Query Planning for Intelligent Information Integration using Constraint Handling Rules

Liviu Badea, Doina Tilivea
AI Lab, National Institute for Research and Development in Informatics
8-10 Averescu Blvd., Bucharest, Romania
badea@ici.ro

Abstract. Intelligent Information Integration (I^3) aims at combining heterogeneous and distributed information sources by explicitly representing and reasoning about their content, giving the user the illusion of interacting with a single, homogeneous system. To achieve this, an intermediate mediator layer is interposed between the information sources and the end user interface, while a specialised query planner is used to reduce user queries to information source queries. In this paper we use *Constraint Handling Rules (CHR)* for implementing such a query planner.

Conceptually, our approach relies on a generalised abductive reasoning mechanism involving so called *partially open predicates*, which support a seamless combination of backward (goal-directed) and forward reasoning. Goal regression is thus interleaved with forward propagation needed for ensuring consistency. Sub-queries to relations of the same source are aggregated to ensure minimal interactions between the mediator and the sources. We argue that the declarative nature of Constraint Handling Rules as well as the specialised constraint reasoning modules offered by the host CLP environment enable the development of an efficient and flexible query planner, dealing with complex source and domain descriptions.

1 Introduction and motivation

The integration of hybrid modules, components and software systems, possibly developed by different software providers, is a notoriously difficult task, involving various extremely complex technical issues (such as distribution, different programming languages, environments and even operating systems) as well as conceptual problems (such as different data models, semantic mismatches, etc.).

Solving the conceptual problems requires the development of a common, explicit, declarative knowledge model of the systems to be integrated. Such a model should be used (by a so-called *mediator*) not only during the development of the integrated system, but also during runtime, when it can be manipulated by a *query planner* to solve problems that could not have been solved by any of the specific information sources alone and might even not have been foreseeable by the system integrator.

The first step consists in wrapping the information sources (components) to be integrated in an agent-like system. Since integration involves more than interfacing these agents (for example exploiting them in a synergetic manner), we shall develop explicit conceptual models of the components to be integrated in order to allow reasoning with and about them. The *mediator* thus provides uniform and flexible access to the heterogeneous information sources on a conceptual level. It deals with the following aspects: *query planning*, *query execution*, *result integration* and *result presentation*.

The *query planner* of the mediator transforms a user query into a sequence of information-source specific queries that jointly solve the original query. The *query execution* module performs the actual source accesses, while the *result integration* component combines the results obtained from the information sources in a globally consistent result. The *result presentation* module provides the user with a personalised and meaningful answer using a set of filters to remove redundant or trivial information (already known by the user), presenting the result in an intensional form, whenever possible.

In this paper we use *Constraint Handling Rules (CHR)* for implementing the query planner of such an I^3 system. Conceptually, our approach relies on *a generalised abductive reasoning mechanism* involving so called *partially open predicates*, which support a seamless combination of backward (goal-directed) and forward reasoning. Goal regression is thus interleaved with forward propagation needed for ensuring consistency.

Query planning aims at preventing as many potential failures as possible at the (early) planning stage, i.e. before actually accessing the information sources. CHRs enable a natural interleaving of query planning with execution. It would be unrealistic to assume that the query planning stage will be able to construct foolproof plans, i.e. plans that do not fail (especially because of the incomplete knowledge available to the planner). Therefore, we should expect failures and hence should provide mechanisms for *replanning* during execution.

Query planning may also involve optimisations. For example, we may want to select low-cost plans from sets of candidate plans by taking into account cost measures such as access costs to sources, estimated execution time, plan length, etc.

2 Modelling information sources

While each source predicate *s* represents the data contents of some information source, the user might prefer to have a *uniform* interface to all the sources. In fact, she may not even want to be aware of the structure of the information sources. Therefore, the mediator provides a uniform knowledge representation language in terms of so-called "base predicates", which represent the user's perspective on the given domain.

There are two main viewpoints on representing sources and their interactions: *"Global as View"* and *"Local as View"* [L99].

In the *Global As View (GAV)* approach, interactions between sources have to be described explicitly for every combination of sources that interact. For example

$$s_1(X,Y) \wedge s_2(Y,Z) \rightarrow b(X,Y,Z). \quad (1)$$

expresses the fact that a particular combination of s_1 and s_2 has property *b*.

The need to consider all relevant combinations of sources has the obvious drawback that adding a new source is complicated, since it involves considering all potential interactions with the other sources. (There can be an exponential number of such interactions.)

On the other hand, in the *Local As View (LAV)* approach, the interactions between sources are being handled by the query planner at the mediator level. All that needs to be done when adding a new source is to describe all its relevant properties (in the mediator language, i.e. in terms of base predicates).

For example, one would independently describe the sources s_1 and s_2 using the implications $s_1(X,Y) \rightarrow b_1(X,Y)$, $s_2(Y,Z) \rightarrow b_2(Y,Z)$ and delegate the detection of potential interactions between them to the query planner, which would need domain specific descriptions of the base predicates, such as

$$b_1(X,Y) \wedge b_2(Y,Z) \rightarrow b(X,Y,Z). \quad (2)$$

Note that the LAV approach only shifts the description of source interactions from the source level (1) to the level of base predicates (2). This is not just a simple technical trick, since descriptions of the form (2) would be part of the *domain ontology* (which would be developed once and for all and therefore wouldn't change as new sources are added). In the LAV approach, the description of the sources is easier, since the knowledge engineer only has to describe the properties of the newly added source in terms of base predicates, without having to worry about potential source interactions. This latter task is delegated to the query planner.

The query planner presented in this paper can deal with both GAV and LAV approaches. However, we will encourage the use of LAV, since it spares the knowledge engineer the effort of explicitly determining and representing the source interactions.

The choice of the language of "base predicates" used to describe a particular domain should be guided by its simplicity and intuitiveness to the end user. It should also be complete w.r.t. the applications to be integrated (i.e. it should be able to express all *relevant* properties in the domain in question).

