discussed in the previous section. This procedure either returns the store, possibly modified, if no inconsistency is detected, or interrupts the search tree traversal (**False** or failure) if an inconsistency is detected.

The minimum set of constraints to be considered are Clark's Free Equality axioms, i.e. the axioms defining the consistency of a set of equations between variables and constants. They will manage the consistency of the store with regard to the substitutions produced by the **unify** procedure. However, the unification procedure could include this test as it is the case in the second algorithm. Here, we rely on Prolog's unification not only for creating the substitutions, but also for applying the substitution and producing a failure

```
abduct(Goal, Result) :-
  setof(Store, sub_abduct(Goal,[], Store), Result).

sub_abduct([], Store, Store).
sub_abduct([Head|Tail], StoreIn, StoreOut):-
           rule([RHead|RTail]),
           append(RTail, Tail, NewGoal),
           propagation(StoreIn, Store),
           sub_abduct(NewGoal, Store, StoreOut).
sub_abduct([Head|Tail], StoreIn, StoreOut):-
           abducible(Head),
           propagation([Head|StoreIn], Store),
           sub_abduct(Tail, Store , StoreOut).
```

Figure 3: First Prolog Vanilla

we an inconsistency occur. Notice that, when constraints are posted in the store, the store may generates new equations. For instance if it contains the two literals $p(a, b)$ and $p(a, X)$ and knows a functional dependency of the second argument of p from the first: $p(X, Y_1), p(X, Y_2) \rightarrow Y_1 = Y_2$, the equation $b = X$ is propagated. We may limit the propagation to within the store, but we may also decide to propagate it outside the store in order to take it into account in the subsequent phases. In Prolog, this is achieved by unifying X with a. The advantage is that the result of such a propagation can be accounted in the next rule selection and unification phase of the resolution.

In general a consistency procedure for the class of integrity constraints we propose to use can be implemented by means of a production system where for a constraint $B(\vec{X}) \rightarrow A(\vec{Y})$ where $\vec{Y} \subset \vec{X}$. The consistency procedure controls the application of the propagation rules in a fixpoint iteration. We have been using the *Constraint Handling Rules* (CHR) library [FH93] of the ECLiPSe parallel logic programming platform.

CHR is a language extension of ECLiPSe for the definition of of constraint solvers. CHR is a rule based language. Rules in the program correspond to individual propagations operated on the store. The host language, Prolog, posts constraints into the store. The propagation is automatically triggered by the posting of a new constraint or an event such as the unification of a variable involved in the store. Figure 4 shows the possible forms of Constraint Handling Rules.

A simplification rule: `Head [, Head] <=> Body.`. Such a rule replaces a part of the store matching the left hand side by the right hand side.
A propagation rule: `Head [, Head] ==> Body.`. Such a rule adds the right hand side to the store whenever some element in the store match the left hand side.
A "simpagation" rule: `Head \ Head <=> Body.`. Such a rule is a short hand for a combination of propagation and simplification.

Figure 4: Constraint Handling Rules

An example of a constraint domain that can be implemented with the CHR is for instance inequalities. Figure 5 gives a simplified excerpt of the code from the "lower or equal", `leq`, constraint from [ECL96] by Thom Frühwirth and Pascal Brisset. Each rule is prefixed by its name and the separator `@`. The body of the rules may contain a guard, a procedural

escape, before the symbol |.

```
reflexivity  @ X leq X <=> true.

antisymmetry @ X leq Y, Y leq X <=> X = Y.

transitivity @ X leq Y, Y leq Z ==>  X leq Z.

subsumption  @ X leq N \ X leq M <=> N<M | true.
subsumption  @ M leq X \ N leq X <=> N<M | true.
```

Figure 5: Constraint Handling Rules for Inequalities

The only implicit elimination we are considering is the duplicate elimination which is optionally performed by the CHR engine.

Each integrity constraint of the form $B(\vec{X}) \rightarrow A(\vec{Y})$ where $\vec{X} \subset \vec{X}$ is compiled into a CHR propagation rule. For instance, the functional dependency $p(X, Y_1) \wedge p(X, Y_2) \rightarrow Y_1 = Y_2$ is compiled into `p(X, Y1), p(X, Y2) ==> Y1 = Y2`.

