Integration of Heterogeneous Knowledge Sources in the CALO Query Manager

Jose-Luis Ambite[1], Vinay K. Chaudhri[2], Richard Fikes[3], Jessica Jenkins[3], Sunil Mishra[2], Maria Muslea[1], Tomas Uribe[2], Guizhen Yang[2]

1. USC Information Sciences Institute, Marina del Rey, CA 90292, USA
2. Artificial Intelligence Center, SRI International, Menlo Park, CA 94087, USA
3. Knowledge Systems Group, Artificial Intelligence Laboratory, Stanford University, Stanford, CA 94305, USA

Abstract. We report on our experience in developing a query answering system that integrates multiple knowledge sources. The system is based on a novel architecture for combining knowledge sources in which the sources can produce new subgoals as well as ground facts in the search for answers to existing subgoals. The system uses a query planner that takes into account different query processing capabilities of individual sources and augments them gracefully. A reusable ontology provides a mediated schema that serves as the basis for integration. We have evaluated the system on a suite of test queries in a realistic application to verify the practicality of our approach.

1. Introduction

The problem of integrating and querying information residing in heterogeneous knowledge sources has been a focus of intensive research during the past several years. A number of information integration systems (for example, TSIMMIS [1], Garlic [2], Information Manifold [3], Infomaster [4], SIMS [5], and HERMES [6]) have been built and deployed successfully in multiple application domains.

In this paper, we report on our effort to build a real system for integrating heterogeneous knowledge sources with different query answering and reasoning capabilities. Our system improves on other traditional information integration systems in the following ways: (1) Knowledge sources integrated into our system may return logical formulas representing new subgoals as well as ground facts in response to queries; (2) The limited reasoning capabilities of individual knowledge sources are augmented by using a powerful query planner; and (3) A reusable, object-oriented ontology provides a mediated schema that serves as the basis for integration.

We are conducting this work in the context of CALO (Cognitive Assistant that Learns and Organizes), a multidisciplinary project funded by DARPA to create cognitive software systems that can reason, learn from experience, be told what to do, explain what they are doing, reflect on their experience, and respond robustly to surprises.[1] The current project is targeted at developing personalized cognitive assistants (which we will refer to as CALOs) in an office environment where knowledge about emails, schedules, people, and contact information, and so on is distributed among multiple knowledge sources. A CALO must be able to access and reason with this distributed knowledge. We have encapsulated this functionality in a CALO module called Query Manager that serves as the unified access point within a CALO for answering queries. In an office, multiple CALOs will each support a single user. Each copy of CALO has its own copy of Query Manager. The following three example queries typically arise in an office environment:

1. *Which meetings will have a conflict if the current meeting runs overtime by an hour?* Answering this query requires knowing the ending time of the current meeting, computing the new ending time, and determining which other meetings have been scheduled to pass over the new ending time. In a CALO, a user's schedules are stored in a personal information knowledge source called IRIS. The functionality of computing the new ending time is implemented in a reasoner, called Time Reasoner, that can evaluate simple functions and predicates on time points and intervals.
2. *Who was present in the meeting in conference room EJ228 at 10 a.m. this morning?* A person is considered by a CALO to be present in a meeting if it can recognize that person at the time of the

[1] See http://www.ai.sri.com/project/CALO/

meeting from images provided by the meeting room camera. This knowledge is stated as a rule in a knowledge source called Knowledge Machine (KM) [7]. Given this query, KM uses that rule to produce subgoals for retrieving meeting participants' information from the image analysis results provided by another independent knowledge source called Meeting Ontology Knowledge Base (MOKB). Therefore, in this example, one knowledge source produces new subgoals that are then evaluated by another knowledge source.

3. *List all the people who are mentioned in an article in which Joe is also mentioned?* This query requires first retrieving all the articles that mention Joe, and then for each of those articles, retrieving all the people mentioned in them. This requires selecting instances of a class based on a condition (articles that mention Joe), and then retrieving values of a property (people mentioned in an article) of each of those instances. This information is stored in IRIS. However, IRIS cannot process queries like this one that require successive retrievals from the same class based on different criteria. Therefore, Query Manager must take that limitation into account when developing query plans, and in this case form a plan that involves multiple calls to IRIS.

The examples above illustrate the challenges facing Query Manager in integrating knowledge sources and answering queries of different sorts. However, the intricacy of the internal workings of the system is completely transparent to its users — be they humans interacting with a CALO, or intelligent agents running on behalf of a CALO. Neither do users of Query Manager have to worry about how knowledge is expressed (e.g., as ground facts or axioms) or distributed in the system. Queries only need to be formulated using CALO Ontology[2], and Query Manager will determine which knowledge sources are required to produce the answers.

2. Query Manager Architecture

The Query Manager architecture is shown in Figure 1. It is based on an object-oriented modular architecture for hybrid reasoning, called the JTP architecture [8], which supports rapid development of reasoners and reasoning systems. The architecture considers each knowledge source to be a *reasoner*.

A CALO component (called a *client*) can obtain answers to a query and explanations for those answers by conducting a *query answering dialogue* with Query Manager. A *query* includes a *query pattern* that is a conjunctive sentence expressed in KIF (Knowledge Interchange Format) [11] in which zero or more logical constants have been replaced by variables. A *query answer* provides *bindings* of constants to some of these variables such that the *query pattern instance* produced by applying the bindings to the query pattern and considering the remaining variables in the query pattern to be existentially quantified is entailed by the knowledge sources in Query Manager. Query Manager can produce multiple answers to a query and be recalled to provide additional answers when needed.

When queries arrive at Query Manager, they first undergo syntactic validation and then are sent to the Asking Control Reasoner, which embodies two reasoning methods: iterative deepening and model elimination. The Asking Control Reasoner sends queries to the Asking Control Dispatcher, which calls three reasoners in sequence: Rule Expansion Reasoner, Query Planner, and Assigned Goal Dispatcher. Each reasoner accepts as input a goal in the form of a query pattern and produces a *reasoning step iterator* as output. A reasoning step is a partial or complete proof of a goal, and a reasoning step iterator is a construct that when pulsed provides a reasoning step as output. The Rule Expansion Reasoner applies rules to its input goal, and provides reasoning steps that contain new subgoals which can be used to produce answers for the input goal. The Assigned Goal Dispatcher answers a query by dispatching (groups of) subgoals to different reasoners per the query plan produced by Query Planner. Note that each knowledge source is integrated into Query Manager by implementing an asking reasoner per the JTP reasoner interface specification (see Section 3 for more details). We refer to such reasoners as *reasoning system adapters*.[3] These adapters encapsulate the source-specific details of evaluating queries.

Ideally, there would not be a separate rule expansion reasoner in Query Manager. Any reasoner should be allowed to produce new subgoals that are not in the original query plan, and the query plan would then be revised to include the new subgoals. But such a control structure would require dynamic query

[2] See http://www.cs.utexas.edu/~mfkb/RKF/tree/specs/ontologies/

[3] For convenience in exposition, from now on we will use the terms *reasoners* and *knowledge sources* interchangeably.

planning, which is outside the scope of the current phase of the project. We worked around this problem by storing rules in a separate reasoner (i.e., the Rule Expansion Reasoner) that is invoked before a query plan is generated. This solution is not completely general because it is not always possible to anticipate all such rules. Extending the architecture to support dynamic query planning is the subject of future work.