We start by giving the intuition behind our modelling of source descriptions using Constraint Handling Rules (CHR) and Constraint Logic Programming (CLP).

Constraint Handling Rules [F95] represent a flexible approach to developing user-defined constraint solvers in a declarative language. As opposed to typical constraint solvers, which are black boxes, CHRs represent a 'no-box' approach to CLP. CHRs can be either *simplification* or *propagation* rules.

A *simplification* rule Head <=> Guard | Body replaces the head constraints by the body provided the guard is true (the Head may contain multiple CHR constraint atoms). *Propagation rules* Head ==> Guard | Body add the body constraints to the constraint store without deleting the head constraints (provided the guard is true). A third, hybrid type of rules, *simpagation rules* $\text{Head}_1 \setminus \text{Head}_2$ <=> Guard | Body replace Head_2 by Body (while preserving Head_1) if Guard is true. (Guards are optional in all types of rules.)

Now, a Local As View source description of the form

$$s(X) \rightarrow b_1(X), b_2(X), \ldots \quad (3)$$

will be implemented using CHR *propagation rules*

$$s(X) ==> b_1(X), b_2(X), \ldots \quad (4)$$

which ensure that every time a query is reduced to a source predicate *s(X),* the source's properties are automatically propagated (thereby enabling the discovery of potential inconsistencies). Propagation rules however, capture only one direction of the implication (3). Since they are unable to deal with the contrapositive of (3), they are inappropriate for regressing a goal in terms of base predicates to a subgoal in terms of the source predicate. This has to be explicitly implemented using *goal regression rules* of the form

$$b_i(X) \text{ :- } s(X). \quad (5)$$

Unfortunately, this simple approach may loop.

For example, a query ?- $b_1(X)$ will be reduced to ?- $s(X)$ using the goal reduction rule (5) and subsequently "saturated" to ?- $s(X), b_1(X), b_2(X), \ldots$ using the propagation rule (4). But now, $b_1(X)$ will again be reduced to $s(X)$ (using (5)) and propagated to $b_1(X), b_2(X), \ldots$ (using (4)), thus leading to a loop.

To solve this problem, we have to distinguish between facts that represent goals (denoted as *p*) and facts, which are asserted to hold, represented as $\mathcal{H}p$. (In the implementation, *p* is represented as *holds(p)*, while goals *p* are represented explicitly as *goal(p)*.)

Propagation rules involve $\mathcal{H}$:

$$\mathcal{H}s(X) ==> \mathcal{H}b_1(X),\ \mathcal{H}b_2(X),\ \ldots \qquad (6)$$

while goal reduction rules take the form:

$$b_i(X) :\!-\ s(X). \qquad (7)$$

The following schema (w.r.t. *p*) takes care of the consistency between goals and facts that simply hold (since goals will be achieved, they also have to hold):

$$p ==> \mathcal{H}p \qquad (8)$$

Note that rule (8) should have a higher priority than (7) ($\mathcal{H}p$ should be propagated before the goal is deleted in (7)). Now a query ?- $b_i(X)$ will propagate $\mathcal{H}b_i(X)$: ?-$b_i(X)$, $\mathcal{H}b_i(X)$, then will be rewritten using (7) to ?-$s(X), \mathcal{H}b_i(X)$. $s(X)$ will now propagate $\mathcal{H}s(X)$ using (8) and then $\mathcal{H}b_1(X)$, $\mathcal{H}b_2(X), \ldots$ using (6): ?-$s(X), \mathcal{H}s(X), \mathcal{H}b_1(X), \mathcal{H}b_2(X), \ldots$.

And finally, $s(X)$ will trigger the actual source query: $s(X)$:- *call_source(s(X)).* [1]

Besides source descriptions (3), we also have to be able to represent logical constraints between base predicates (for example concept hierarchies, integrity constraints, etc.). The same implementation strategy can be used in this case. For example, an implication of the form

$$b_1(X,Y) \wedge b_2(Y,Z) \rightarrow b(X,Z)$$

can be represented using a CHR propagation rule

$$\mathcal{H}b_1(X,Y),\ \mathcal{H}b_2(Y,Z) ==> \mathcal{H}b(X,Z)$$

and a goal reduction rule $b(X,Z)$:- $b_1(X,Y)$, $b_2(Y,Z)$.

Example. Assume that in a software company we want to allocate qualified personnel to a new project while not exceeding a given budget. Assume that the company's information system contains separate databases for each department (in our case ***administrative*** and ***development***, which are regarded as source predicates), as well as a database of ***student***s (part-time employees) who could be employed in the project, in case the overall project budget is exceeded by using full-time personnel (students are assumed to have a lower salary).

The end user of the integrated system (for example the company's manager) would use ***base predicates*** such as: *employee(Employee),qualification(Employee,Qualification), salary(Employee, Salary).* The following description rules are used to represent ***domain-specific knowledge*** (descriptions for base predicates):

qualification(Empl, 'C') → *programmer(Empl)* (e1)

qualification(Empl, 'Prolog') → *programmer(Empl)* (e2)

programmer(Empl1), non_programmer(Empl2) → *Empl1* ≠ *Empl2* (e3)

The following *description rules* are used to characterize the ***source predicates***:

administrative*(Empl, Qual, Sal)* →
employee(Empl), qualification(Empl, Qual), salary(Empl, Sal), non_programmer(Empl). (e4)

development*(Empl, Qual, Sal)* →
employee(Empl), qualification(Empl, Qual), salary(Empl, Sal), Sal > 2000, programmer(Empl). (e5)

student*(Empl, Qual)* →
employee(Empl), qualification(Empl, Qual), salary(Empl, Sal), Sal = 1000, programmer(Empl). (e6)

The above description rules assert that:

- people with qualifications in *C* and *Prolog* are programmers, while programmers and non-programmers are disjoint concepts.
- administrative employees are non-programmers
- employees from the development department are programmers (with a salary higher than 2000)
- students are also programmers and are assumed to have a fixed salary of 1000.