It is sufficient to declare the different abducible predicate as constraints and to post them into the store (`post/1` which is then equivalent to the Prolog built-in `call/1`). The consistency checking and constraint propagation will be performed automatically and triggered, as we wished, by constraint posting and unification. When the search tree traversal terminates, the constraints are collected from the store by the predicate `store/1`. The revised procedure now is shown on figure 6.

```
abduct(Goal, Result) :-
  setof(Store, abduct_and_collect(Goal, Store), Result).

abduct_and_collect(Goal, Store):-
           sub_abduct(Goal),
           store(Store).

sub_abduct([]).
sub_abduct([Head|Tail]):-
           rule([RHead|RTail]),
           append(RTail, Tail, NewGoal),
           sub_abduct(NewGoal).
sub_abduct([Head|Tail], StoreIn, StoreOut):-
           abducible(Head),
           post(Head),
           sub_abduct(Tail).
```

Figure 6: Second Prolog Vanilla

We see that normal SLD-resolution is not only the skeleton of the search tree traversal strategy but is actually implemented directly by the first and second clauses of `sub_abduct/1`. In appendix A we illustrate the application of this general procedure with a simple classical example not related to mediation.

4.2 Semantic Query Optimization

Semantic Query Optimization as described in [CGM90], is the process of optimizing (increasing the potential for an efficient evaluation) of database queries using the semantic information contained in the integrity constraints.

A query may be answered without accessing the database if enough knowledge is contained in the integrity constraints. For instance, a query requesting the names and security prices of companies whose pre-tax earnings are lower than $2.5 million and which take part in the Dow Jones definition may be answered (no such company exists) on the premise of integrity constraints. The constraints are stating that a company must have earned more than $2.5 million before tax to be listed in the New York Stock Exchanged (NYSE) and that the Dow Jones is exclusively composed of companies from the NYSE.

When accessing the database can not be avoided, the potential for the evaluation of a query can still be improved by the introduction or elimination of elements. A query requesting the price of the securities participating in the definitions of both the Dow Jones and the Value Line Composite Index can restrict its access to the relations reporting prices from the NYSE because of the constraint on the Dow Jones (the Value Line Composite Index combines securities from NYSE, AMEX and other markets).

King has identified six types of transformations that can result from the use of integrity constraints knowledge:

1. Index Introduction: the specialization of one element of an accessed relation that can be used to improve the indexed access;
2. Join Elimination: the elimination of a redundant access to a relation (typically in the case of an inclusion dependency);
3. Scan Reduction: the introduction of a constraint (e.g. $company.netsales < 25000000$) which may optimize the scanning of the relation;
4. Join Introduction: the introduction of an access to a relation which may be used to accelerate the elimination of tuples (e.g. a join index [VB86]);
5. Detection of Unsatisfiable Conditions: cases where the result of the query will be empty regardless of the database content.
6. No Transformation: cases where no transformation is useful even if possible.

We see that introduction and elimination are opposite actions and that a strategy must be define for the utilization of the constraints.

Chakravarthy et al. [CGM90] present a method for compiling integrity constraints into residues attached to the rules (view definitions) of a deductive database program and show how to exploit the residues at the query optimization stage to identify opportunities for the application of one of the six cases. The compilation of the residue for a rule is based on a partial subsumption algorithm. The different types of transformation proposed by King are shown to correspond to different possible transformations corresponding to the type of residue obtained. There are four different types of residue: the null clause, a goal clause, a unit clause, a Horn clause with non-empty body and head.

In comparison, our algorithm is based on a plain subsumption because it is used during the resolution rather than at compile time. There is no significant loss in efficiency: indeed, our constraint propagation also implement the decision of which transformation is to be made. This is possible because of the current restrictions we made to the integrity constraints we are using. We are mainly aiming at the detection of inconsistencies which, in our case, not only optimize the subsequent evaluation of the query but also optimize, because of the early detection in the traversal of the search tree, the mediation process itself. We need however to guarantee that the constraint solver we use is sound and converges.

Minimality of the result and completeness of the solver are wishable properties but are not necessary. In general we are considering to use off-the-shelf CHR libraries.

For the above reason, we only compile integrity constraints into propagation rules which propagate constraint literals.