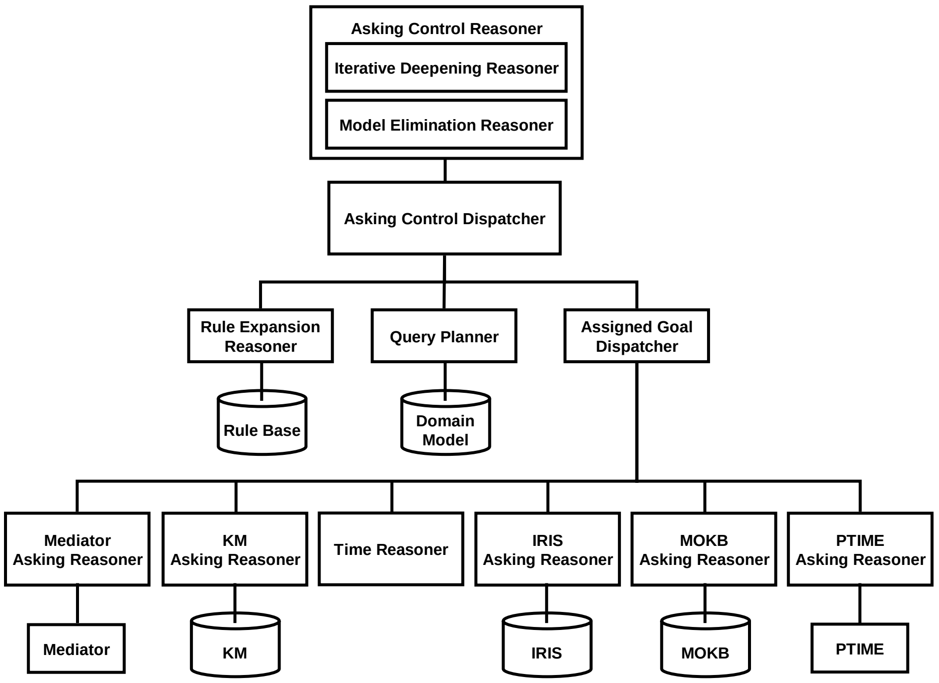


Figure 1. Query Manager Architecture

The design of Query Manager is based on a shared ontology called the CALO Ontology that serves as the *mediated schema* for all reasoners. All the reasoners are expected to implement (parts of) CALO Ontology. Developing such an ontology is usually the biggest challenge in implementing systems that access heterogeneous knowledge sources. We bootstrapped this process by reusing a large shallow ontology that we had developed in a previous project [9]. Clearly, this ontology was inadequate for the office domain that was the focus of the current project and needed to be extended. Our strategy for extending the ontology was to set up a collaborative process between the developers of the knowledge sources and the knowledge engineers maintaining the ontology. This strategy is not always possible in a typical project that integrates heterogeneous sources, but we had the advantage that the developers of the knowledge sources were also members of the team. For instance, development of CALO Office Ontology was a result of collaboration between IRIS developers and the knowledge engineers, and CALO Meeting Ontology a result of collaboration between MOKB developers and the knowledge engineers. This approach, however, may not generalize to other systems or may be expensive to implement if there is no control on the schemas used by individual knowledge sources. In these cases, a lot of translation work must be done.

In addition to the knowledge sources that we have introduced and shown in Figure 1 (KM, MOKB, Time Reasoner, and IRIS), two other reasoners are currently integrated into Query Manager: Mediator and PTIME. Mediator is a data integration system, developed independently by ISI, which extracts useful information from the Web [10]. For example, it is able to access the Web sites of Office Depot and CDW to extract product and pricing information for laptops. In its current deployment in the CALO project, Mediator is interfaced with about two dozen Web sites in the office equipment domain. PTIME is a general-purpose constraint reasoning system. It is currently used for scheduling meetings: given the constraints of the meetings to be scheduled, together with a user's preferences and calendar information, it can suggest alternative meeting times in the presence of schedule conflicts.

The current implementation of Query Manager is limited to only conjunctive queries. This is not an inherent limitation of the Query Manager architecture. The use of model elimination in Asking Control Reasoner permits acceptance of queries in full first-order logic. If we use a query language more expressive than conjunctive queries, then Query Planner will also need to be extended (a subject for future research). We also note that the use of iterative deepening provides termination guarantees in the presence of recursive rules, thanks to controlled search capabilities (this subject is not the main focus of this paper).

Next we illustrate the workings of Query Manager using the following example query: *Which meetings will have a conflict if the CALO Test meeting runs overtime by an hour?* This query can be formally expressed using KIF syntax as follows (note that names preceded by "?" represent variables):

```
(and   (CurrentCaloUser ?user)  (is-calendar-attendee ?user ?meeting)  (Event-Entry ?meeting)
       (calendar-summary ?meeting "CALO Test")  (has-end-date ?meeting ?end-date)
       (time-add ?end-date "PT60M" ?new-end-date)  (Event-Entry ?affected-meeting)
       (time-inside-event ?new-end-date ?affected-meeting)  (is-calendar-attendee ?user ?affected-meeting))
```

When Asking Control Reasoner receives this query, it first sends it to Rule Expansion Reasoner. For this example query, the knowledge base contains the following rule for the time-inside-event predicate:

```
(=>    (and (has-start-date ?e ?e-start) (has-end-date ?e ?e-end) (time-lt ?e-start ?time) (time-gt ?e-end ?time))
       (time-inside-event ?time ?e))
```

Using the above rule, Rule Expansion Reasoner rewrites the input query as follows (based on the techniques of backward chaining and unification):

```
(and   (CurrentCaloUser ?user)  (is-calendar-attendee ?user ?meeting)  (Event-Entry ?meeting)
       (calendar-summary ?meeting "CALO Test")  (has-end-date ?meeting ?end-date)
       (time-add ?end-date "PT60M" ?new-end-date)  (Event-Entry ?affected-meeting)
       (has-start-date ?affected-meeting ?e-start)  (has-end-date ?affected-meeting ?e-end)
       (time-lt ?e-start ?new-end-date)  (time-gt ?e-end ?new-end-date)
       (is-calendar-attendee ?user ?affected-meeting))
```

Since there is only one rule defining the time-inside-event predicate, Rule Expansion Reasoner returns this query above as the *only* reasoning step of a reasoning step iterator. Asking Control Dispatcher sends the goal of the resulting reasoning step, which is the rewritten query above, to Query Planner. Query Planner then generates a query plan with five groups: G1, G2, G3, G4, and G5 listed below. Note that we use a syntax like G1@IRIS to denote that the literals in G1 are to be processed at the knowledge source IRIS.

```
G1@IRIS (What are the current CALO user's schedule meetings?):
(and  (CurrentCaloUser ?user)  (is-calendar-attendee ?user ?meeting))

G2@IRIS (When does the CALO Test meeting end?):
(and  (Event-Entry ?meeting)  (calendar-summary ?meeting "CALO Test")  (has-end-date ?meeting ?end-date))

G3@Time Reasoner (What is the new end time?):
(time-add ?end-time "PT60M" ?new-end-date)

G4@IRIS (Which other meetings pass over the new end time?):
(and   (Event-Entry ?affected-meeting)  (has-start-date ?affected-meeting ?e-start)
       (has-end-date ?affected-meeting ?e-end)  (time-lt ?e-start ?new-end-date)  (time-gt ?e-end ?new-end-date))

G5@IRIS (Is this meeting on the current CALO user's calendar?):
(is-calendar-attendee ?user ?affected-meeting)
```

Query Planner generates the plan above by taking into account the capabilities of the knowledge sources that it knows about (details of Query Planner are in Section 3.2). For example, IRIS does not do time arithmetic; therefore, the portion of the query that involves adding two time points is sent to Time Reasoner. IRIS can only answer queries about instances of a single class, so the queries to IRIS need to be decomposed properly. The reasoning step iterator capturing the above query plan is returned to Asking Control Reasoner. The reasoning step[4] in this iterator contains G1, G2, G3, G4, and G5 as the set of unproven subgoals. Asking Control Reasoner processes this reasoning step by sending to Assigned Goal Dispatcher one subgoal at a time. The first subgoal G1 is processed at IRIS. The results from IRIS (returned as a collection of reasoning steps also; see Section 3.1 for more details) each specify a binding for the variable ?meeting. In processing each result, Asking Control Reasoner propagates the binding of ?meeting to the next subgoal G2 to be processed again at IRIS. When evaluation of G2 is finished, its results are returned to Asking Control Reasoner and variable bindings are propagated to subgoal G3. Such

[4] For simplicity in exposition we assume that there is only one reasoner step here.

retrieval and propagation of bindings continue until subgoals G3, G4, and G5 are all evaluated at the corresponding reasoner one after another. The series of bindings obtained by evaluating all the subgoals constitutes one answer to the original query.