The user might want to allocate two employees *Empl1* and *Empl2* with qualifications *'C'* and *'Prolog'* such that their summed up salaries do not exceed 3500:

query(Empl1,Empl2) :- *employee(Empl1), qualification(Empl1,'C'), salary(Empl1,Sal1), employee(Empl2), qualification(Empl2,'Prolog'), salary(Empl2,Sal2), Sal1+Sal2<3500.*

The above query would first trigger the goal reduction rules for *employee*, regressing the goal *employee(Empl1)* to ***administrative****(Empl1, 'C', Sal1)*. The latter would then propagate with (e4) *non_programmer(Empl1)* (among others). The next goal, *qualification(Empl1, 'C')*, will then propagate with (e1) *programmer(Empl1)*, which will produce an inconsistency with the previously propagated *non_programmer(Empl1)* (using (e3)). This will trigger backtracking to the goal *employee(Empl1)*, which will now be reduced to the source predicate ***development****(Empl1, 'C', Sal1)* (which propagates *programmer(Empl1)*, that is consistent with the next goal *qualification(Empl1, 'C')).* ***development****(Empl1, 'C', Sal1)* also propagates (with (e5)) *Sal1 > 2000*.

A similar chain of reasoning steps will be performed for the sub-goals involving *Empl2*, leading to the source access ***development****(Empl2, 'Prolog', Sal2)* and the constraint *Sal2 > 2000*. Unfortunately, the constraint *Sal1+Sal2<3500* is now violated by *Sal1>2000, Sal2 > 2000*, leading to a backtracking step in which the subgoal *employee(Empl2)* is solved by accessing the source ***student****(Empl2, 'Prolog')*, whose salary *Sal2 = 1000* is consistent with the constraint *Sal1+Sal2 < 3500*. All the above reasoning steps belong to the query *planning* phase.

The resulting plan (consisting of source accesses and constraints)

development*(Empl1,'C',Sal1),* ***student****(Empl2,'Prolog'), Sal2=1000, Sal1+Sal2<3500*

[1] In fact, this step is more complicated, since the various queries that can be dealt with by a single source have to be packaged in order to minimize the transfer of data from the sources to the mediator (this is done by taking the source capabilities into account. See Section 3.4 below.)

will be split into queries for the separate sources (***development*** and ***student*** being assumed to be tables in different databases). While doing this, the constraints treatable internally by the various sources will be attached to the corresponding source queries. In the above example, the constraint *Sal1+Sal2<3500* involves variables of two separate sources and will not be directly treatable by any of the sources. However, since *Sal2=1000*, it will be simplified to *Sal1 < 2500*, which is treatable internally by the first source. Therefore, the execution of the plan above will consist in querying the first source with ***development****(Empl1, 'C', Sal1), Sal1 < 2500* and the second with ***student****(Empl2, 'Prolog')*.

Note that querying the first source with the whole subplan ***development****(Empl1, 'C', Sal1), Sal1 < 2500* (instead of eagerly querying it with ***development****(Empl1, 'C', Sal1)*, without bothering to construct plans), will typically transport much less tuples from the sources to the mediator (because *Sal1 < 2500* acts as a filter!).

Planning and execution will be interleaved in a seamless manner. For example, the second sub-query ***student****(Empl2, 'Prolog')* might fail (at query time we might discover that there are no students in the database qualified in *Prolog*). One would then have to backtrack to the first subgoal *employee(Empl1)* and obtain it from ***student****(Empl1, 'C')*, presumable allowing *employee(Empl2)* to be solved by using a more expensive permanent employee from ***development****(Empl2, 'Prolog', Sal2)*.

Note that if we knew (in the source description (e6)) that students do not know *'Prolog'*, we could have obtained the correct solution without accessing the source ***student***. This remark shows that source descriptions are very useful for pruning the search space at planning time (before actually querying the sources).

3 Query planning with Constraint Handling Rules

In the following, we concentrate on a detailed description of the representation language for source descriptions and domain models, as well as on their implementation using Constraint Handling Rules. We also show that the notion of *open predicates* (which generalise abductive predicates from Abductive Logic Programming) can be used to describe our approach at a conceptual level.

3.1 Base predicates, source predicates and description rules

The content of the various information sources is represented by so-called *source predicates*, which will be described at a declarative level in terms of the so-called *"base predicates"*. More precisely, we distinguish between *content* predicates and *constraint* predicates.

Content predicates (denoted in the following with *p*, *q*, possibly with subscripts) are predicates which directly or indirectly represent the *content* of information sources. They can be either *source* predicates or *base* predicates.

Source predicates (denoted with *s*) directly represent the content of (part of) an information source (for example, a table in a relational database, or the services provided by the interface of a procedural application). Their definitions (bodies) are not explicit at the mediator level, but can be accessed by querying their associated information source.

Base predicates (denoted with *b*) are user-defined predicates for describing the domain, as well as the information content of the sources.

As opposed to content predicates, *constraint predicates* are used to express specific constraints on the content predicate descriptions. (They do not represent the content of the information sources.) For example, a source *s* containing information about underpaid employees (with a salary below 1000), would be described as:

$$s(Name, Salary) \rightarrow employee(Name) \wedge salary(Name, Salary) \wedge Salary < 1000.$$

In the above, *s* is a source predicate, *employee* and *salary* are base predicates, while *'<'* is a constraint predicate.

Constraint predicates can be either

- *internal* (treatable internally by the query engine of the source), or
- *external* (constraints that can only be verified at the mediator level, for example by the built-in constraint solvers of the host CLP environment).
- Constraints can be processed either by:
- *verification* (execution of the Prolog code denoting their semantics), or by
- *propagation* (of new constraints based on old ones, without actually executing their "bodies").

Constraints treatable *internally* by the sources can be *verified* at the source level (by the query engines of the sources), but they are also *propagated* at the mediator (CLP) level. Constraints treatable only *externally* need to be verified and propagated at the mediator level.