However, it is clear that, if we can decide of a strategy to apply other transformations such as join introduction or join elimination, we can compile the integrity constraint into simplification or "simpagation" constraints handling rules. For instance let us assume a situation where we are querying two relations in two separate sources:

- $r_1(cname, revenue, currency)$, a relation reporting for several companies their revenue and the currency the revenue is expressed in;
- $r_2(cname, ticker, last)$ a relation reporting for the same companies as above their ticker name (the company identification code in the stock exchange) and the last price of the share.

Let us consider the query:

```
select r1.cname, r2.last from r1, r2 where r1.cname=r2.cname;
```

An integrity constraint expressing that the correspondence between company names and tickers is also accessible from a third relation $r_3(cname, ticker)$ can be used to generate the transformed query:

```
select r1.cname, r2.last from r1, r2, r3
       where r1.cname=r2.cname
       and r3.cname = r1.cname
       and r3.ticker = r2.ticker;
```

The constraint in Datalog: $r_1(N, R, C) \wedge r_2(N, T, L) \rightarrow r_3(C, N)$ is compiled into a propagation rule of the form `r1(N, R, C), r2(N, T, L) => r_3(C, N).` which will add $r_3(C, N)$ to the store whenever r_1 and r_2 tuples matching the rule are found.

Such a join introduction is not only interesting if r_2 is indexed on the ticker value but also, as it is the case with information sources such as wrapped web sites [BL] or on line services, when the capabilities of the source are limited [BFG+97b, LRO96, PGGMU95, Ce95]. In this case the auxiliary relation r_3 provides a means to generate values for the ticker before querying r_2. This type of use of the semantic query optimization mechanism available in our system is part of our future plans. The question being to determine the strategy guiding the compilation of integrity constraints.

5 Conclusion

We have gathered results coming from several different research areas, theorem proving and abductive logic programming, constraint solving and constraint logic programming, deductive databases and semantic query optimization, and we have applied them in the unifying framework of abductive logic programming to design a procedure for the mediation of queries to disparate information sources.

The resulting algorithm is powerful yet it has a particularly simple implementation (Figure 6) when one uses state of the art, industrial strength, constraint logic programming technology.

Such a result comes from the fact that the logic of mediation is declaratively encoded into a COINL program which is the control of the procedure. This was made possible by the deductive and object oriented features of the COIN language which constitutes an appropriate and and expressive support for both the representation of semantic knowledge and the reasoning about semantic heterogeneity. It also encourage us to exploit opportunities to

extend our approach to the application of mediation to new areas such as query planning, integrity management, and update management. We have good guidelines for this future work since a number of extensions of the generic abductive framework to these type of non monotonic problems have already been proposed.

We have not only succeeded to keep the "declarativeness" of the context knowledge, i.e. its relative independence from the particular task of query mediation, but also to maintain a separation between the mediation itself and the subsequent generation and realization of a query execution plan. Nevertheless, as we see opportunities to exploit context knowledge to optimize and execute queries, we are constantly challenging this frontier.

The procedure we have presented is operational in a complete mediation prototype environment which has been discussed [BFG+97b] and demonstrated [BGa97]. The prototype is used in several application domains such as financial analysis and logistic in liaison with our industry partners.

A A Simple Example

In order to illustrate the use of the abductive procedure independently from explaining its application to mediation, we will use a simple classical example of circuit analysis proposed in [FG85] and solved by the *Residue* procedure.

The example is the one of a circuit with two inputs A and B taking positive integer values, two outputs E and D, E outputs positive integer values and D outputs 1 or 0. The circuit of figure 7. It is composed of two adders and one comparator.

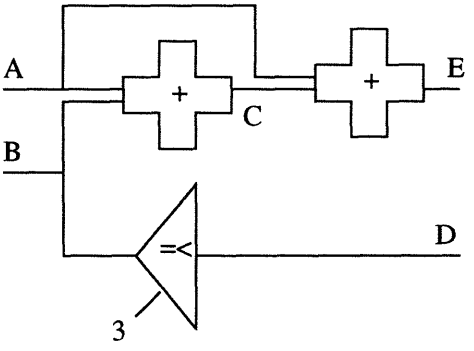

Figure 7: Circuit Analysis.

