Intelligent Caching in Heterogeneous Reasoning and Mediator Systems *

Sibel Adalı and V.S. Subrahmanian
Department of Computer Science
Institute for Advanced Computer Studies &
Institute for Systems Research
University of Maryland
{sibel, vs}@cs.umd.edu

ABSTRACT

Heterogeneous Reasoning and Mediator Systems (HERMES), first proposed by Lu, Nerode and Subrahmanian, may be used to access and logically integrate information from external databases and/or software packages using functions already defined in those domains. When processing queries in HERMES, we need to execute these external programs, and this is often an expensive process, especially when these programs are located at remote sites on, say a network. Consequently, the problem of minimizing access to such external routines without compromising completeness is an important one. In this paper, we will present methods to cache the results of previous computations and use the knowledge about external programs to make intelligent use of the cached information to reduce calls to external programs.

1 Introduction

During the past few decades, the world has witnessed a spectacular explosion in the *quantity* of data available in one electronic form or another. This vast quantity of data has been gathered, organized, and stored by a small army of individuals, working for different organizations on varied problems in ways that were best suited to accomplish the task in question. To handle this increasing volume of data, Wiederhold [11] proposed the important concept of a *mediator*, which can be regarded as a program for performing semantic integration over multiple data sources. Intuitively, a mediator is a program that accesses and manipulates multiple databases and/or software packages. The user of a mediated system interacts directly with the mediator, which in turn passes along appropriate subqueries to different software packages and/or databases in the mediated system. Extending mediators coded using C-like languages requires understanding the program, and isolating and modifying parts of the program that express how to semantically integrate information, which is a very hard task.

In order to address these concerns, Lu, Nerode and Subrahmanian [8] have developed a formal theoretical framework called *Hybrid Knowledge Bases* that provides a rule-based framework for expressing mediators. Subsequently, Subrahmanian et. al. [10] have developed a system called HERMES (HEterogeneous Reasoning and MEdiator System) that can be used for developing mediators in. HERMES currently runs on both the Sparc/Unix/X platform as well as the PC/Windows platform. HERMES currently integrates the following systems: PARADOX,

*This work was supported by the Army Research Office under Grant Nr. DAAL-03-92-G-0225, by the Air Force Office of Scientific Research under Grant Nr. F49620-93-1-0065, by ARPA Order Nr. A716, by Rome Labs contract F30602-93-C-0241, and by NSF Young Investigator award IRI-93-57756.

DBASE V.0, and INGRES (all commercially available relational DBMSs), a face recognition system, a path planning system developed by the US Army, a multimedia system, a text database system and a spatial database management system. It has been used to develop mediators for robotic applications at the National Institute of Standards and Technology [5], for federal law enforcement applications [9], and for missile siting problems [4]. All these applications involve processing several Gigabytes of widely varying forms of data, using widely varying data representation and reasoning techniques.

The primary aim of this paper is to investigate ways to make the processing of queries in heterogeneous reasoning systems more efficient. We advocate the intelligent use of high-speed caches to avoid computations whenever possible. To accomplish this we introduce the concept of an "invariant", i.e. an expression about the known input/output relationships of a program that can be processed by the mediator. In this paper, we will describe how such caches may be maintained, and how the query processing procedure can make better use of these caches, given the knowledge about different packages, to reduce the complexity of query execution. Our methods are sound and complete, and apply to real-life implemented examples [4, 5, 9].

2 An overview of HERMES

In our framework, a mediator is a ruled based program written in a special syntax and it operates on information coming from different sources of data. The clauses of the mediator specify how this information should be integrated in order to answer a specific query. These sources of data can be as different as relational, deductive, object oriented, text, spatial and temporal databases, problem solving domains such as spread sheets, data stored in different data structures as well as application programs written on top of such programs. Lu, Nerode and Subrahmanian proved that [8] these programs can be viewed as constraint domains and executing these programs can be viewed as solving a constraint in a given domain. We will refer to these domains as external programs.

HERMES syntax is an extension of generalized annotated programs (GAPs) introduced by Kifer and Subrahmanian [6]. Annotated logic is an extension of logic programming where atoms are marked explicitly with values which may be viewed as confidence factors, degrees of belief, or truth values. We will refer to these values as simply truth values. The set of possible truth values form a complete lattice under a given ordering, for example for the real values we can choose the $\leq$ ordering. The atom $\mathtt{A} : \mathtt{0.7}$ can be read as: "the atom `A` is true with certainty of at least `0.7`". The GAP framework has been studied extensively, and it has been shown that it is possible to express uncertainty, temporal information, etc. in this framework.

The lattice used in this paper is that of uncertainty and time points which is defined as the set of all functions $\mathtt{f} : \mathbf{R}^+ \rightarrow [\mathtt{0}, \mathtt{1}]$. In other words, an atom of the form: `temperature(obj,high):[0.7,{1,3}]` can be read as: the temperature of `obj` is `high` with certainty of at least `0.7` at both time points `1` and `3`. Hence, the annotation $[0.7, \{1, 3\}]$ denotes the function `f` where `f(i)=0.7` for `i=1,3` and `f(i)=0` otherwise. The ordering on this lattice is given as follows: $\mathtt{f_1} \preceq \mathtt{f_2}$ iff for all $\mathtt{r} \in \mathbf{R}^+$, $\mathtt{f_1(r)} \leq \mathtt{f_2(r)}$ where $\leq$ is the usual less-than-or-equal-to ordering on the reals.