A *complete* description of the source predicates in terms of base predicates is neither possible nor useful, since there are way too many details of the functioning of the sources that are either unknown to the user, or irrelevant from the point of view of the application. Thus, instead of *complete* (iff) descriptions, we shall only specify approximate (necessary) definitions of the source predicates in terms of base predicates (thus, only the relevant features of the sources will be encoded). For example, when describing a source of ***supplier***s using the base predicate *offer*, we may ignore some details (such as quantities) which are uninteresting from the point of view of the user:

$$\textbf{supplier}(Id,Item,Quantity,Price) \rightarrow offer(Item,Price).$$

In the following, we use a uniform notation for the **domain and source description rules**:

$$\forall \overline{X}.\ p_1(\overline{X}_1) \wedge \ldots \rightarrow \exists \overline{Z}.\, b_1(\overline{Y}_1) \wedge \ldots \wedge s_i(\overline{Y}_i) \wedge \ldots \wedge c_j(\overline{Y}_j) \wedge \ldots \quad (9)$$

where $\overline{X}=\bigcup_i \overline{X}_i$, $\overline{Y}=\bigcup_j \overline{Y}_j$ $\overline{Z}=\overline{Y}-\overline{X}$ (with variable tuples viewed as sets).

Description rules are necessary definitions of (combinations of) source or base predicates in terms of base predicates and constraints. (*Source descriptions* are special cases of description rules. *Integrity constraints* are description rules with *'fail'* in the consequent.[2])

The antecedent of such a description rule may contain only content predicates (both source and base predicates), but no constraints (any relevant constraints are to be placed in the consequent).[3]

Typically, sources contain many details that are irrelevant from the user's point of view (their content being too fine grained). Source description rules abstract away these irrelevant details and are therefore typically *coarser grained* than the information content of the sources. However, there are situations in which the source description is *finer grained* than the explicit source content, typically due to (meta-level) knowledge about the content of the source that is not explicitly contained in the source.

Examples. A source of cheap cars may not contain the actual prices of the cars, but we may possess the meta-level knowledge that these prices are lower than 10000:

$s(Model) \rightarrow \exists P.\ car(Model) \wedge price(Model, Price) \wedge P<10000.$

Another source may contain information about grandparents without making the (intermediate) parents explicit:

$s(X,Y) \rightarrow \exists Z.\ parent(X,Z) \wedge parent(Z,Y).$

Such conceptual descriptions that are finer grained than the source content involve existential variables in the consequent and are, in a sense to be made precise shortly, "nondecomposable". Existential variables can appear in rule consequents also due to *conversions* from a format used in sources to the common format used at the mediator level.

Example. Prices may be kept in \$ at the source level, but in thousands of \$ at the mediator level:

$s(Model, Price) \rightarrow \exists P.\ car(Model) \wedge P = Price/1000 \wedge price(Model, P).$

Occurrences of source predicates in the consequent do not represent updates to the corresponding source. Rather, they are typically used for *conversions* (where the source predicate plays the role of a conversion table or conversion function), or more generally for expressing *constraints* that are encoded in the given source predicate. These need to be *verified* rather than propagated.

Example. Prices of cars may be stored in a different currency, for which a source *s_exchange* (functioning as a conversion table) is available:

$s(Model, Price) \rightarrow \exists P.\ car(Model) \wedge s\_exchange(Price, P) \wedge price(Model, P).$

As opposed to *base* predicates occurring in the consequent (for which the description rule functions as a sufficient definition[4]), the description rule is *not* a (sufficient)definition for the *source* predicates occurring in its consequent. These are only regarded as *constraints*.

Description rules provide a convenient way of *describing* sources as well as the domain knowledge (involving base predicates). For implementing the associated *reasoning* mechanisms of the query planner, we employ Constraint Handling Rules (CHR) [F95]. CHRs provide a flexible and declarative means of writing user-defined constraint solvers. Their tight embedding in Constraint Logic Programming (CLP) systems (like ECLiPSe or SICStus) enables the use of their efficient built-in constraint solvers for constraint propagation during query planning.

A convenient way of modelling the reasoning of the query planner consists in introducing the notion of *open predicates*. The query planner essentially interleaves two reasoning phases: *goal regression* and *forward propagation* of properties ("saturation"). While goal regression uses the current "closure" (body) of a given base predicate, forward propagation may produce *new instances* of the base predicate (which have to hold for the partial query plan to be consistent). Thus, we need to be able not only to refer to the closure of a given predicate, but also to allow it to be "open".

3.2 Open predicates

In Logic Programming, normal predicates are *closed*: their definition is assumed to be complete (Clark completion). On the other hand, abducibles in Abductive Logic Programming (ALP) [KMM98, KKT98] are *completely open*, i.e. they can have any extension consistent with the integrity constraints. For dealing in a formal manner with forward propagation rules in an abductive framework, we need to allow a generalisation of abducibles, namely (partially) *open predicates*.

Unlike normal abducibles, open predicates can have definitions $p \leftarrow Body$, but these are not considered to be complete, since during the problem solving process we can add (by forward propagation) new abductive instances of p to these definitions. The definition of an open predicate therefore only partially constrains its extension.

In our CHR embedding of the abductive procedure we use two types of constraints, p and $\mathcal{H}p$, for each open predicate p. While $\mathcal{H}p$ represent facts explicitly propagated (abduced), p refers to the *current closure* of the predicate p

[2] Actually, an integrity constraint of the form $\leftarrow p(X),q(X)$ will be written as $p(X1),q(X2)$==>$X1 \neq X2$ rather than $p(X),q(X)$ ==> *fail*.

[3] The difference between $s(X),c(X) \rightarrow b(X)$ and $s(X) \rightarrow b(X),c(X)$ is that the first asserts the property b only for the tuples of s verifying c, while the second asserts both b and c for *all* tuples of s.

[4] i.e. a definition providing *sufficient* conditions for an object to verify the source predicate.

(i.e. the explicit definition of p *together* with the explicitly abduced literals $\mathcal{H}p$). Thus, informally we have

$p = def(p) \vee \mathcal{H}p$.