We can write the following Horn clauses to describe the circuit:

- $a(X_1), b(X_2), X_1 + X_2 = X_3 \rightarrow c(X_3)$;
- $a(X_1), c(X_2), X_1 + X_2 = X_3 \rightarrow e(X_5)$;
- $b(X), X \leq 3 \rightarrow d(1)$;
- $b(X), X > 3 \rightarrow d(0)$.

Additionally, some constraints can be stated:

- We need several constraints stating that each input or output has at most one value: e.g. $a(X_1), a(X_2) \rightarrow X_1 = X_2$ (notice that these constraints are functional dependencies);
- $a(X_1), c(X_2), X_1 + X_2 = X_3 \rightarrow e(X_3)$

- We constrained the input and output to be strictly positive integers. here, without going into the details, we assume a simple solver for integer linear arithmetic (cf. [Wal96] for references).

The initial goal is the negation of the wished output of the circuit, D is 1 and D is 1. We write the goals and the resolvant in their conjunctive counterpart:

$$e(14) \wedge d(1)$$

the resolution with the second rule leads to the resolvant:

$$a(X_1) \wedge c(X_2) \wedge X_1 + X_2 = 14 \wedge d(1)$$

The integer linear arithmetic figures out in the propagation phase that X_1 and X_2 may only have values between zero and fourteen. The next selection, $a(X_1)$ is then posted in the store. $c(X_2)$ is resolved with the first rule (after renaming variables):

$$a(X_3) \wedge b(X_4) \wedge X_3 + X_4 = X_2 \wedge X_1 + X_2 = 14 \wedge d(1)$$

$a(X_3)$ is then posted in the store. The functional dependency constraint applies and figures out that $X_3 = X_1$.

$$b(X_4) \wedge X_1 + X_4 = X_2 \wedge X_1 + X_2 = 14 \wedge d(1)$$

before the resolvant is reduced to $d(1)$, the propagation in the store discovers that $2 * X_1 + X_4 = 14$. Then $d(1)$ is resolved against the first rule for d and after further resolution and propagation of the functional dependency for b, the propagation can discover That X_4 has to be even and is lower or equal to three. Therefore it must be two and X_1 is six. The store once the resolvant is emptied (the result of the procedure) is:

$$[a(6), b(8)]$$

References

[ALUW93] Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Walker. Methods and rules. In *Proceedings of the ACM SIGMOD Conference*, pages 32–41, Washington, DC, May 1993.

[BFG+97a] Stéphane Bressan, Kofi Fynn, Cheng Hian Goh, Marta Jakobisiak, Karim Hussein, Henry Kon, Thomas Lee, Stuart Madnick, Tito Pena, Jessica Qu, Andy Shum, and Michael Siegel. The COntext INterchange mediator prototype. In *Proc. ACM SIGMOD/PODS Joint Conference*, Tuczon, AZ, 1997. To appear.

[BFG+97b] Stéphane Bressan, Kofi Fynn, Cheng Hian Goh, Stuart Madnick, Tito Pena, and Michael Siegel. Overview of prolog implementation of the context interchange mediator. In *Proc. of the Fifth International Conference on the Practical Application of Prolog*, London, UK, April 1997. To appear.

[BGa97] S. Bressan, C. Goh, and al. The context interchange mediator prototype. In *Proc. of ACM SIGMOD conference*, 1997.

[BHP92] M.W. Bright, A.R. Hurson, and S.H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, 25(3):50–60, 1992.

[Bid91] N. Bidoit. Negation in rule-based database language: a survey. *Theoretical Computer Science*, 78:3–83, 1991.

[BJ87] W. Bibel and Ph. Jorrand, editors. *Fundamentals of Artificial Intelligence.* Springer Verlag, 1987.

[BL] S. Bressan and T. Lee. Information brokering on the world wide web. submitted.

[Bry90] F Bry. Intensional updates: abduction via deduction. In *Proceedings of the 7th International Conference on Logic Programming*, 1990.

[Ce95] M. J. Carey and et al. Towards heterogeneous multimedia information systmes: the Garlic approach. In *Proceedings of the Fifth International Workshop on Research Issues in Engineering (RIDE): Distributed Object Management*, 1995.

[CGM90] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, 1990.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer Verlag, 1990.

[Cha70] C.L. Chang. The unit proof and the input proof in theorem proving. *Journal of the ACM*, 4(17), 1970.