Adalı and Emery [1] have shown that constraints can be viewed as proving that the output of an external program has certain characteristics when different parameters are input. In other words, we assume that for each external program, we have a different set of functions that return a set of objects, each of which is an answer returned by that function call. Hence, we can link these functions in each program to HERMES, and specify their input and output types in the mediator. For example, in the relational database domain, the $\mathtt{select}_=$ function returns a set of tuples for which the value of a given field is equal to the input value. We introduce special predicates to execute external functions, and access the answers returned by these functions. These special predicates are:

$$\mathtt{in(Ans, Dom : Func(A1, \ldots, Am))}$$

which executes the (implementation of the) function

Func on arguments A1,...,Am on domain Dom. This function will return a set of answers back to the mediator. Ans is instantiated to one of these answers. This predicate will succeed as many times as the number of answers returned by the function Dom:Func. Every time this predicate is reevaluated, Ans will be instantiated to a different answer from the set answers returned by Dom:Func. Here, typically Ans is a complex object with different fields.

$$= (\mathtt{Ans.F1\ldots Fn}, \mathtt{Var})$$

This predicate succeeds whenever the field F1...Fn of Ans is equal to the value of Var, or if Var is a variable, then its value is set to the value of Ans.F1...Fn. Similar conditional predicates are implemented in the mediator such as $<, \geq, >, \leq$ and etc.

Consider the following constraint: "John Smith" is an employee of the company "Widget Co." in the table "Companies" over domain "relation". This can be written in the mediator as:

```
in(Emp,relation:select=(''Companies'',
                 name,''John Smith'')) &
= (Emp.company,''Widget Co.'').
```

Now, we will give an example of a mediatory clause and show how it is executed.

Example 2.1 Suppose, a robot is placed in a room with several objects. The objects are stored in a spatial database with two different index structures: spatial and relational. The spatial index can perform range queries and it returns tuple ids (tids) corresponding to objects in the database. In addition to this, there are some sensors in the system which sense the temperature of the objects. This information is stored in a deductive database and combined using certain rules in the mediator. The temperature data for each object (stored according to the name) contains uncertainty factors and temporal information associated with it. Finally, the location of the robot is stored in the mediator. The layout of the room and the current temperature information is shown in figure 1.

We want to write a clause which will tell the robot to spray coolant on one of the hot objects close to the robot. For this reason, at any given time point (in some time unit used by the robot), we will search all the objects that are inside a circle of radius 3 (units) from the robot. Then, we will check if their temperature is recorded for this time point and if it is greater than 70 degrees with certainty of at least 0.8.

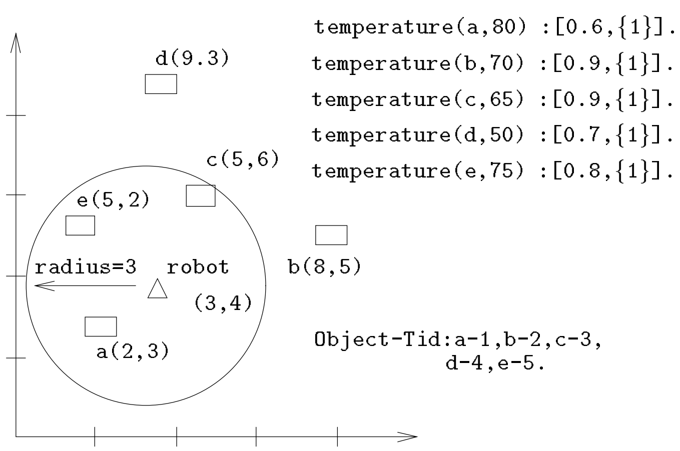

Figure 1: The layout of the room

```
spray_coolant(Xpos,Ypos):[1,{Vt}] ←
     robot(X1,Y1):[1,{Vt}] &
     in(Tid,spatial:range(''Locations'',
                 X1,Y1,3)) &
     in(Object,relational:select=("Objects",
                 tid,Tid)) &
     =(Object.name,Oname) &
     in(true,deductive:query(''Sensors'',
       temperature(Oname,Temp):[0.8,{Vt}])) &
     Temp ≥ 70 & =(Object.xpos,Xpos) &
     =(Object.ypos,Ypos).
```

Suppose we ask the query:
← spray_coolant(X,Y):[1,{1}]. Part of the processing of this query is shown in figure 2 using resolution as the inference rule. The resolution rule used in this example is given in [8]. Note that the queue for the outputs of functions relational:select= and deductive:query are not shown in the figure since they contain only one item. The atom spray_coolant(Xpos,Ypos):[1,{1}-{1}] and the constraint $1 \geq 1$ are removed since {1}-{1}=$\emptyset$ and $1 \geq 1$ is true.