While propagating $\mathcal{H}p$ amounts to simply assuming p to hold (abduction), propagating p amounts to trying to prove p either by using its definition *def(p)*, or by reusing an already abduced fact $\mathcal{H}p$. This distinction ensures that our CHR embedding conforms to the usual '*propertyhood view*' on integrity constraints:

Definition $M(\Delta)$ is a *generalised stable model* of the abductive logic program $\langle P,A,I\rangle$ for the abductive explanation $\Delta \subseteq A$ iff (1) $M(\Delta)$ is a stable model of $P\cup\Delta$, and (2) $M(\Delta) \models I$.

The distinction between propagating $\mathcal{H}p$ and p respectively can be explained best using an example. Earning a certain income should propagate the *obligation* of having one's taxes paid (represented by the closure *taxes_paid*). However, propagating *taxes_paid* does not by itself ensure that taxes are indeed paid: it is just a *goal* to be solved -- an *obligation*. Only propagating $\mathcal{H}$*taxes_paid* ensures that taxes have actually been paid.

The use of *open predicates* allows mixing *forward propagation* of abduced predicates $\mathcal{H}p$ with *backward reasoning* using the closures p. Forward propagation can be implemented using CHR propagation rules, while backward reasoning involves unfolding predicates with their definitions. The definition *def(p, Body)* of a predicate p is obtained by Clark completion of its 'if' definitions. For each such predicate we will have an *unfolding rule* (a CHR simplification rule) *p <=> def(p, Body)* | *Body*, but also a CHR simpagation rule[5] for matching a goal p with an existing abduced fact $\mathcal{H}p$:

$\mathcal{H}p(X_1) \setminus p(X_2)$ <=> $X_1=X_2$; $X_1 \neq X_2, p(X_2)$.

This rule should have a higher priority than the unfolding rule in order to avoid reachieving an already achieved goal. Note that, for completeness, we are leaving open the possibility of achieving $p(X_2)$ using its definition or reusing other abduced facts.

Our treatment of open predicates $p = def(p) \vee \mathcal{H}p$ is slightly different from the usual method [KKT98] of dealing with (partially) open predicates p, where a new predicate name p' (similar to our $\mathcal{H}p$) is introduced and the clause $p \leftarrow p'$ is added to the definition of p:

$$\{p \leftarrow Def,\ \ p \leftarrow p'\}. \qquad (*)$$

The difference is that whenever we refer to p we are implicitly trying to prove p, either by using its definition *def(p)* or by reusing an already abduced fact $\mathcal{H}p$, *but without allowing such an* $\mathcal{H}p$ *to be abduced in order to prove p* (whereas in (*) treating $p \leftarrow p'$ as a *program clause*[6] would allow p' to be abduced when trying to prove p). This is crucial for ensuring a correct distinction[7] between goals p and abducibles $\mathcal{H}p$ mentioned above (otherwise we would treat the propagation of goals and obligations incorrectly). Without making this distinction, *we wouldn't even be able to refer to the current closure of p.*

3.3 Implementing description rules in CHR

A description rule of the form (9) will be implemented in CHR using

- *goal reduction rules*: for reducing queries given in terms of base predicates to queries in terms of source predicates
- *propagation rules*: for saturating (intermediate) descriptions in order to allow the discovery of potential inconsistencies.

An implication of the form $p \rightarrow q_1 \wedge q_2 \wedge \ldots$ is decomposable into a conjunction of implications $(p \rightarrow q_1) \wedge (p \rightarrow q_2) \wedge \ldots$ whenever the consequent includes no existential variables. In the general case, however, we need to Skolemize the description rules (9) before decomposing them (decomposition being necessary for generating goal reduction rules):

$$\begin{aligned}&\forall \overline{X}.p_1(\overline{X}_1) \wedge \ldots \rightarrow \\ &\ldots \wedge sk(k, Z_k, \overline{X}) \wedge \ldots b_1(\overline{Y}_1) \wedge \ldots s_i(\overline{Y}_i) \wedge \ldots c_j(\overline{Y}_j) \wedge \ldots\end{aligned} \qquad (10)$$

where $sk(k, Z_k, \overline{X})$ denotes the fact that the existential variable Z_k depends on the variable tuple $\overline{X}$. Essentially, this is a way of writing the Skolem term $Z_k = f_k(\overline{X})$ using a predicate *sk* instead of a function f_k. This simplifies our dealing with such Skolems in our CHR implementation, in which *sk* is a CHR constraint subject to the following propagation rules:

sk(K,Z1,X), sk(K,Z2,X) ==> Z1=Z2.
sk(K,Z,X1), sk(K,Z,X2) ==> X1=X2.

These rules ensure that *X1=X2* iff $f_k(X1)=f_k(X2)$.

Goal reduction rules

Now, (10) is decomposable into:

$$\forall \overline{X}.\ \ p_1(\overline{X}_1) \wedge \ldots \ \ \rightarrow\ \ \ldots \wedge sk(k, Z_k, \overline{X}) \wedge \ldots\ \wedge b_l(\overline{Y}_l). \qquad (11)$$

for every base predicate b_l occurring in the consequent of (10) (in the consequent of (11) we only keep the Skolems for which $Z_k \in Y_l$).

(11) can easily be viewed as a goal reduction rule for b_l:

$$b_l(\overline{Y}_l)\ \ :-\ \ \ldots, sk(k, Z_k, \overline{X}), \ldots,\ p_1(\overline{X}_1), \ldots \qquad (12)$$

[5] The rule below is more complicated in practice, due to implementation details. The simpagation rule only considers the *first* $\mathcal{H}p$ matching p. In the implementation, we use a combination of propagation and simplification rules to achieve the desired effect.

[6] In fact, $p \leftarrow p'$ should be treated as an integrity constraint and not as a program clause.

[7] This distinction is essential only for *partially* open predicates and not for completely open predicates (abducibles).