To retrieve all the answers, Asking Control Reasoner backtracks in a depth-first manner, so it will first attempt to find additional solutions to subgoal G5. Once all the solutions to (the current instantiated version of) subgoal G5 are exhausted, it unwinds to G4 and looks for more answers. If there is one more answer to G4, Asking Control Reasoner propagates the bindings down to G5 and starts the evaluation of G5 anew. The process repeats until the answers for subgoals G4, G3, G2, and G1 are all exhausted in succession.

Note that in our architecture, as the reasoners return results, control always returns back up to Asking Control Reasoner. One may wonder about the role played by Asking Control Dispatcher if control is to always go up to Asking Control Reasoner. In fact, Asking Control Dispatcher is also implemented like a reasoner. It encapsulates the interface to individual reasoners so that they are not visible to Asking Control Reasoner and appear like a single, unified knowledge base. This hierarchical reasoner design feature is a main strength of the JTP architecture — knowledge sources can be grouped and integrated separately using control logics that are most suitable for the application.

3. Query Manager Components

We first introduce Query Manager's reasoner interface, and then present Query Planner and the reasoning system adapters. Finally, we present an overview of Query Manager API.

3.1 Reasoner Interface

In the Query Manager architecture, the reasoner interface is needed to specify a protocol for integrating knowledge sources into Query Manager. This interface is based on an abstraction called a *reasoning step*. A reasoning step consists of the following elements:

1. A *claim*, which is a sentence that this reasoning step justifies;
2. A set *of premises* on which the justification relies, each of which is a sentence;
3. A set of *child reasoning steps* on whose claims this justification relies
4. A set of *variable bindings*;
5. An *atomic justification* for the claim of this reasoning step given its premises and the claims of its children.

An atomic justification can be either "Axiom", or an inference rule. Reasoning steps justified as Axiom do not have any premises or children, while an inference rule serves to infer the reasoning step's claim from its premises and the claims of its child reasoning steps.

Query Manager interacts with a reasoner through the interface by sending the reasoner a goal to prove. The reasoner responds by returning reasoning steps that are either *partial* or *complete proofs* of the goal. A complete proof is a reasoning step that has no premises and no descendant reasoning steps that have any premises. A *partial proof* is a reasoning step that has at least one unproved premise, either directly in itself or indirectly in a descendant reasoning step. Query Manager can derive a complete proof from a partial proof by proving these premises. Note that any reasoning step whose atomic justification is Axiom is a proof. To facilitate returning of multiple reasoning steps corresponding to different proofs of a goal, a reasoner actually returns a *reasoning step iterator*. Query Manager accesses the reasoning steps generated by a reasoner via the reasoning step iterator returned. The reasoner determines the order in which the reasoning steps are accessed via the iterator, and Query Manager determines when to retrieve reasoning steps from the iterator. This provides a framework for streaming answers and in which reasoners can generate answers as needed.

We first show an example of how this reasoning interface is used for interfacing Query Planner with Query Manager. Suppose Query Planner receives the following query as input: ($b \wedge d \wedge a \wedge c$), where b, d, a, and c are positive, atomic literals, and decomposes this query into subgroups in two different ways. Query Planner will produce a reasoning step iterator. Each time Query Manager invokes this iterator a reasoning step will be returned corresponding to one way of decomposing the given query. For example, the following illustrates the two reasoning steps generated:

```
Query Planner Reasoning Step 1:
Claim: (b ∧ d ∧ a ∧ c)
Premise 1: b
Premise 2: (d ∧ a ∧ c)
Justification: {Premise 1, Premise 2; AND introduction}

Query Planner Reasoning Step 2:
Claim: (b ∧ d ∧ a ∧ c)
Premise 1: (b ∧ d)
Premise 2: (a ∧ c)
Justification: {Premise 1, Premise 2; AND introduction}
```

Note that in order to complete the proof of the claim in each reasoning step, Query Manager needs to find proofs of the two premises in each reasoning step. As another example, let us revisit the example presented in Section 2. The query sent to TIME Reasoner is subgoal G3:

```
(time-add ?end-date "PT60M" ?new-end-date)
```

Assume that the preceding subgoal l G2, sent to IRIS, retrieves a binding for ?end-date, say "20050520T083000" which is a lexical representation of 8:30 a.m. on May 5, 2005. The response from Time Reasoner is a reasoning step iterator with a single reasoning step with the following structure:

```
Time Reasoner Reasoning Step 1:
Claim: (time-add "20050520T083000" "PT60M" ?new-end-date)
Justification: Axiom
Variable bindings: ?new-end-date "20050520T093000"
```

As the last example of a reasoning step, we consider the reasoning step returned by Rule Expansion Reasoner shown below. For this reasoning step, the input query is the claim. The rule application in the knowledge base is a child reasoning step. The expanded query is the resulting premise. The justification structure encodes the inference rules used in expanding the query.

```
Rule Expansion Reasoner Reasoning Step:
Claim: (and (CurrentCaloUser ?user)  (is-calendar-attendee ?user ?meeting)  (Event-Entry ?meeting)
            (calendar-summary ?meeting "CALO Test")  (has-end-date ?meeting ?end-date)
            (time-add ?end-date "PT60M" ?new-end-date)  (Event-Entry ?affected-meeting)
            (time-inside-event ?new-end-date ?affected-meeting)  (is-calendar-attendee ?user ?affected-meeting))
Child Reasoning Step 1:
  Claim: (=> (and (has-start-date ?e ?e-start) (has-end-date ?e ?e-end) (time-lt ?e-start ?time) (time-gt ?e-end ?time))
             (time-inside-event ?time ?e))
  Justification: Axiom
Premise 1: (and  (CurrentCaloUser ?user)  (is-calendar-attendee ?user ?meeting)  (Event-Entry ?meeting)
                 (calendar-summary ?meeting "CALO Test")  (has-end-date ?meeting ?end-date)
                 (time-add ?end-date "PT60M" ?new-end-date)  (Event-Entry ?affected-meeting)
                 (has-start-date ?affected-meeting ?e-start)  (has-end-date ?affected-meeting ?e-end)
                 (time-lt ?e-start ?new-end-date)  (time-gt ?e-end ?new-end-date)
                 (is-calendar-attendee ?user ?affected-meeting))
Justification: {Child Reasoning Step 1, Premise 1, AND elimination, Modus Ponens, AND introduction}
```

3.2 Query Planner

The key capabilities of Query Planner are *subgoal grouping* and *subgoal ordering*. As an example of subgoal grouping, consider a query (b ∧ d ∧ a ∧ c) that could be broken into two groups, one consisting of b to be evaluated at one reasoner and the other (d ∧ a ∧ c) at another reasoner. For an example of subgoal ordering, consider a query (u ∧ v ∧ s ∧ t) in which the conjuncts must be evaluated in the following order: s, v, u, and then t.

The subgoal grouping and ordering produced by Query Planner satisfies the following requirements:

- Satisfaction of binding pattern restrictions [12] of the predicates accepted by each reasoner.
- Modeling of predicate completeness and overlap, that is, whether a reasoner can provide the complete extension of a predicate or just a subset. Query Planner uses this information to minimize the number of reasoners accessed.