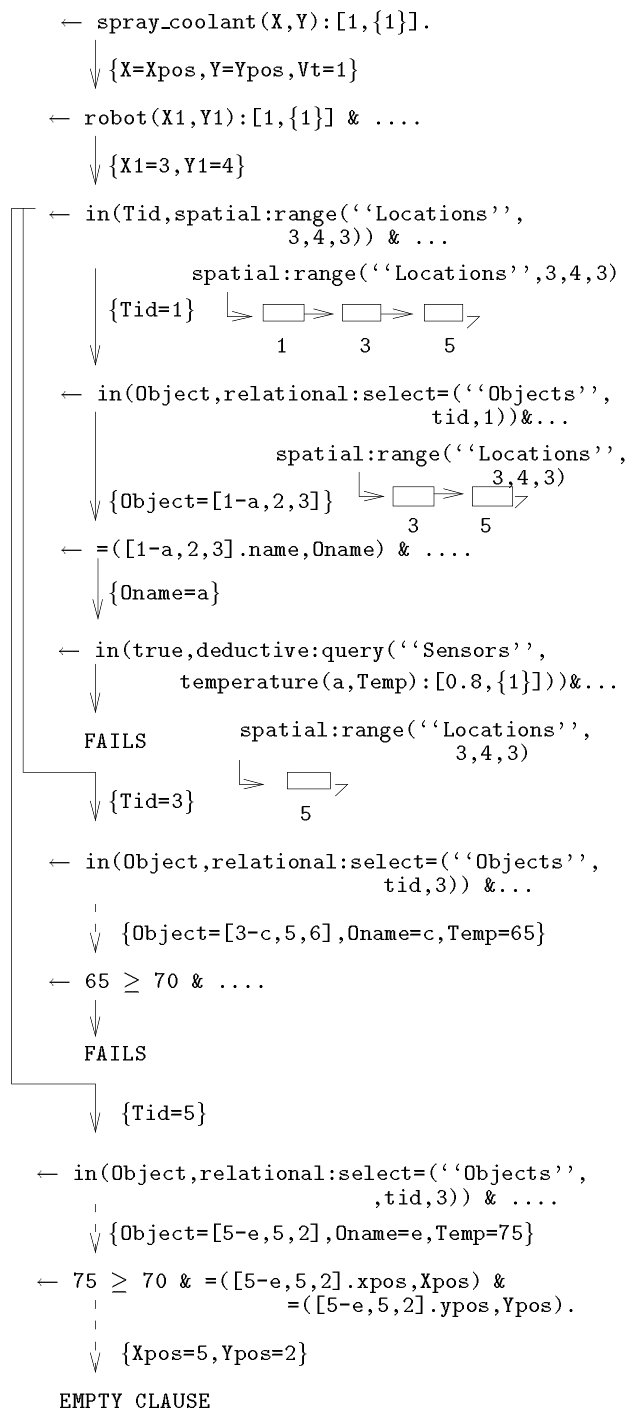

Figure 2: The processing of query ← `spray_coolant(X,Y):[1,{1}]` in the mediator.

2.1 Processing of queries in the mediator

As explained above, external programs are accessed by executing functions whose input and output types are specified in the mediator. These functions are referred to as external functions to indicate that their execution takes place outside the mediator (though the invocation of the external routines is from inside the mediator). Below, we give the general query processing procedure for mediators.

Definition 2.1 (Query Processing in the Mediator (QPM)) Let M be a mediator, Q_0 an initial query. A *deduction* from Q_0 in the mediator is a sequence of distinct queries $Q_0, \ldots, Q_m$ such that Q_i is obtained from a query Q_j $(j < i)$ in the sequence in one of the following ways:

1. Q_i is obtained from Q_j by solving an annotated atom `A` using an appropriate resolution rule. (Many such resolution rules have been defined in the literature, e.g. Kifer and Subrahmanian [6], Adalı and Subrahmanian [3] and Lu, Murray and Rosenthal [7]. Which of these is used does not matter, as long as the one picked is sound and complete.)
2. Q_i is obtained from Q_j by solving an `in(Ans, d:f(Args))` predicate, hence invoking the function `d:f`. In this case, if $\mathtt{Ans}\vartheta$ is a solution to the instance $\mathtt{d:f(Args}\vartheta)$ of the original function (hence making $(\mathtt{in(Ans,d:f(Args))})\vartheta$ true), then $Q_i = (Q_j - \mathtt{in(Ans,d:f(Args))})\vartheta$ where $Q_j - \mathtt{in(Ans,d:f(Args))}$ is the query obtained by deleting the given `in` predicate from Q_j.
3. Q_i is obtained from Q_j by solving an equality predicate `=(A,B)` or similar conditional predicates for processing complex objects. In this case, if θ is the substitution that makes $\mathtt{=(A,B)}\theta$ true, then $Q_i = (Q_j - \mathtt{=(A,B)})\theta$.

A "*refutation*" from an initial query Q_0 is a deduction $Q_0, \ldots, Q_m$ of queries produced by the mediatory query processor where Q_m is the empty clause. Completeness in mediated systems is defined a bit differently than the normal logic programs. The definition is given below:

Definition 2.2 (Completeness) Suppose M is a mediator that possesses the fixpoint reachability property and $\leftarrow Q$ is a query. Let M_d be an instance of M where $M_d \models Q$ and all the predicates in M_d of the form: `in(Ans,d:f(Args))` have the property that `d:f(Args)` is evaluable and terminates. Then, there is a refutation of $\leftarrow Q$ from M_d.

Intuitively, a mediator M is said to possess the fixpoint reachability property (cf. Kifer and Subrahmanian [6]) iff whenever there is a infinite refutation for a query Q in M (i.e. involving an infinite sequence of queries produced by the mediatory query processor), then it is possible to find a finite refutation for Q in M. The fixpoint reachability property is critical for completeness because otherwise, we need to take recourse to infinitary proofs. *In [8], Lu, Nerode and Subrahmanian have proved that the above strategy is sound and complete, as long as the resolution rule used in step 1 of the above definition is sound and complete for annotated logic.*

The above definition of query processing shows how special predicates are treated when solving queries. However, the interface between external function calls and the query processor needs further explanation. We assume that there is an interface program for every external program (not necessarily different) that makes the actual calls, stores the output and fetches (different) answers from the output whenever they're needed by the mediator. Whenever the predicate `in(Ans,d:f(Args))` needs to be solved, the query processor sends a request to this interface to fetch an answer. The function `d:f(Args)` is executed only once for every query Q_i in the query processing sequence, i.e. the first time the above predicate is chosen to be solved in Q_i. It is the interface's responsibility to fetch a new answer stored for this call and return it to the query processor. Similarly, whenever a refutation is found for a query the interface should terminate all the programs that are still running.

3 Invariants

It is possible that in the processing of the rules in the mediator, "similar" function calls to external programs will need to be executed several times since the same kind of information may be requested over and over by different users. Backtracking is another reason for such a situation. Calling an external program is usually a costly operation because of the memory, CPU requirements and possible network delays [2]. Furthermore, actual packages may levy charges for accessing them. Suppose there is a way to guess "some" of the answers that will be returned by an external call. If a refutation is found by substituting one of these answers, then there is no need to execute the external domain call. This is accomplished by caching the answers returned by previous external calls and re-using them when needed. Similarly, if there is a way of knowing that a function is not defined for some inputs, whenever it is called for these inputs, we can terminate the search down a path of the search space.

The challenge of this approach lies in deciding which sets of answers are relevant, and in representing the input/output behavior of some external functions. This information is stored in the system with the help of some explicit rules which will be referred to as "invariants". Invariants are expressions specifying the relation between the set of answers returned by an external call, its arguments and other possible external calls.