Since (10) is *not* a sufficient definition for the source predicates s_i appearing in its consequent (s_i playing here the role of constraints), we do not write goal reduction rules of the form (11) or (12) for s_i. Neither do we write such rules for the constraints c_j.

Propagation rules

A set of subgoals in terms of source predicates (induced by backward chaining from the user query using the goal reduction rules) may not be necessarily consistent. Applying (10) as forward propagation rules ensures the "saturation" of the (partial) query plan and enables detecting potential conflicts *before* actually accessing the sources.

As base predicates p are subject not just to normal backward reasoning, but also to forward propagation, they will be treated as open predicates. The forward propagation rules will thus involve their "open" part $\mathcal{H}p$:

$$\mathcal{H}p_1(\overline{X}_1) \wedge \ldots \rightarrow \\ \ldots, sk(k, Z_k, \overline{X}), \ldots, \mathcal{H}b_1(\overline{Y}_1), \ldots, s_i(\overline{Y}_i), \ldots, \mathcal{C}c_j(\overline{Y}_j), \ldots \quad (13)$$

Note that source predicate occurrences $s_i(\overline{Y}_i)$ in the consequent refer to the "closure" of s_i (rather than their open parts $\mathcal{H}s_i$), since sources in the consequent are used as constraints to be *verified*, rather than propagated.

We have already mentioned the fact that we have to distinguish between goals involving a given predicate p (which will be treated by the normal Prolog backward chaining mechanism) and the instances $\mathcal{H}p$ of p which trigger forward propagation rules. Operationally speaking, while goals p are "consumed" during goal regression, the fact that p should persist holds (even after the goal has been achieved) to enable the activation of the forward propagation rules of p. Goals p will therefore have to propagate

$\mathcal{H}p$: p ==> $\mathcal{H}p$ (14)

before applying the goal reduction rules for p:

p :− body(p).

A predicate p for which $\mathcal{H}p$ is propagated only by rule (14) is *closed* (*p = body(p)*) since all propagated $\mathcal{H}p$ instances will verify *body(p)*. Propagating $\mathcal{H}p$ instances in the consequents of other propagation rules makes p an open predicate.

Delaying source predicate calls during query planning

Instead of directly executing source predicate calls (by actually accessing the sources), sub-goals $s_i(\overline{Y}_i)$ are delayed to allow query planning. Delaying the source predicate calls enables their aggregation into more specific sub-queries, thereby transporting less tuples at the mediator level.

The goal reduction rules for source predicates s post constraints of the form $\mathcal{C}s$, which denote the delayed source call: *s(Y) :−* $\mathcal{C}$*s(Y).*

A *query plan* consists of a number of such source predicate calls $\mathcal{C}s_i(\overline{Y}_i)$ as well as additional constraints $\mathcal{C}c_j(\overline{Y}_j)$ (propagated by forward rules (13)). Note that both types of constraints, $\mathcal{H}s$ and $\mathcal{C}s$, are needed, the latter being "consumed" (deleted) after query execution (so that the source will not be queried again with *s(Y)*). On the other hand, $\mathcal{H}s$ should persist even after executing the source access, because of potential interactions with constraints that may be propagated later on.

3.4 Source capabilities and query splitting

Information sources are viewed as collections of source predicates that can be accessed via a specialized query interface. (Such information sources can be for instance databases, functional applications, etc.)

The query planner reduces a query formulated in terms of base predicates to a query in terms of source predicates. However, such a query can involve source predicates from *several* information sources, so that the query planner also has to split such a global query into sub-queries that can be treated by the separate information sources. Since each information source may have its own query processor, we need to explicitly represent the capabilities of this query processor. For example, a specific database interface may not be able to deal with arbitrary joins, may allow only certain types of selections and may also require certain parameters to be inputs (i.e. known at query time). Dataflow issues such as input-output parameters are especially important in the case of procedural applications.

Source capabilities

We shall describe the capabilities of a given information source using the following items

- the *name* of the source Src
- the *source predicates* that can be accessed from this source (for a relational database, these source predicates are relational tables, for a procedural application they represent services of the application interface)
- the *equations* that can be treated internally by the query processor of the source (in the case of a database, equation $X_i = X_j$ involving attributes of the same table represent selections, while $X_i = X_j$ for attributes of different tables represent joins; for applications, such equations represent possible dataflow patterns).
- the *constraints* that can be treated internally by the query processor (used as selections).

More precisely, we shall use the following internal representation of source capabilities:

source_capabilities(*Src,*
Preds = [*p1(+X1,Y1,-Z1), p2(+X2,-Y2)*, ...],
Eqs = [*X1=Y2, ?=X1, ?=?, X1=type, ?=type*, ...],
Constrs = [*X1>Y1, X>type, ? > type, ? > ?*, ...]).

Note that all variable occurrences in the source predicates are distinct. They are also annotated with input(+) or output(-) markers, according to the following convention:

- input parameters have to be ground at query time
- output parameters have to be completely uninstantiated at query time (they have to be unbound Prolog variables). After the query, such output parameters are guaranteed to be bound to a ground term.
- parameters that are not labelled with + or - are not restricted in any way by the input-output specification. They are also guaranteed to be bound to a ground term after the query.

The input-output specifications determine the dataflow in the query. Such specifications are especially useful in the case of procedural applications. For databases, where any query pattern might be allowed, we may simply ignore writing any input-output specifications. Thus we delegate the query optimization to the database query processor (which might reorder the various relational operations in the query).

For describing which joins and selections are treatable by a given database, we shall explicitly represent these as *equations* in the *source_capabilities* template. Such equations might also be used in the case of applications for describing the dataflow (which outputs can be linked to which inputs in a call). If all joins and selections are allowed, we shall use the notation *?=?*. Similarly, if all equations involving *X* are allowed, we shall write *X=?* in the template.

Queries with an explicit input argument (given at planning time!) are described by an equation of the form *X=type* (*X* can also be *?*), where *type* is a unary Prolog predicate that verifies the type of the corresponding argument.