[CP86] P. T. Cox and T. Pietrzykowski. Causes for events: their computation and applications. In J. H. Siekmann, editor, *Proceedings CADE-86*, pages 608–621. Springer-Verlag, 1986.

[Dec] H. Decker. An extension of SLD by abduction and integrity constraints for view updating in deductive databases.

[DS92] Marc Denecker and Danny De Schreye. SLDNFA: an abductive procedure for normal abductive programs. In *Proc. Int Conf and Symposium on Logic Programming.* 686–700, 1992.

[ECL96] ECRC. *ECRC parallel constraint logic programming system*, 1996.

[EK89] K. Eshghi and R. Kowalski. Abduction compared with negation as failure. *Proc of 6th Intl: Conf. on Logic Programming*, 1989.

[Esh88] K. Eshghi. Abductive planning with event calculus. In *Proc of the 5th Intl. Conf. on Logic Programming*, 1988.

[Esh93] Kave Eshghi. A tractable set of abduction problems. In *Proc. 13th International Joint Conference on Artificial Intelligence*, pages 3–8, Chambery, France, 1993.

[FG85] J.S. Finger and M.R. Genesereth. Residue: A deductive approach to design synthesis. Technical Report TR-CS-8, Stanford University, 1985.

[FH93] T. Frühwirt and P. Hanschke. Terminological reasoning with constraint handling rules. In *Terminological Reasoning with Constraint Handling Rules*, 1993.

[GBMS96a] C.H. Goh, S. Bressan, S.E. Madnick, and M.D. Siegel. Context Interchange: Representing and Reasoning about Data Semantics in Heterogeneous Systems. Technical Report #3928, Sloan School of Management, MIT, 50 Memorial Drive, Cambridge MA 02139, October 1996.

[GBMS96b] Cheng Hian Goh, Stéphane Bressan, Stuart E. Madnick, and Michael D. Siegel. Context interchange: Representing and reasoning about data semantics in heterogeneous systems. Sloan School Working Paper #3928, Sloan School of Management, MIT, 50 Memorial Drive, Cambridge MA 02139, Oct 1996.

[GM78] H. Gallaire and J. Minker, editors. *Logic and Data Bases.* Plenum Press, 1978.

[GMS94] Cheng Hian Goh, Stuart E. Madnick, and Michael D. Siegel. Context interchange: overcoming the challenges of large-scale interoperable database systems in a dynamic environment. In *Proceedings of the Third International Conference on Information and Knowledge Management*, pages 337–346, Gaithersburg, MD, Nov 29–Dec 1 1994.

[Goh97] Cheng Hian Goh. *Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Systems.* PhD thesis, Sloan School of Management, Massachusetts Institute of Technology, Jan 1997.

[JM96] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *J. of Logic Programming*, 1996.

[KKT93a] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[KKT93b] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6), 1993.

[KLW95a] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, 4, 1995.

[KLW95b] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, 4:741–843, 1995.

[KM90] A.C. Kakas and P. Mancarella. Database updates through abduction. In *Proceedings 16th International Conference on Very Large Databases*, 1990.

[Llo87] J. W. Lloyd. *Foundations of logic programming.* Springer-Verlag, 2nd, extended edition, 1987.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogenous information sources using source descriptions. In *Proc. of the 22nd Conf. on Very Large Databases.*, 1996.

[McC92] Francis G. McCabe. *Logic and Objects.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proc of the 4th Intl. Conf. on Deductive and Object-Oriented Databases*, 1995.

[Sha89] M. Shanahan. Prediction is deduction but explanation is abduction. In *Proc. of the IJCAI-89*, pages 1055–1060, 1989.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1994.

[VB86] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *proc. of the First Intl. Conf. on Expert database Systems*, 1986.

[Wal96] M. Wallace. Practical applications of constraint programming. *Constraints, An International Journal*, 1(1):139–168, 1996.

[WKT95] G. Wetzel, R. Kowalski, and F. Toni. A theorem proving approach to CLP. In *PROC. OF the 11th WORKSHOP ON LOGIC PROGRAMMING*, 1995.

[WKT96] G. Wetzel, R. Kowalski, and F. Toni. Procalog: Programming with constraints and abducibles in logic. JICSLP Poster session, September 1996.