As an example, consider the following invariant :

$$\texttt{T2} \geq \texttt{T1} \;\Rightarrow\; \texttt{f(T2)} \supseteq \texttt{f(T1)}$$

The above expression can be read as follows: if $\texttt{T2} \geq \texttt{T1}$, then all the solutions of `f(T1)` are also solutions of `f(T2)`. Hence, if the set of answers for `f(T1)` were previously stored in a cache, then these answers can be re-used whenever the function `f(T2)` is called; if none of the answers to `f(T1)` satisfies the rest of the query then `f(T2)` needs to be computed. An example of such an invariant is given below:

Example 3.1 Suppose `relation` is a constant in the mediator which refers to a relational database with the usual selection operators. For example,

$\texttt{select}_{\leq}(\texttt{R},\texttt{F},\texttt{V})$ selects all the tuples in table R such that the value of the field F is less than or equal to V. Then, the following are possible invariants for different select functions.

$$\texttt{T2} \leq \texttt{T1} \Rightarrow \texttt{relation}:\texttt{select}_{\leq}(\texttt{R},\texttt{Field},\texttt{T1}) \supseteq \texttt{relation}:\texttt{select}_{\leq}(\texttt{R},\texttt{Field},\texttt{T2}).$$

$$\texttt{T2} \geq \texttt{T1} \Rightarrow \texttt{relation}:\texttt{select}_{\geq}(\texttt{R},\texttt{Field},\texttt{T2}) \supseteq \texttt{relation}:\texttt{select}_{\geq}(\texttt{R},\texttt{Field},\texttt{T1}).$$

The first invariant can be read as: For any given database R and field Field in the domain relation, whenever $\texttt{T2} \geq \texttt{T1}$ is satisfied, all the tuples that are in $\texttt{relation}:\texttt{select}_{\geq}(\texttt{R},\texttt{Field},\texttt{T2})$, are also in $\texttt{relation}:\texttt{select}_{\geq}(\texttt{R},\texttt{Field},\texttt{T1})$.

Example 3.2 Suppose the domain spatial is a spatial data structure such as a point quadtree storing points in two-dimensional space. The function vertical_slice(File,X,Dist) in this domain returns all the points that have X-coordinates between X+Dist and X-Dist, in other words all the points that are in the vertical slice taken from X-Dist to X+Dist. The following is an invariant about this function:

$$\texttt{Dist1} \leq \texttt{Dist2} \Rightarrow \texttt{spatial}:\texttt{vertical\_slice}(\texttt{File},\texttt{X},\texttt{Dist2}) \supseteq \texttt{spatial}:\texttt{vertical\_slice}(\texttt{File},\texttt{X},\texttt{Dist1}).$$

which states that whenever the X-coordinate is fixed, the points in a vertical slice are contained in any of the bigger vertical slices. We can easily write similar invariants for other spatial functions. The invariant for the horizontal_slice function is the same as vertical_slice. As for the range(X,Y,Dist) function which returns all the points that are at distance Dist from point (X,Y) (i.e. all points (X1,Y1) such that $(\texttt{X}-\texttt{X1})^2+(\texttt{Y}-\texttt{Y1})^2 \leq \texttt{Dist}^2$ we can write the following invariants:

$$\texttt{Dist1} \leq \texttt{Dist2} \Rightarrow \texttt{spatial}:\texttt{range}(\texttt{File},\texttt{X},\texttt{Y},\texttt{Dist2}) \supseteq \texttt{spatial}:\texttt{range}(\texttt{File},\texttt{X},\texttt{Y},\texttt{Dist1}).$$

$$|\texttt{X1}-\texttt{X2}| \leq |\texttt{Dist1}-\texttt{Dist2}| \Rightarrow \texttt{spatial}:\texttt{range}(\texttt{File},\texttt{X2},\texttt{Y},\texttt{Dist2}) \supseteq \texttt{spatial}:\texttt{range}(\texttt{File},\texttt{X1},\texttt{Y},\texttt{Dist1}).$$

Example 3.3 Suppose text is a text database containing news articles. The function heading(DB,Word) returns all the articles from the database DB that contain Word in their heading. Suppose also that the function Dict(Word) returns true when Word is in the local dictionary. (This function can be a call to a spelling check program.) Then, we can write the following invariant for the heading function:

$$\texttt{Dict}(\texttt{Word}) = \texttt{false} \Rightarrow \texttt{text}:\texttt{heading}(\texttt{DB},\texttt{Word}) = \emptyset.$$

The above invariant enforces a spell-check before a more costly call is executed. Especially, if a special dictionary of keywords is available, searching in this dictionary before the whole text database is searched will avoid time consuming calls.

Finally, it is possible to have invariants with empty premises. These can be viewed as hard-coded information about the external program. Below we give such an example.

Example 3.4 Suppose the spatial data structure contains a function called rectangle which finds all the points within a rectangle specified by the X,Y coordinates of two opposite corners. Then, we have the following invariant:

$$\Rightarrow \texttt{spatial:rectangle(File,X1,Y1,X2,Y2)} = \texttt{spatial:vertical\_slice(File,X1,X2)} \cap \texttt{spatial:horizontal\_slice(File,Y1,Y2)}.$$

Now, we're ready to define some of the concepts used in the previous examples. We will first define what an external call is , then give the format of the invariants.

Definition 3.1 (External Call) Given a domain d and an n-ary ($\texttt{n} \geq 0$) function f in domain d, the expression d:f(A1,...,An) is called an *external call* iff for all i, $1 \leq \texttt{i} \leq \texttt{n}$, Ai is either a variable, an object of the same type as the the ith argument of f as specified in the mediator with some uniform specification language.

Suppose d:f(A1,...,An) is an external call. An answer pair to this external call is a pair $(\texttt{Ans},(\texttt{d}:\texttt{f}(\texttt{A1},\ldots,\texttt{An}))\vartheta)$ such that Ans is in the set returned by executing the function f in domain (program) d on arguments $A1\vartheta, \ldots, An\vartheta$. The *type* of this external call is the type of Ans. The *object space* of this external call is the set of all possible objects of the same type as Ans.