The *source_capabilities* template also describes which *constraints* in a query can be treated internally by the query processor of the source. For databases, such constraints may be arithmetic comparisons, for example. In such constraint expressions, we can also use *?* to denote any argument and *type* to denote a constant of the given type.

Query splitting

Our query splitting algorithm (also implemented in CHR) incrementally selects the source predicate constraints $\mathcal{C}p$ (from the global query) that can be dealt with by a given source. In doing this, it has to separate the equations (and constraints) that can be treated *internally* by the source's query processor from the ones that need to be treated *externally* (at the level of the mediator). In order to minimize the answer sets of queries (which need to be transferred to the mediator level), the source queries should be made as *specific* as possible. This can be achieved by delegating to the source's query processor as many equations and constraints as possible. (Only the ones that are not treatable internally, will be dealt with externally.)

More precisely, let $\mathcal{C}$ be the set of constraints (including equations). Attaching to the query to source *i* just the constraints from $\mathcal{C}$ that can be internally treated by source *i* (i.e. $int_i(\mathcal{C})$), does not produce maximally specific queries. For example, if the source capabilities mention the equations *X1=X2, X1=X3*, for a query involving *X1=X2, X2=X3* the equation *X1=X2* will be treated internally, while *X2=X3* will be dealt with externally (since it doesn't explicitly occur in the source capabilities). To achieve maximum specific queries, we need to propagate the constraints from $\mathcal{C}$ before splitting them into internal and external ones. In our example, propagating the constraints from the query *X1=X2, X2=X3* produces *X1=X3*, which can be dealt with internally by the source (since *X2=X3* occurs in the source capabilities).

More generally, we attach to each source *i* the constraints $int_i(Prop(\mathcal{C}))$, where $Prop(\mathcal{C})$ is the set of constraints obtained as a result of propagation, and int_i selects the constraints internally treatable by source *i*.[8] Although, theoretically, equations are special types of constraints, we distinguish between equations and constraints because constraint propagation is in general incomplete, while we want a complete treatment of equations at least (in the example above, the propagated equation *X1=X3* can *replace* *X2=X3*).

Example Consider the sources *s1* and *s2* with the following capabilities:

source_capabilities(*s1,*
  [*p1(+T1,X1,-Y1), q1(+U1,+V1,-W1)*],
  [*X1=U1, T1=number*], [*U1>W1, ?>number*]).
source_capabilities(*s2*, [*p2(+T2,X2,-Y2), q2(+U2,V2)*],
  [*Y2=U2, ?=number*], [*?>?*]).

A query of the form (generated by the query planner from the user query posed in terms of base predicates, where *p1, q1, p2, q2* are source predicates):

?- p1(15,A,B), q1(A,B,C), A>C, A>2,
p2(C,B,D), q2(D,E), C>E, A>E.

will be split into the following source-specific queries followed by certain external equations and constraints:[9]

?-*source_call*(*s1*, [*p1(15,X1,-Y1), X1=U1,*
  q1(+U1,+V1,-W1), U1>W1, X1>2]),
Y1=V1, W1=T2, Y1=X2,
source_call(*s2*, [*p2(+T2,X2,-Y2), Y2=U2,*
  q2(+U2,V2), T2>V2]),
X1>V2.

[8] A more sophisticated implementation avoids propagating internal constraints among themselves, since this would only produce *redundant* constraints (which would not additionally filter the answer sets). More precisely, we would attach to each source query the constraints $int_i(Prop(int_i(\mathcal{C}), \mathcal{C}-int_i(\mathcal{C}))$, where $Prop(\mathcal{C}_1,\mathcal{C}_2)$ denotes the set of constraints obtained by propagating the constraints $\mathcal{C}_1$ with the constraints $\mathcal{C}_2$.

[9] Which cannot be treated internally by the sources, since they do not conform to the source capabilities.

4 Concluding remarks and related work

There is a large number of information integration systems, making an exhaustive comparison impossible. However, there are only two main categories of approaches to information integration. In the *classical* approach (*multi-databases* and *federated databases*), fixed global schemas are developed and SQL-like languages are used for queries. Such systems are appropriate only if the schemas and the information users do not change frequently.

For dynamically evolving schemas, so called *intelligent information integration* systems have been developed. These approaches rely on a *mediator* architecture that functions as a middle layer between information sources (information producers) and users (information consumers). Queries posed in a user language are automatically reformulated in terms of the sources, thereby coping with dynamically evolving sources. More *procedural* approaches, like TSIMMIS [GPQ97] and MedLan [AART97], use explicit query reformulation rules (predefined query templates).[10] Our approach is closer to the more declarative systems like SIMS [AKH96], Information Manifold [LRO96], Infomaster [DG97], MOMIS [BCVB99].

Our system is similar in spirit with the Information Manifold (IM). However, there are certain differences: while we are using a general Constraint Handling module, the IM uses a special purpose query planning algorithm. Since CHRs offer a convenient trade-off between expressiveness and computational effectiveness, we argue that this approach allows the easy development of much more *flexible* intelligent information integration frameworks (the descriptions are declarative, while also allowing efficient reasoning about them).

Also, the query planning algorithm used in the IM cannot be easily extended to deal with Global as View descriptions (in which interactions between sources are explicit)[11]. Queries involving source descriptions with existential variables are also not treated completely (see the remark in Section 5 of [DG97b]).

Technically, our use of an optimized constraint propagation mechanism (like the one provided by CHR) may help discover inconsistencies earlier than in IM (which checks for consistency all possible combinations of sources from the buckets)[12].

On the other hand, IM allows using a description logic (DL) in source descriptions. Such DL descriptions can also be reformulated using our description rules. However, we cannot guarantee *complete* reasoning with such descriptions (on the other hand, *full* CARIN is highly intractable in the worst case). IM also doesn't allow reasoning with integrity constraints. In order to guarantee polynomial complexity of the algorithms for dealing with source capabilities, IM uses a more limited source capabilities language than ours (however, the cases of intractability may not appear in real-world cases).