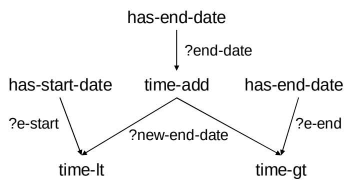
- Satisfaction of the query processing capabilities of different reasoners.
- Grouping of the subgoals in the query into maximal subgroups that can be handled by each reasoner, satisfying both the binding pattern restrictions and the reasoner query answering capabilities.

The information about binding constraints, predicate completeness, and query answering capabilities is encoded in what is called a *capability model* in Query Manager (due to space limitations we will omit the details of its syntax here and outline its key ideas). The query planning algorithm in Query Manager consists of the following phases:

1. **Satisfaction of binding constraints**. It constructs a dependency graph to satisfy the binding pattern restrictions of the predicates in the query. Since the dependency graph is a partial order, there may be many different groupings, depending on how we process the dependency graph. In our implementation, we output either a single group or all the groups.
2. **Completeness reasoning**. A predicate appearing in multiple reasoners may have one of two semantics:
   a. Incomplete (Open-world): Each reasoner may contribute different bindings to the predicate. Query Manager needs to query all the reasoners and union the results. For example, if two sources know about the predicate (Laptop-Computer X), then each reasoner may provide different laptops.
   b. Complete (Closed-world): Each reasoner has a replica of the predicate. Query Manager needs to query only one of the reasoners (preferably the most cost-effective one). For example, a predicate like ISI-Employee(X) may correspond to the directory listing at ISI and have all ISI employees; there is no need to go to any other sources.
3. **Selection assignments.** Each (in) equality predicate (=, >, <, ≥, ≤, ≠) is assigned to the first group where it can go according to the dependency graph in order to filter intermediate results as soon as possible.
4. **Enforcement of reasoner query answering capabilities**. Currently, the only query restriction is that IRIS cannot execute joins across classes. An analysis step further decomposes queries to IRIS into subgroups such that each group is a single-class query.

To clarify the behavior of Query Planner, consider the motivating example query in Section 2. The fragment of the capability model for the predicates and the dependency graph for this example are as follows:

```
(CurrentCaloUser  Person:f)@[KM, IRIS]
(is-calendar-attendee  Event:f  Person:f)@[KM, IRIS]
(Event-Entry  Event:f)@[KM, IRIS]
(calendar-summary  Event:f  String:f)
(has-end-date  Event:f  dateTime:f)@[KM, IRIS]
(time-add  dateTime:b  timeDuration:b  dateTime:f)@[Time Reasoner]
(has-start-date  Event:f  dateTime:f)@[KM, IRIS]
(time-lt  dateTime:b  dateTime:b)@[Time Reasoner/IRIS]
(time-gt  dateTime:b  dateTime:b)@[Time Reasoner/IRIS]
```



Note that nodes in the dependency graph represent predicates in the query, and edges are labeled with variable names. For instance, a directed edge labeled with ?end-date points from has-end-date to time-add. This means time-add depends on has-end-date to provide a binding for variable ?end-date. The entry (is-calendar-attendee Event:f Person:f)@[KM, IRIS] in the capability model states that the relation is-calendar-attendee is known to both KM and IRIS, its first argument is an Event, its second argument is a Person, and both arguments can be free (which also means both can be bound). The entry (time-lt dateTime:b dateTime:b)@[Time Reasoner/IRIS] says that both Time Reasoner and IRIS can evaluate the predicate time-lt and both of its arguments must be bound. The "/" symbol separating these two reasoners means that this predicate is complete. Therefore, Query Planner can choose either Time Reasoner or IRIS to evaluate the predicate. By default, predicates are deemed incomplete, which means that all relevant reasoners of a predicate must be queried to get all the answers.

Given the specification above, Query Planner first annotates the given query with the reasoners that can handle each of the predicates. The annotated query is as follows:

```
(and    (CurrentCaloUser ?user)@[KM, IRIS]
        (is-calendar-attendee ?user ?meeting)@[KM, IRIS]
        (Event-Entry ?meeting)@[KM, IRIS]
        (calendar-summary ?meeting "CALO Test")@[KM, IRIS]
        (has-end-date ?meeting ?end-date)@[KM, IRIS]
        (time-add ?end-date "PT60M" ?new-end-date)@[Time Reasoner]
        (Event-Entry ?affected-meeting)@[KM, IRIS]
        (has-start-date ?affected-meeting ?e-start)@[KM, IRIS]
        (has-end-date ?affected-meeting ?e-end)@[KM, IRIS]
        (time-lt ?e-start ?new-end-date)@[Time Reasoner/ IRIS]
        (time-gt ?e-end ?new-end-date)@[Time Reasoner/IRIS]
        (is-calendar-attendee ?user ?affected-meeting)@[KM, IRIS])
```

Semantically, the annotated query represents a conjunction of disjunctive queries. For instance, the statement (CurrentCaloUser ?user)@[KM, IRIS] is equivalent to the following disjunctive query:

```
(or     (CurrentCaloUser ?user)@KM  (CurrentCaloUser ?user)@IRIS)
```

Query Planner searches the annotated query for maximal groups of predicates that can be executed at the same reasoner while satisfying the binding constraints as shown in the above dependency graph. During the search the planner ensures that the predicates sharing object IDs belong to the same reasoner. In general, object IDs are internal to each reasoner and thus cannot be "joined" across different reasoners.[5] For example, consider the following fragment from the above annotated query:

```
        (Event-Entry ?meeting)@[KM,IRIS]  (is-calendar-attendee ?user ?meeting)@[KM,IRIS]
```

These are four possible ways of combining these two predicates:

(Event-Entry ?meeting)@[KM] (is-calendar-attendee ?user ?meeting)@[KM]	(Event-Entry ?meeting)@[KM] (is-calendar-attendee ?user ?meeting)@[IRIS]
(Event-Entry ?meeting)@[IRIS] (is-calendar-attendee ?user ?meeting)@[KM]	(Event-Entry ?meeting)@[IRIS] (is-calendar-attendee ?user ?meeting)@[IRIS]

Observe that variable ?meeting will be bound to object IDs. Therefore, of these plan fragments, only those (two) plans where variable ?meeting is bound by the same reasoner in both literals are valid; the others would simply not produce any answers. This heuristic has a significant impact as the size of the query (in terms of the number of predicates) increases. So after taking into account the assignment of predicates to reasoners and the dependency graph for the query, the query planning algorithm produces the following two query plans. One plan uses only KM to retrieve calendar information:

```
(and    (and    (CurrentCaloUser ?user)  (is-calendar-attendee ?user ?meeting)
                (Event-Entry ?meeting)  (calendar-summary ?meeting "CALO Test")
                (has-end-date ?meeting ?end-date)
                (Event-Entry ?affected-meeting)  (has-start-date ?affected-meeting ?e-start)
                (has-end-date ?affected-meeting ?e-end)  (is-calendar-attendee ?user ?affected-meeting))@KM

        (time-add ?end-date "PT60M" ?new-end-date)@Time Reasoner

        (and    (time-lt ?e-start ?new-end-date)  (time-gt ?e-end ?new-end-date))@Time Reasoner)
```

and the other uses IRIS only for calendar information:

[5] In the future, we plan to use record linkage techniques to match objects in different reasoners that represent the same object in the real world.

```
(and   (and    (CurrentCaloUser ?user)  (is-calendar-attendee ?user ?meeting)
               (Event-Entry ?meeting)  (calendar-summary ?meeting "CALO Test")
               (has-end-date ?meeting ?end-date))@IRIS

       (time-add ?end-date "PT60M" ?new-end-date)@Time Reasoner

       (and    (Event-Entry ?affected-meeting)  (has-start-date ?affected-meeting ?e-start)
               (has-end-date ?affected-meeting ?e-end)  (time-lt ?e-start ?new-end-date)
               (time-gt ?e-end ?new-end-date)  (is-calendar-attendee ?user ?affected-meeting))@IRIS)
```

Observe that both query plans have three groups. Since only Time Reasoner can handle the predicate time-add, it must be evaluated separately. All these groups in each plan must be evaluated in succession so as to satisfy the binding constraints: evaluation of the first group will provide bindings for ?end-date needed by the predicate time-add in the second group, which will in turn provide bindings for the variable ?new-end-date needed by the time comparison operators in the third group.