Definition 3.2 (Invariant) An invariant is an expression of the form either

$$A_1 \,\&\, \ldots \,\&\, A_m \;\Rightarrow\; E_0 \supseteq s(E_1, \ldots, E_n)$$

or,

$$A_1 \,\&\, \ldots \,\&\, A_m \;\Rightarrow\; E_0 = s(E_1, \ldots, E_n)$$

where i) E_0 is an external call (that is being evaluated as part of the query), ii) for all $1 \leq i \leq n$, E_i is either an external call or a set of objects, and if *Obj* is the object space of E_i's, then s is a function from Obj^n to *Obj*, and $A_k, 1 \leq k \leq m$ are atoms in the mediator. All the variables in the invariant are considered to be universally quantified. In the above invariant, the expression $A_1 \,\&\, \ldots \,\&\, A_m$ will be called the *premise*. The external call E_0 will be referred to as the *defined external call*, and the function $s(E_1, \ldots, E_n)$ is called the *substitute answer set* for E_0.

An example of the s function found in the conclusion of an invariant is the 0-ary function which returns the constant $\emptyset$ as given in example 3.3. Other examples of the s function are the following functions: the set intersection function s(X,Y)=X ∩ Y as given example 3.4, and the identity function s(X)=X as given in example 3.2. The s functions can also be more complex, they may contain conditional operations and different set operators.

3.1 Query Processing with Invariant-Based Answer Caching

The query processing procedure discussed in section 2.1 uses only the program clauses, an inference rule and calls to external programs to process queries. In many cases, external calls made during query processing may bear a close relationship to external calls made previously to the same external program. Invariants were introduced to reflect the possible relationships between the previous calls to an external function and the current call. In this section, we will discuss how invariants can be used to avoid making costly calls to external programs. First, we will define a cache table that will store the answers obtained from an external call. Then, we will define how these cached answers are used in conjunction with invariants to speed up query execution.

Definition 3.3 (External Call Cache) is a table which stores pairs of the form $(call, answers)$ where *call* is an external call of the form domain:function(A1,...,An) and *answers* is the complete set of answers returned by this call.

As an example, an external call to the selection function in a relational database will contain a set of tuples as the answers.

Definition 3.4 (Substitute Answer Pair) Suppose domain:function(A1,...,An) is an external call and

$$I \equiv\; P \;\Rightarrow\; E_0 \;\mathcal{R}\; s(E_1, \ldots, E_m)$$

is an invariant associated with the mediatorΦΦed system M where $\mathcal{R} \in \{\supseteq, =\}$. Let ϑ be the most general substitution such that $M \models P\vartheta$ and

$$P\vartheta \;\Rightarrow\; E_0\vartheta \;\mathcal{R}\; s(E_1\vartheta, \ldots, E_m\vartheta)$$

is an instance of I where

- $E_0\vartheta$ is an instance of domain:function(A1,.. ..,An),
- for all $E_i\vartheta (1 \leq i \leq m)$, Ans_i is the union of all set of answers in the cache for external calls E'_i that are instances of $E_i\vartheta$. If there are no such entries, then $Ans_i = \emptyset$. ($\mathcal{R}$ is changed internally to $\subseteq$ iff any one of $E_i\vartheta$ is not in the cache.)

Then $(Ans\vartheta, E_0\vartheta)$ is a *substitute answer pair* for the external call domain:function(A1,...,An) according to invariant I and the cache where $Ans\vartheta = s(Ans_1, \ldots, Ans_m)$

Intuitively, the following definition can be interpreted as follows: Whenever the set of answers to an external call domain:function(A1,...,An) needs to be determined, choose an invariant such that domain:function(A1,...,An) is the defined external call (i.e. the first component of the consequent of an invariant). Moreover, suppose the appropriate instance of the premise is true in the mediated system, and the substitute answer set contains external calls which are cached. Then, the substitute answer set will replace the "real" call to `domain`. Note that the processing of invariants doesn't change according to the relation between defined call and its substitute (i.e. "$\supseteq$" vs. "$=$".) However, it will make a difference in the implementation, since the relation "$\supseteq$" states that different answers may be obtained the real external call and in such a case the real call should be made if none of the substitute answers leads to a refutation.

Example 3.5 Recall the query processing example shown in figure 2. The external call `spatial :range (''Locations'',3,4,3)` was executed, and the set of tuple-ids $\{1, 3, 5\}$ was obtained. Suppose this is cached in the table as the pair: (`spatial:range(''Locations'',3,4,3)`,$\{1, 3, 6\}$). Now, recall the following invariant given for the `range` function:

$$I \equiv \mathtt{Dist1} \leq \mathtt{Dist2} \Rightarrow \mathtt{spatial:range(File,X,Y,Dist2)} \supseteq \mathtt{spatial:range(File,X,Y,Dist1)}.$$

Now, suppose we are executing the external call `spatial:range(''Locations'',3,4,6)`. Substitute answer pairs for this call according to the above invariant are obtained as follows:

- Let $\vartheta = \{\mathtt{X} = 3, \mathtt{Y} = 4, \mathtt{Dist1} = 3, \mathtt{Dist2} = 6,$ `File=''Locations''`$\}$.
- Clearly the premise of the invariant is satisfied, i.e. $3 \leq 6$ is true in the mediator.
- The cache contains the set of answers for the call `spatial:range(''Locations'',3,4,3)`.
- Then, the substitute answer set of the above invariant for substitution ϑ contains the answers `1,3,5`. Then, all the following are substitute answer pairs according to the given invariant and the cache: `(1,spatial:range(''Locations'',3,4,6))`, `(3,spatial:range(''Locations'',3,4,6))`, `(5,spatial:range(''Locations'',3,4,6))`.