Our approach is also similar to Infomaster (although few details are available in the published papers [DG97,DG97b]), the main difference being that we are using CHRs for reasoning (query planning), while Infomaster uses a model elimination theorem prover. Due to its use of *"safe" iff definitions* for representing sources (and domain knowledge) Infomaster employs *gensym predicates* for emulating unidirectional (necessary) source definitions (which represent the typical case, since only very rarely do we have *complete* source descriptions). On the other hand, iff definitions allow reasoning about *complete* sources, which is useful in certain cases.

After finishing the paper, our attention ws drawn to the work of [BG97] on the COIN project. COIN also uses a CLP framework (Eclipse) for abductive reasoning and CHRs for implementing integrity constraints. However, integrity constraints can be imposed in COIN *only on source predicates*. Therefore, the domain knowledge (expressing constraints between what we call base predicates) reduces to Prolog reduction rules, which are used *only backwards* (during goal regression). The lack of forward propagation rules involving base predicates (and not just sources) makes the discovery of potential interactions between base predicates impossible (because there is no "saturation" phase for base predicates). For example, for the source descriptions $b1 \leftarrow s1$, $b2 \leftarrow s2$, and the domain axioms $b3 \leftarrow b2$, $\leftarrow b1, b3$ (the last being an integrity constraint involving base predicates), the query $:- b1, b2$ would be reduced to $:- s1, s2$ thereby missing the discovery of the inconsistency between *b1* and *b3* (the latter should have been propagated forward by *b2*).[13] Our use of *partially* open predicates solves this problem. Another difference between our system and COIN is that the latter doesn't have a full-fledged query planner (for example, COIN doesn't aggregate sub-queries to relations of the same source, thereby facing potential problems with transporting large/huge numbers of tuples from the source to the mediator level).

The system presented in this paper (implemented in ECLiPSe and SICStus CHR) is fully operational and has been used in a prototype application involving the integration of the information systems of a medium-size company. Source and domain models are described using rules of the form (9), which are then automatically

[10] Such query templates correspond to our goal reduction rules. However, there is no equivalent of our forward propagation rules (which allow more sophisticated query planning).

[11] The interactions between sources from different "buckets" (in the terminology of [LRO96]) need to be taken into account.

[12] Consistency tests in a sophisticated language like CARIN [LR98] being "lazy" (an external complex consistency verifier for CARIN needs to be invoked).

[13] Alternatively we could unfold the integrity constraints (ICs) until they involve only source predicates. However, this can lead to a blow-up of the number of such ICs. This is also problematic in the case of recursive rules.

translated by the system into CHR goal reduction and propagation rules (12) and (13).

Real-world applications of Constraint Programming typically involve heterogeneous types of constraints as well as constraint solving strategies. The main problems faced in our I^3 application have shown the need of a *unified* framework for both representing and reasoning with such heterogeneous constraints. We argue that the combination of Constraint Handling Rules with the specialised constraint reasoning modules offered by the host CLP environment enables the development of an efficient and *flexible* query planner, dealing with complex source and domain descriptions. Thus, CHRs are appropriate not only for declarative representations of constraints, but also for the flexible control of the reasoning involved. Note that the ability to reason about such information source models during query planning is a form of semantic query optimisation (early detection of inconsistencies at planning time enables sometimes answering queries without accessing the sources if enough knowledge is available in the source and domain dscriptions).

The system currently deals only with *information* integration. We are currently extending our system to deal with database updates and applications with side-effects in the context of *system* integration.

References

[AART97] Aquilino D., Asirelli P., Renso C., Turini F., MedLan: a Logic-based Mediator Language, IEI Technical Report B4-16, November 1997.

[AKH96] Y. Arens, C.A. Knoblock, Chun-Nan Hsu. Query Processing in the SIMS Information Mediator, Advanced Planning Technology, Austin Tate (ed), AAAI Press, 1996.

[BCVB99] S. Bergamaschi, S. Castano, M. Vincini e D. Beneventano. Intelligent Techniques for the Extraction and Integration of Heterogeneous Information, IJCAI-99 WS on III, Stockholm.

[BG97] S. Bressan and C.H. Goh. Answering queries in context. In Proceedings of the International Conference on Flexible Query Answering Systems, FQAS'98, Roskilde, Denmark, 1998.

[DG97] O.M. Duschka, M.R. Genesereth. Infomaster - An Information Integration Tool. Proc. International Workshop "Intelligent Information Integration", KI-97, Freiburg, Sept. 1997.

[DG97b] O.M. Duschka and Michael R. Genesereth. Query Planning in Infomaster, Proc.12th Annual ACM Symposium on Applied Computing, SAC '97, San Jose, Feb. 1997.

[F95] Fruewirth T. Constraint Handling Rules, in Podelski A. (ed) Constraint Programming: Basic and Trends, LNCS 910, 90-107, 1995.

[GPQ97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, J. Widom. The TSIMMIS approach to mediation: Data models and Languages. In Journal of Intelligent Information Systems, 1997.

[KKT98] Kakas A., Kowalski R., Toni F. The role of abduction in logic programming, Handbook of logic in AI and LP 5, OUP 1998, 235-324.

[KMM98] Kakas A., Michael A., Mourlas C. ACLP: a case for non-monotonic reasoning, Proc. NM'98.

[KTW98] Kowalski R., Toni F., Wetzel G. Executing suspended logic programs, Fundamenta Informaticae 34 (1998), 1-22.

[L99] Alon Y. Levy, Combining Artificial Intelligence and Databases for Data Integration. LNAI: Artificial Intelligence Today; Recent Trends and Developments 1999.

[LRO96] Alon Y. Levy, A. Rajaraman, J.J. Ordille, Querying Heterogeneous Information Sources Using Source Descriptions Proc. 22nd VLDB Conference, Bombay, India. 1996.

[LR98] A. Levy, M.C. Rousset, Combining Horn Rules and Description Logics in CARIN, AIJ 104, 1998.