Finally, Query Planner must still take into account the fact that IRIS can only answer queries about instances of a single class. This, however, does not result in a multiplication of the number of plans. Instead, a further rewrite of the second query plan above suffices. Its final plan is as follows:

```
(and   (and    (CurrentCaloUser ?user)  (is-calendar-attendee ?user ?meeting))@IRIS

       (and    (Event-Entry ?meeting)
               (calendar-summary ?meeting "CALO Test")  (has-end-date ?meeting ?end-date))@IRIS

       (time-add ?end-date "PT60M" ?new-end-date)@Time Reasoner

       (and    (Event-Entry ?affected-meeting)  (has-start-date ?affected-meeting ?e-start)
               (has-end-date ?affected-meeting ?e-end)
               (time-lt ?e-start ?new-end-date) (time-gt ?e-end ?new-end-date))@IRIS

       (is-calendar-attendee ?user ?affected-meeting)@IRIS)
```

3.3 Reasoning System Adapters

We report on our experience in implementing reasoning system adapters for the knowledge sources integrated into Query Manager. Additional coding is needed to implement the reasoner interface (explained in Section 3.1) for each knowledge source to satisfy the architectural requirements of Query Manager. The technical difficulties encountered in performing this integration can be broadly categorized into the following three categories: ontology mismatches, impedance mismatches, and evaluation model mismatches.

Ontology Mismatches. Each reasoner is required to accept queries expressed using the mediated schema: CALO Ontology. This commitment was easily fulfilled by KM, because it has native support for CALO Ontology. In some cases, the individual reasoners did not have an ontology from which to start. For such cases, the ontology was developed collaboratively as described in Section 2, and there were no ontology mismatch issues. One notable exception is Mediator, which maps the ontologies used by the Web extraction agents into CALO Ontology. These agents produce XML documents that correspond to a flat relational schema. For example, an agent that wraps the flatbed scanner portion of the compusa.com Web site exposes the following flat relational schema (we have omitted numerous other extracted attributes for the sake of brevity):

```
compusaflatbedscanner(hasModelName, hasManufacturer, hasVendor, hasTotalPrice, hasHorizontalScanSize,…)
```

The primary difficulty with this representation is that it leaves many things implicit. For example, it does not say the units of price or scan size. In general it is good design style to make these things implicit. To address the ontology mismatch issue, Mediator uses local-as-view translation axioms [3, 10] to make the implicit semantics explicit and to facilitate query translation. For instance, the following translation axiom was added to the information extraction agent for compusa.com (note that in the Datalog-style rule body, unary predicates correspond to class membership queries while binary predicates represent property-value queries in CALO Ontology):

```
compusaflatbedscanner(hasModelName, hasManufacturer, hasVendor, hasTotalPrice, hasHorizontalScanSize,…) ⇐
  Flatbed-Scanner(X),  hasModelName(X, hasModelName),
  hasManufacturer(X, M),  Company(M),  called(M, hasManufacturer),
  hasVendor(X, V),  Company(V),  called(V, hasVendor),
  hasTotalPrice(X, WVal),  Worth-Value(WVal),  magnitude-cardinal(WVal, "*usdollar", hasTotalPrice),
  hasHorizontalScanSize(X, HSS),  Length_Value(HSS),  magnitude_cardinal(HSS, "*inch", hasHorizontalScanSize),
  …
```

Using translation axioms like these, Mediator mapped the data returned by Web extraction agents and provided it with semantics according to CALO Ontology. The resulting system was cleanly integrated and very easy to work with. In addition, it had the advantage that one did not always need to specify all the parameters of the query, and could retrieve only a subset of the attributes.

Another reasoner, PTIME, only supports interfaces similar to the initial design of Mediator. Moreover, part of its interface is nonrelational, that is, it supports nested relations. This kind of unconformity caused substantially high integration overhead.

Impedance Mismatches. Query Manager uses KIF as its interlingua and expects all reasoners to support compatible interfaces. However, not all the reasoners support KIF (or any logic language interface at all). For instance, IRIS does not have a logical query language through which users can pose queries. Instead, it provides a Java-based API, called Semantic Objects API, for querying its knowledge store. Therefore, the IRIS reasoning system adapter needs to map a query stated in a logical syntax into a Java function call. For example, consider the following query (in KIF syntax), which retrieves the meetings scheduled to start between 9:30 a.m. and noon and to end no later than 4:30 p.m. on May 1, 2005:

```
(and  (Meeting ?meeting)  (hasStartDate ?meeting ?start)  (hasEndDate ?meeting ?end)
      (time-gte ?start "20050501T093000")  (time-lte ?start "20050501T120000")
      (time-lte ?end "20050501T163000")
```

For the query above, the IRIS reasoning system adapter needs to assemble the following Java data structure for querying IRIS via its Semantic Objects API:

```
Query query = new IntersectionQuery(
        new MemberQuery("Meeting"),
        new IntersectionQuery(
                new IntersectoinQuery(
                        new PropertyValueQuery("hasStartDate", GTE, date1),
                        new PropertyValueQuery("hasStartDate", LTE, date2)),
                new PropertyValueQuery("hasEndDate", LTE, date3)));
```

where date1, date2, and date3 stand for Java Date variables corresponding to the dateTime values as specified in the strings "20050501T093000", "20050501T120000", and "20050501T163000", respectively.

IRIS cannot evaluate comparison constraints separately. Instead, comparison constraints must be "folded" into a relevant Semantic Objects API. For instance, in the translation above, the following two predicates:

```
(hasStartDate ?meeting ?start)  (time-gte ?start "20050501T093000")
```

have been folded into one single API:

```
PropertyValueQuery("hasStartDate", GTE, date1)
```

After constructing the data structure above, the IRIS reasoning system adapter invokes the following Semantic Objects API to retrieve answers from IRIS KB:

```
ItemSet msgSet = kbConnection.run(query).getResultSet();
```

We should note that conceptually the object-oriented Semantic Objects API returns query results in terms of the set of objects that meet the specified conditions. Because queries expressed in logic are meant to retrieve bindings for variables, the IRIS adapter also needs to perform appropriate postprocessing to retrieve bindings for the variables present in the original query. In our implementation, this is done by building a Hash map to gather the necessary variables information during the preanalysis stage (which is also needed to fold comparison constraints into Semantic Objects API). After the query results are returned from IRIS, using the mappings between variables and slot names (binary predicates) or class names (unary predicates), the IRIS adapter queries each individual object to retrieve the bindings needed. The following skeleton code illustrates the programming structure used to retrieve bindings from Semantic Objects:

```
// The variable-property pairs are stored in a Hash map data structre: this.mappings.
Variable[] var = (Variable[]) this.mappings.keySet().toArray(new Variable[nMappings]);

// Construct property mappings for variables.
String[] propertyURI = new String[nMappings];
for (int n = 0; n < nMappings; n++)
   propertyURI[n] = (String) this.mappings.get(var[n]);

for (int x = 0; x < nResults; x++) {
   SemanticObject oneResult = (SemanticObject) (resultSet.getItem(x));

   for (int n = 0; n < nMappings; n++) {
      initialBindings[n] = oneResult.getAllValues(propertyURI[n]);
      ...
```

Finally, it is worth noting that the translation from KIF queries to Semantic Objects API has linear time complexity in our implementation.