Note that, if the call was executed on the spatial domain, object `b` will also be found to be in the 6 units range to the robot. Hence its id `4` should be in the answer queue to this call. This is expected, since the relation in the invariant was partial (given by $\supseteq$).

Substitute answer pairs can be used to access cached answers for external calls as well. To do this we can write the invariant $\Rightarrow \mathtt{d:t(Args)} = \mathtt{d:t(Args)}$ for all the external calls `d:t(Args)` that are cached in the mediator. Hence, all the answers stored for an instance of an external call `d:t(Args)` will be used to obtain substitute answer pairs. Now, we're ready to define a query processing procedure which will make use of the cache and the invariants.

Definition 3.5 (Query Processing with Invariants and Answer Caching (QPIAC)) Suppose M is a mediator, M_I is a set of invariants and Q_0 is a query. Then, a query computation of Q_0 with invariant-based answer caching is a sequence of pairs:

$$\langle Q_0, \Sigma_0 \rangle, \ldots, \langle Q_p, \Sigma_p \rangle.$$

where $\Sigma_0, \ldots, \Sigma_p$ are distinct external call caches, and $Q_0, \ldots, Q_p$ are distinct queries such that $\ldots Q_{p+1}, \Sigma_{p+1} \rangle$ is obtained either using rules 1 and 2 in definition 2.1, or the following rule:

3. Suppose Q_{i+1} is obtained from Q_j for some $j \leq i$, by solving an `in(Ans,Xcall)` predicate. In this case, there are three cases to consider:

(a) (Cache Use) Suppose there exists an entry ($\mathtt{Xcall}\vartheta, \mathtt{Answers}$) in the cache such that $\mathtt{Ans}\vartheta \in \mathtt{Answers}$. Then, $Q_{i+1} = (Q_j - \mathtt{in(Ans, Xcall)})\vartheta$ and $\Sigma_{p+1} = \Sigma_p$.

(b) (Use of Invariants) There exists a substitute answer pair ($\mathtt{Ans}\vartheta, \mathtt{Xcall}\vartheta$) for `Xcall` according to some invariant I in M_I and the cache Σ_p: Then, $Q_{i+1} = (Q_j - \mathtt{in(Ans, Xcall)})\vartheta$ and $\Sigma_{p+1} = \Sigma_p$.

(c) (External Function Call and Cache Update) Let ($\mathtt{Ans}\vartheta, \mathtt{Xcall}\vartheta$) be an answer pair for the (real) external call (i.e. make the actual external call and get one of the answers returned by it.) Then, $Q_{i+1} = (Q_j - \mathtt{in(Ans, Xcall)})\vartheta$ and the new answer pair (($\mathtt{Ans}\vartheta, \mathtt{Xcall}\vartheta$)) is inserted into Σ_p to obtain Σ_{p+1}.

As can be seen from the above definition, in this new query processing procedure the cache is used both directly (step 3(a)) and indirectly (step 3(b)) with the help of invariants to answer a query. In the new procedure all results of previous function calls are cached. It is possible to make use of a function which determines which external functions are cached, and the invariants and cache will be used only for those functions. Furthermore, QPIAC doesn't specify in which order steps 3(a,b,c) must be executed. In practice, the cache will be consulted first (step 3(a)), then the invariants will be evaluated against the cache (step 3(b)), and if both of these fail, then the actual external call will be executed (step 3(c).) To be able to prove the soundness and completeness of the QPIAC query processing procedure we must make sure that the invariants are correct statements about their respective domains. In other words, a substitute answer is guaranteed to be an actual answer pair.

Definition 3.6 (Well-formed Invariants) An invariant I is said to be well-formed iff for all possible caches Σ, and mediators M with the same domain as I, if `(Ans,Xcall)` is a substitute answer pair according to invariant I, mediator M, and cache Σ, then it is also an answer pair (i.e. `Ans` is an answer for the external call `Xcall`.)

Below, we prove that mediatory query processing procedure and QPIAC are equivalent.

Theorem 3.1 (Soundness and Completeness of QPIAC) Suppose M is a mediator and M_I is a set of invariants all of which are well-formed. Then, Q_0 has a refutation (i.e. without answer caches and/or invariants) iff Q_0 has a QPIAC-refutation (i.e. with answer caching and invariants). □

In particular, this means that as long as a sound and complete query processing procedure for annotated logic is used, the strategy of caching answers and using invariants to optimize external calls is sound and complete when used in conjunction with the annotated logic query processing procedure.

3.2 Cache and Invariant Management

In this section, we will define a new external program called "*Cache and Invariant Manager* (`CIM`)". This program will store the cache, process the invariants and mediate between the external call interfaces in the mediator and the external programs. We will then show that the original query processing procedure can simulate QPIAC using the cache and invariant manager (`CIM`) as an additional external program. As a side effect, we will show that it is possible to cache only the answers obtained for a set of domains and/or a set of selected functions for each domain.

Definition 3.7 (Answer Pairs returned by `CIM`) Suppose `CIM` is the cache and invariant manager, M_I is a set of invariants processed by the `CIM` and `d:f(Args)` is an external call cached in `CIM`. Then, $(\mathtt{Ans}\vartheta, \mathtt{d:f(Args)}\vartheta)$ is an answer pair for `CIM` iff one of the following is true:

- $\mathtt{Ans}\vartheta$ is stored in the cache as an answer for $\mathtt{d:f(Args)}\vartheta$, or
- $(\mathtt{Ans}\vartheta, \mathtt{d:f(Args)}\vartheta)$ is a substitute answer pair for `d:f(Args)` according to an invariant $I \in M_I$ and the cache, or
- $(\mathtt{Ans}\vartheta, \mathtt{d:f(Args)}\vartheta)$ is an answer (of the real external call) `d:f(Args)`.

Next, we will give a transformation procedure which takes a mediator and a set of invariants as its input and produces another mediator with the extra external program `CIM`.

Definition 3.8 (`CIM` Transformation) Suppose M is a mediator, M_I is a set of invariants. Then, the `CIM` transformed program M' is obtained as follows:

1. Add `CIM` as an extra external program to the mediator.
2. For each external function `f` called for program `d`, do the following: Create a new function `d|f` for program `CIM` and copy the input and output types of `d:f` to `CIM : d|f`.
3. Change each external call `d:f(Args)` in the mediator to the call `CIM : d|f(Args)`.

4. Store all the invariants in M_I in `CIM`.

As a result of the transformation, whenever an external call `d:f(Args)` needs to be executed in the mediator, the call $\texttt{CIM} : \texttt{d}|\texttt{f}(\texttt{Args})$ will be initiated. `CIM` will use the cache and the invariants to find the answers to an external call. Below, we define a new query processing procedure (QPCIM) which uses the QPM query processing algorithm together with the cache and invariant manager as a new domain.

Definition 3.9 (Query Processing with Cache and Invariant Manager (QPCIM)) Suppose M is a mediator and M_I is a set of invariants. If M' is the `CIM` transformation of M and M_I then QPCIM uses the mediatory query processing function (QPM, see section 2.1) on M' to execute queries.

Theorem 3.2 (Soundness and Completeness of QPCIM)) Suppose M is a mediator and M_I is a set of invariants. Then a query Q_0 has a QPCIM-refutation iff Q_0 has a QPIAC-refutation. □

3.2.1 Comparison of QPIAC and QPCIM Query Processing Procedures

Although QPIAC and QPCIM compute the same answers, they provide two different solutions to the same problem with their respective advantages and disadvantages. QPCIM has the advantage of providing for many different special purpose caches while the query processor remains general purpose. On the other hand, since the query processor has no control over how the caches are used and the external calls are made, some unnecessary calls cannot be avoided. Although, QPIAC eliminates these disadvantages of QPCIM, it leads to the modification of the query processor as well as the specification language. Moreover, changing caching services provided by the query processor becomes harder and harder.

In the next section, we introduce a new concept, viz. invariant products. We show how invariants can be combined to produce new and more specific invariants. We will then introduce a new query processing function which makes use of invariant products as well as the invariants.

3.3 Invariant Products

Suppose, the mediator wants to avoid making external function calls as much as possible. Hence, whenever an "`in`" predicate needs to be executed, the mediator will check if there is some other function call stored in the cache which can be substituted for this call. Invariants are used for this purpose. It is possible to process invariants to obtain other possible ways of constructing the answer to an external function call. The following example illustrates this informal intuition.

Example 3.6 Recall the following invariants given for the spatial data structures:

$$\Rightarrow \texttt{spatial:rectangle(File,X1,Y1,X2,Y2)} = \texttt{spatial:vertical\_slice(File,X1,X2)} \cap \texttt{spatial:horizontal\_slice(File,Y1,Y2)}.$$

$$\texttt{Dist1} \leq \texttt{Dist2} \Rightarrow \texttt{spatial} : \texttt{vertical\_slice}(\texttt{File1},\texttt{X},\texttt{Dist2}) \supseteq \texttt{spatial} : \texttt{vertical\_slice}(\texttt{File1},\texttt{X},\texttt{Dist1}).$$

Now, we have two inequalities in the conclusions of the above invariants of the form $\texttt{A} = \texttt{B}_1$ and $\texttt{B}_2 \supseteq \texttt{C}$. If $\texttt{B}_1 = \texttt{B}_2$ then, from these two, we can conclude $\texttt{A} \supseteq \texttt{C}$. Hence, we will perform a special type of inference which we will call "Invariant Production". Notice that $\texttt{B}_1\vartheta = \texttt{B}_2\vartheta$ where $\vartheta = \{\texttt{File1} = \texttt{File}, \texttt{X} = \texttt{X1}, \texttt{Dist2} = \texttt{X2}\}$. Hence, the result of this production (which will be called the "invariant product" of the two invariants) is given as:

$$\texttt{Dist1} \leq \texttt{X2} \Rightarrow \texttt{spatial:rectangle(File,X1,Y1,X2,Y2)} \supseteq \texttt{spatial} : \texttt{vertical\_slice}(\texttt{File},\texttt{X1},\texttt{Dist1}) \cap \texttt{spatial:horizontal\_slice(File,Y1,Y2)}.$$

The advantage of having this new invariant is that, given an external call `spatial:rectangle(File,X1, Y1,X2,Y2)`, it may not be possible to find a cached call to `vertical_slice` with `X1`, `X2` as arguments. But, it may possible to find calls that are cached in the table for smaller values of `X2`. Similar invariants exist for `horizontal_slice` and for different values of `X1` and `Y1`. Note that it may not be possible to compare two external calls beforehand as in the above example.

As an example, we cannot compare the external function calls d:f(a) with d:f(b) to obtain a product as above without knowing the output of these two functions. Since the cache provides this opportunity, we may be able to combine two invariants in a radically different way. Below, we will give such an example of invariant production which makes use of the cache.

Example 3.7 Recall the following two invariants which were presented earlier in the paper (variables are renamed to standardize the invariants apart).

$\mathtt{Dist11} \leq \mathtt{Dist21} \Rightarrow$
$\quad \mathtt{spatial:vertical\_slice(File11,X1,Dist21)} \supseteq$
$\quad \mathtt{spatial:vertical\_slice(File11,X1,Dist11)}.$
$\mathtt{Dist12} \leq \mathtt{Dist22} \Rightarrow$
$\quad \mathtt{spatial:range(File2,X2,Y2,Dist22)} \supseteq$
$\quad \mathtt{spatial:range(File2,X2,Y2,Dist12)}.$

Now, suppose the cache contains the following pairs: (spatial:vertical_slice(''qtree'',2,3), {a,b,c,d,e}),
(spatial:range(''qtree'',3,1,5),
{a,b,d}). Then, since $\{a,b,d\} \subseteq \{a,b,c,d,e\}$, we can write the following invariant product from the two invariants above by using the cached information:

$3 \leq \mathtt{Dist21}\ \&\ \mathtt{Dist12} \leq 5 \Rightarrow$
$\quad \mathtt{spatial:vertical\_slice(``qtree'',2,Dist21)} \supseteq$
$\quad \mathtt{spatial:range(``qtree'',3,1,Dist12)}.$

which indicates that as long as we shrunk the circle of the range query and enlarge the area of the vertical slice, the relation stated in the invariant will hold. In this way we related two different functions of the same type that had no connection to each other according to the invariants stated for them by using the dynamic cache.