Evaluation Model Mismatches. The Query Manager architecture is designed so that any time a reasoner is unable to fully answer a query, it can return a partial proof that encodes a list of unproven goals. We leveraged this feature to a great advantage in interfacing with Query Planner: a query plan is simply a list of unproven goals. Reasoners that are tailored to be components in hybrid systems often feature such "partial proof" mechanisms, allowing them to be used, for example, in architectures such as theory resolution [13]. Most "black box" or off-the-shelf reasoners, however, are not set to return a list of unproven goals. Here, we take KM as an example reasoner, and show how it can be adapted, at low cost, to produce partial proofs.

The normal mode of operation for KM is to receive a query and produce an answer using the knowledge it has available locally. When answering a query requires accessing facts residing in other sources such as IRIS or MOKB, this model breaks down. The ground facts must be retrieved by invoking external reasoners, but we do not know which ground facts are needed until the query is partially evaluated. The need for retrieving such ground facts from other reasoners can be naturally modeled using partial proofs. We can do this by suspending a query at a point where ground facts are required, and resuming the query once the data is available.

Assume that a subgoal (G x_1, ..., x_n) is given to KM, where x_1, ..., x_n are free variables. While evaluating this goal, one of the following scenarios may happen:

1. KM succeeds unconditionally, returning a set of bindings for x_1, .., x_n. For example, for the subgoal (extension Joe ?x), KM may return the binding {?x: "5555"}.
2. KM returns a new subgoal, which, if proved, implies the original goal. For example, if the subgoal (G x_1, ..., x_n) is (Computer ?x), KM could do taxonomic inference to return two subgoals: (Laptop-Computer ?x) and (Desktop-Computer ?x), which represent subclasses of computers.
3. In its computation, KM encounters a subgoal that it does not know how to evaluate. For example, KM needs to compute (next-event p q), but the knowledge about next-event may reside in a separate plan monitoring or execution module. If KM knew this value, it could continue the computation and, perhaps, find an answer. In addition, some bindings for the other variables may have already been computed. We can solve this problem by having KM return a special formula that indicates

   a. predicate(s) that KM needs to know about in order to continue its work
   b. subgoal it was working on when it got stuck
   c. bindings obtained so far

   Query Manager will then apply a special control strategy to this kind of formula: it will always first work on the predicate(s) that KM needs to know about. When these are satisfied, Query Manager will give the subgoal that KM was working on back to KM again, but this time with the newly satisfied predicate(s) as background information/assertions for KM.

This approach illustrates how a reasoner can be adapted to work with the hybrid reasoning architecture of Query Manager. The only change required of the reasoner is the ability to identify the missing predicates that caused a partial proof attempt to fail. Depending on the implementation of the reasoner, it may be possible to include a continuation together with the partial proof so that the reasoner's work that led to interruption of the subgoal need not be redone. However, a simpler approach would be to just ask the original subgoal, providing data for the missing predicate(s).

The main challenge of the partial proof mechanism is that it may require replanning of query plans if evaluation of missing predicates further produces subgoals. Since dealing with dynamic planning is out of the scope for the current work, this is left open for future research.

3.4 Query Manager API

Query Manager provides an API for other CALO components to use in conducting *query-answering dialogues* with the Query Manager to obtain answers to queries and explanations for those answers. The API is based on the OWL QL[6] candidate standard language and protocol for query-answering dialogues among Semantic Web computational agents.

The API shares many features with standard SQL APIs, such as binding patterns, bundling of answers, and ordering.[7] Even though these features have been standard in conventional database management systems, it is not common for heterogeneous knowledge integration systems to support such features. The primary difficulty in supporting these features in a heterogeneous system is that not every individual knowledge source may be capable of supporting each of the features of the API. In such cases, either the capabilities of the API must be relaxed or some wrapping code must be implemented to fill in the missing functionality. As an example, consider the concept of answer bundles that requires a specific number of answers to be returned at a time while the constraint reasoner PTIME does not support this capability.

We consider here two features of the Query Manager API that are novel: subsumption, and returning answers in the order of increasing quality. The subsumption feature is highly desirable when the answers may be logical formulas, and one needs to ensure that successive answers are more specific. We consider these two features in more detail.

Subsumption. Query Manager ensures that each answer it returns to a client in a query-answering dialogue is not a duplicate of and is not subsumed by any of the answers it has returned earlier in the dialogue.[8] One answer subsumes another if its bindings are a superset of the bindings in the other answer. Thus, Query Manager may return an answer that does not include bindings for all the may-bind variables and later return a more specific answer that is the same as the earlier answer except that there are bindings for additional may-bind variables. However, Query Manager will not gratuitously return answers that are less specific than answers it has already returned.

Ranking. The Query Manager API supports ranking of query answers with respect to the binding of a variable in the query pattern as follows. A query can optionally designate that one of the must-bind variables is to be the "scoring variable". The best answers are then considered to be those that produce the highest values of the bindings of this scoring variable. If a query pattern does not contain a variable that is appropriate to be used as a scoring variable, a new literal can be conjoined to the query pattern that consists of a functional predicate, one or more arguments that are variables occurring elsewhere in the query pattern, and a final argument that is a new variable which can serve as the scoring variable. Determining a value for the final argument in such a literal then becomes a part of the task of answering the query.

A query designates a scoring variable by making it the value of keyword *best-first-by* or of keyword *better-next-by*. If the scoring variable is the value of best-first-by, then the answers in the first answer bundle will be the best answers that Query Manager can produce as ranked by the scoring variable. The answers in each successive answer bundle will be the next best answers that Query Manager can produce, and the answers in each answer bundle will be ordered best-first using the scoring variable. If the scoring variable is the value of keyword better-next-by, then all the answers in each successive answer bundle will be better than any of the answers in the previously returned answer bundle as ranked by the scoring variable, and the answers in each answer bundle will be ordered best-last using the scoring variable.

[6] See http://ksl.stanford.edu/projects/owl-ql/

[7] See http://www.postgresql.org/docs/8.0/static/index.html

[8] Thus, in OWL-QL terminology, Query Manager would be considered to be a serially terse server.

Finally, a query and a Query Manager continuation can optionally include a *threshold* number specifying that all the answers in the next answer bundle should have a value of the scoring variable no less than that number.

With these features, the API supports anytime reasoning, as well as branch-and-bound type of search, where answers of increasing quality are produced, contingent on the available resources.

4. Evaluation

We consider both qualitative and quantitative evaluation of the Query Manager approach for querying information across multiple knowledge sources. In describing qualitative evaluation, we consider questions such as: *What were the benefits of using the hybrid reasoning architecture such as JTP? How well did CALO Ontology work as a mediated schema? How effective was the query planner in producing query plans? Is the overall approach scalable?* In the quantitative evaluation, we consider a suite of sample queries and look at metrics such as query planning time, number of plans generated, and so on.

Quantitative Evaluation. We evaluated the performance of Query Manager in the context of an application of CALO in the office domain. Specifically, we chose a set of queries from the following categories and studied various aspects of answering these queries in Query Manager (for simplicity, we have omitted the KIF representation of these queries and stated them in English without all the details):

1. Task Fulfillment: Purchasing a piece of office equipment such as a laptop, printer, or scanner.

   (TF1) List all vendors from which purchase of flatbed scanners can be made.
   (TF2) Get real-time quotes, including price, vendor information, horizontal and vertical resolution, for flatbed scanners with price less than $200.
   (TF3) Show models of flatbed scanners having the greatest number of positive product reviews.
   (TF4) Show purchase request forms that were filled in for the latest purchase of flatbed scanners.

2. Task Setup: Setting up an office task such as arranging a meeting.

   (TS1) List all schedule conflicts if the meeting "CALO Test" goes overtime by an hour.
   (TS2) Show the best times to schedule a one-hour meeting between 9 a.m., May 10, 2005 and 3 p.m., May 12, 2005.
   (TS3) List all existing meetings that will be canceled if the current meeting is scheduled.

3. Task Discussion: Observing a meeting in which participants are discussing a project schedule.

   (TD1) List all participants of a particular meeting.
   (TD2) Show all meeting artifacts that were in focus in a particular meeting.
   (TD3) Show the person who answered the question raised by Joe during a particular meeting.

For the queries above, we consider the following metrics in our test (results are shown in columns of Table 1):

- Number of literals in the original KIF encoding of the query (Column 21)
- Number of rule expansions involved in answering the query (Column 3)
- Number of conjunctive query plans generated by Query Planner (Column 4)
- Number of knowledge sources involved in answering the query (Column 5)
- Number of groups in query plans generated for this query (Column 6)
- Total running time taken by Query Planner in answering the question (Column 71)

We conducted our test using a publicly released version of CALO v2.0. Our test machine was an IBM T42 laptop with 1.5 GB of main memory, 1.7 GHz Intel Centrino CPU, and running on Windows XP Professional. Our system was implemented in Java and we used JRE v1.4.2 for our test. To get an idea of the performance of Query Manager in real running environments, we used the measurement of elapsed time. We executed each query 10 times. The running time of Query Planner as shown in Column 6 of Table 1 was the average over these 10 runs.

First, observe that the running time taken by Query Planner alone in answering these questions ranges from 40 to 200 ms. This level of performance is satisfactory in real running environments since many of these queries take much longer than 200 ms to execute. For instance, TF2 normally takes a couple of minutes because in answering this query Mediator needs to fetch HTML documents from multiple Web sites and then perform data extraction operations on these documents. Therefore, the overhead introduced by Query Planner is not significant at all.

In contrast to questions in the Task Fulfillment and Task Setup categories, answering questions in the Task Discussion category requires various degrees of rule expansion because of the limited reasoning capability of MOKB, which mainly serves as a persistent triple store and does not support reasoning.

Query	# Literals	# Expansion	# Plans	# Sources	# Groups	QP Time (ms)
TF1	10	0	1	1	1	42.1
TF2	13	0	2	2 (15)[a]	2	52.1
TF3	5	0	2	2 (17)[b]	2	20.0
TF4	9	0	1	1	1	38.1
TS1	15	0	72	5	585	77.1
TS2	20	0	2	3	4	85.1
TS3	22	0	1	2	3	98.0
TD1	1	3	38	4	150	85.1
TD2	10	6	72	4	314	202.1
TD3	12	1	18	4	47	69.1

[a] Query Manager called Mediator which in turn called 15 Web information extraction agents, one per Web site.

[b] In addition to the 15 Web information extraction agents, two other sources were used for retrieving reviews by Mediator.

Table 1. Evaluation of Test Questions

Clearly, the number of query plans increases with the number of rule expansions involved, because each different rule expansion results in a different query plan. This correlation can be validated in questions TD1, TD2, and TD3. The number of knowledge sources involved also has an impact on the number of query plans generated, as can be observed in question TD3. The large number of query plans generated for questions TS1, TD1, TD2, and TD3 is due to a limitation in our current implementation: our domain model is not expressive enough for pruning all query plans that are semantically equivalent. In particular, questions TS1, TD1, TD2, and TD3 used a large number of time comparison operators that are declared to be implemented by several reasoners in their domain models. Even though the "Completeness" reasoning discussed in Section 3.2 can be utilized to address this problem, predicates with operator-like semantics need to be optimized with additional constraints: Ideally, operators should be pushed down as close to the appropriate knowledge sources as possible; randomly selecting a knowledge source for this operator in general does not have good performance guarantees. Currently, we are working on developing a new domain model (in combination with a statistics model) for Query Manager to address this problem.

In Column 2 of Table 1, the number of literals listed is for literals in the original KIF encoding. In most cases, rule expansion will add literals to a query. If no rule expansion is involved, the numbers in Column 2 are good indications of the size of the query submitted to Query Planner. In these cases (questions TF1-4 and TS1-3), we can observe that the running time of Query Planner is proportional to the size of its input. The seemingly long running times for questions TD1 and TD2 need some further explanation. In the case of question TD1, rule expansion results in two final conjunctive queries, one with 15 literals and the other with 3 literals. Therefore, the total input size to Query Planner is 18 and we can see that the total running time on these 18 literals (85.1 ms) is comparable to that of question TS2 (20 literals, 85.1 ms). In the case of TD2, rule expansion eventually results in two conjunctive queries, each with 23 literals. We can observe that in this case the total running time roughly doubles that of question TS3 (22 literals, 98.0 ms).

Qualitative Evaluation. We now turn to the qualitative evaluation of the Query Manager approach for querying heterogeneous knowledge sources and answer various questions raised at the beginning of this section.

What were the benefits of using the JTP hybrid reasoning architecture? The primary contributions of the JTP architecture are: (1) the specification of interfaces between the knowledge sources; (2) an interlingua based on KIF; and (3) a reasoning framework based on model elimination and iterative deepening. The reasoning framework of model elimination provides a way to handle expressive queries, and iterative deepening guarantees termination. Since the present work was limited to conjunctive queries, these capabilities were of limited use. Use of KIF as an interlingua made the integration principled. Alternative interlingua such as OWL could have been used, but they will quite likely be limited as their expressiveness is still limited, and they are under active development. The specification of reasoner interfaces provided an elegant framework for interfacing knowledge sources into KM. This aspect of the JTP architecture can be adapted by the implementers of other heterogeneous querying systems.

How effective was the use of a shared ontology as a mediated schema? It is difficult to quantify the benefits of using ontology in the interfaces between the Query Manager and various knowledge sources. A well-defined interface obtained by using ontology can save expensive mistakes. For example, mismatch in the use of units was the key source for the failure of the Mars Rover[9]:

The Mars Climate Orbiter (MCO) Investigation Board has determined that the root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, "Small Forces," used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM_FORCES (small forces). A file called Angular Momentum Desaturation (AMD) contained the output data from the SM_FORCES software. The data in the AMD file was required to be in metric units per existing software interface documentation, and the trajectory modelers assumed the data was provided in metric units per the requirements.

We believe that the use of ontology in all the interfaces of the Query Manager make the system principled, and reduce the room for errors. A concrete illustration of this discipline is the ontology mismatch example in Section 3.4. Even though dealing with such mismatches was a substantial implementation effort, it does not necessarily lead to enhanced functionality in the short term.

How effective was the query planner? The quantitative results showed that the performance of Query Planner was promising, and it was able to handle all the queries in the test suite. In its current form, however, it is quite limited. First, it works in a static mode – we cannot dynamically replan during query evaluation. Second, so far it has been limited to only conjunctive queries. Clearly, more work needs to be done on these two fronts.

Is this approach scalable? We believe that for the specific problem we have solved in this paper, the approach is scalable. We do not expect the number of sources to become too large, so a collaborative process of defining the shared ontology and thus minimizing the translation work is feasible. This approach will not scale if the number of sources is large, for example, 20. In those cases, we will need to use an approach as we used for Mediator: it is responsible for integrating a large number of sources over which we have no control, and it acts like a single source for which it is practical for us to collaboratively define the mediated schema.

5. Related Work

The CALO Query Manager is conceptually a hybrid reasoning system that integrates an array of heterogeneous knowledge and information sources. Here, we briefly survey related work in the literature on information integration systems developed by the database community and hybrid reasoning systems developed by the artificial intelligence community, respectively.

Information Integration. The pioneering SDD-1 system was one of the first database systems to address the issue of query evaluation in distributed databases [14]. It proposed an algorithm for optimizing the cost of semijoins needed to transmit intermediate results over a network to a central site that performs the final query evaluation. Although the problem of source capabilities was not of primary concern in [14], more

[9] See http://discovery.larc.nasa.gov/discovery/MCO_report_2.pdf.

recently several systems based on the Mediator Architecture [15] have been developed for integrating heterogeneous information sources, such as TSIMMIS [1], Garlic [2], Information Manifold [3], Infomaster [4], SIMS [5], and HERMES [6].

The various aspects of information integration have been addressed, such as ordering of subgoals to satisfy binding constraints [1, 16], limited query answering capabilities in sources [2], and query planning and rewriting using views [17-20], [3, 5], [2]. In [21] a context-free-grammar-like language was proposed as a foundation for wrapper construction and source capabilities description. The work of [21] was later extended in [2] and more recently further developed in [22] to provide much more expressive power in the language.

Built on previous results in information integration, the CALO Query Manager seeks to integrate heterogeneous information sources as well as reasoning systems. Thanks to the use of CALO Ontology, which serves as the mediated schema for all knowledge sources integrated, the CALO Query Manager avoids the high-complexity problem of query rewriting using views that are inherent to systems like Garlic and Information Manifold. The design was made on purpose even though as a result the CALO Query Manager was less powerful as a very generic information integration system.

The CALO Query Manager has a powerful query planning module that takes into account the limited query-answering capabilities of its knowledge sources. Its domain model is inspired by the work in SIMS. However, apart from most information integration systems, the CALO Query Manager was geared specifically for integrating reasoning systems, which can return unsolved subgoals in addition to ground facts. In such a setting, dynamic query planning is highly desirable. As a result, the design of the CALO Query Manager and its query planning module is very different from traditional database integration systems like TSIMMIS, Garlic, and Information Manifold.

Hybrid Reasoning. Approaches to incorporating external reasoners into first-order deductive systems include procedural attachment [23], theory resolution [13], and resolution with constraints [24]. The JTP resolution might best be viewed as implementing the theory resolution approach, in a model-elimination setting. Procedural attachment can be seen as a special case of theory resolution that considers only one literal at a time. In general, a single theory resolution step can operate on multiple literals and produce a partial proof, or "residue".

These hybrid reasoning frameworks do not specify how the literals should be grouped together, in what order they should be handled, or which reasoner to try first when there is a choice. More generally, our work addresses query optimization problems (such as grouping and ordering of literals) that are likely to appear in any hybrid reasoning system using heterogeneous sources.

Constraint resolution implementations, including Constraint Logic Programming, often include heuristics for choosing the next resolution step that is relevant to ours, including choosing the most constrained literal, or the one with the smallest domain. In [25] procedural attachment is used to incorporate external sources into the SNARK theorem prover. Furthermore, deductive program synthesis methods are used to construct proof plans: source capabilities are represented using axioms, and theorem proving is used to reduce a query into invocations of external sources. There is no explicit planning for how best to evaluate or order the query.

There has been much research in creating general-purpose reusable schemas. The most notable is the Cyc Knowledge Base, which provides thousands of general-purpose concepts [26]. Instead of trying to capture every imaginable distinction, the CALO Ontology approach [9] uses a smaller set of basic building blocks.

6. Conclusion and Future Work

We have presented a novel system for integrating heterogeneous knowledge sources with the help of a reusable ontology, a query planner, and a hybrid reasoning architecture. The main advantages of the approach include semantically well-defined interfaces, efficient execution of queries, and the ability to leverage reasoning for answering queries. We have also presented empirical results of evaluating our system that demonstrated its efficiency.

This work can be extended in several ways. First and foremost, we will make query planning dynamic: as query evaluation proceeds and new subgoals are produced, we must plan for these new subgoals.

Second, we will incorporate statistical information about the information sources in query planning. Third, we will develop a richer model for representing source capabilities, for example, to encode that each predicate may not have the same binding pattern every time it occurs, and that certain groups of predicates should always be bundled together. With these extensions, we will be able to produce more efficient query plans. Finally, we will extend the query manager so that it can work in a multi-CALO environment: when there are multiple CALOs, each CALO has its own query manager. Therefore, one CALO's query manager must be able to ask questions of other query managers, which act like its "peers". Since each CALO has a different user, the user-specific parts of the ontology between multiple CALOs will most likely diverge, and each copy of the query manager should still be able to query its peers even though they share only parts of the ontology.

Acknowledgment

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, or the Department of Interior-National Business Center (DOI-NBC).

References

1. Li, C., *et al. Capability Based Mediation in TSIMMIS*. in *SIGMOD*. 1998.
2. Papakonstantinou, Y., A. Gupta, and L.M. Haas. *Capabilities-Based Query Rewriting in Mediator Systems*. in *International Conference on Parallel and Distributed Information Systems*. 1996.
3. Levy, A.Y., A. Rajaraman, and J.J. Ordille, *Querying Heterogeneous Information Sources Using Source Descriptions*, in *VLDB*. 1996.
4. Genesereth, M.R., A.M. Keller, and O.M. Duschka. *Infomaster: An Information Integration System*. in *SIGMOD*. 1997.
5. Arens, Y., C.A. Knoblock, and W.M. Shen. *Query Reformulation for Dynamic Information Integration*. in *Journal of Intelligent Information Systems*. 1996.
6. Subrahmanian, V.S., *et al.*, *HERMES: A Heterogeneous Reasoning and Mediator System*, . 1997.
7. Clark, P. and B. Porter, *KM -- The Knowledge Machine: Users Manual*, . 1999.
8. Fikes, R., J. Jenkins, and G. Frank. *JTP: A System Architecture and Component Library for Hybrid Reasoning*. in *Proceedings of the Seventh World Multiconference on Systemics, Cybernetics, and Informatics*. 2003. Orlando, FL, USA.
9. Barker, K., B. Porter, and P. Clark, *A Library of Generic Concepts for Composing Knowledge Bases*, in *Proc. 1st Int Conf on Knowledge Capture (K-Cap'01)*. 2001. p. 14--21.
10. Thakkar, S., J.L. Ambite, and C.A. Knoblock. *A Data Integration Approach to Automatically Composing and Optimizing Web Services*. in *ICAPS Workshop on Planning and Scheduling for Web and Grid Services*. 2004.
11. Genesereth, M.R. and R.E. Fikes, *Knowledge Interchange Format, Version 3.0 Reference Manual.* 1992(Logic-92-1).
12. Ullman, J., *Principles of Database and Knowledge Base Systems, Volume 2.* 1989.
13. Stickel, M., *Automated Deduction by Theory Resolution.* Journal of Automated Reasoning, 1985. **4**: p. 333-355.
14. Bernstein, P.A., *et al.*, *Query Processing in a System for Distributed Databases SDD-1.* ACM Transactions on Database Systems (TODS), 1981. **6**(4): p. 602-625.
15. Wiederhold, G., *Mediators in the Architecture of Future Information Systems.* IEEE Computer, 1992. **25**(3): p. 38-49.
16. Morris, K.A. *An Algorithm for Ordering Subgoals in NAIL!* in *PODS*. 1988.
17. Levy, A.Y., *et al. Answering Queries Using Views*. in *PODS*. 1995.
18. Rajaraman, A., Y. Sagiv, and J.D. Ullman. *Answering Queries Using Templates with Binding Patterns*. in *PODS*. 1995.

19. Chaudhuri, S., *et al. Optimizing Queries with Materialized Views*. in *ICDE*. 1995.
20. Qian, X. *Query Folding*. in *ICDE*. 1996.
21. Papakonstantinou, Y., *et al. A Query Translation Scheme for Rapid Implementation of Wrappers},*. in *DOOD*. 1995.
22. Vassalos, V. and Y. Papakonstantinou, *Expressive Capabilities Description Languages and Query Rewriting Algorithms.* Journal of Logic Programming (JLP), 2000. **43**(1): p. 75-122.
23. Myers, K., *Hybrid Reasoning Using Universal Attachment.* Artificial Intelligence, 1994. **67**: p. 329-375.
24. Hans-Jurgen and Burckert, *A Resolution Principle for a Logic with Restricted Quantifiers*. 1991: Springer-Verlag.
25. Waldinger, R., *et al. Question Answering from Multiple Resources*. in *New Directions in Question Answering , AAAI*. 2004.
26. Lenat, D.B. and R.V. Guha, *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project.* 1990: p. 336.