Definition 3.10 (Invariant Production) Suppose I_1 and I_2 given below are invariants in the mediator and Σ is an external call cache:

$$I_1 \equiv \mathtt{P_1} \Rightarrow \mathtt{E1_0}\ \mathtt{R_1}\ \mathtt{s_1(E1_1, \ldots, E1_m)}.$$
$$I_2 \equiv \mathtt{P_2} \Rightarrow \mathtt{E2_0}\ \mathtt{R_2}\ \mathtt{s_2(E2_1, \ldots, E2_n)}.$$

where $\mathtt{R_1}, \mathtt{R_2} \in \{=, \supseteq\}$, $\mathtt{E1_0}, \mathtt{E2_0}$ are external calls, and for all $1 \leq i \leq m$, and all $1 \leq j \leq n$, $\mathtt{E1_i}$ and $\mathtt{E2_j}$'s are external calls or expressions denoting sets of objects, and $\mathtt{P_1}, \mathtt{P_2}$ are conjunctions of atoms in the mediatory language.

1. Suppose, there exists a substitution ϑ such that $\mathtt{E2_0}\vartheta = \mathtt{E1_p}\vartheta$ for some $1 \leq p \leq m$. Then, the following expression is an "invariant product" of I_1 with I_2 according to cache Σ:
$$(\mathtt{P_1}\ \&\ \mathtt{P_2})\vartheta \Rightarrow \mathtt{E1_0}\vartheta\ (\mathtt{R_1} \diamond \mathtt{R_2})\ \mathtt{s_1(E1_1, \ldots, E1_{p-1},}$$
$$\mathtt{s_2(E2_1, \ldots, E2_n), E1_{p+1}, \ldots, E1_m)}\vartheta.$$
2. Suppose, there exist substitutions ϑ_1, ϑ_2 such that there exist entries $(\mathtt{E2_0}\vartheta_1, \mathtt{Ans2_0})$ and $(\mathtt{E1_p}\vartheta_2, \mathtt{Ans1_p})$ in the cache and that $\mathtt{Ans1_p} \supseteq \mathtt{Ans2_0}$[1], then the following expression is an "invariant product" of I_1 with I_2 according to cache Σ:
$$\mathtt{P_1}\vartheta_1\ \&\ \mathtt{P_2}\vartheta_2 \Rightarrow \mathtt{E1_0}\vartheta_1\ (\mathtt{R_1} \diamond \mathtt{R_2})$$
$$\mathtt{s_1(E1_1}\vartheta_1, \ldots, \mathtt{E1_{p-1}}\vartheta_1, \mathtt{s_2(E2_1, \ldots}$$
$$\mathtt{\ldots, E2_n)}\vartheta_2, \mathtt{E1_{p+1}}\vartheta_1, \ldots, \mathtt{E1_m}\vartheta_1).$$

The operation $\diamond$ is defined as:

$\diamond$	$=$	$\supseteq$
$=$	$=$	$\supseteq$
$\supseteq$	$\supseteq$	$\supseteq$

We now define the set of invariants that can be generated using the cache and the invariant production rule.

Definition 3.11 Suppose M_I is a set of invariants and Σ is an external call cache. Then, the set of all possible invariant products for M_I and Σ denoted by $IP(M_I, \Sigma)$ is the set of all invariants I where either (1) $I \in M_I$, or (2) I is an invariant product of I_1 and I_2 according to cache Σ and $I_1, I_2 \in IP(M_I, \Sigma)$.

Suppose s is a function that takes sets of objects as its arguments and produces a set as output. Then, s is said to be *monotonic in its arguments* for all $1 \leq i \leq m$ $\mathtt{E_i} \subseteq \mathtt{E'_i}$ implies that $\mathtt{s(E_1, \ldots, E_i, \ldots, E_m)} \subseteq \mathtt{s(E_1, \ldots, E'_i, \ldots, E_m)}$. For example, if the function s is computed by any combination of union and intersections ($\cup, \cap$) only, then s is monotonic in its arguments.

Theorem 3.3 Suppose I_1 and I_2 are invariants where

$$I_1 \equiv \mathtt{P_1} \Rightarrow \mathtt{E1_0}\ \mathtt{R_1}\ \mathtt{s_1(E1_1, \ldots, E1_m)}.$$

[1] Note that for this to hold, both $\mathtt{E1_p}$ and $\mathtt{E2_0}$ must denote objects of the same type

If $\mathtt{s_1}$ is monotonic in its arguments then the invariant product of I_1 with I_2 is well-formed whenever I_1 and I_2 are both well-formed. □

Now, we are ready to define a query processing procedure which makes use of the invariant products as well as the invariants to re-use the information stored in the cache.

Definition 3.12 (Query Processing Invariant Production (QPIP)) Suppose M is a mediator, M_I is a set of invariants and Q_0 is a query. Then, a query computation of Q_0 with invariant production is a sequence of pairs:

$$\langle Q_0, \Sigma_0 \rangle, \ldots, \langle Q_p, \Sigma_p \rangle.$$

$\Sigma_0, \ldots, \Sigma_p$ are distinct external call caches, $Q_0, \ldots, Q_p$ are distinct queries such that $\langle Q_{p+1}, \Sigma_{p+1} \rangle$ is obtained in the same way as in the QPIAC query processing procedure as given in definition 3.5 with the exception that condition 3(b) ("There exists a substitute answer pair $(\mathtt{Ans}\vartheta, \mathtt{Xcall}\vartheta)$ for `Xcall` according to some invariant in M_I.") is changed to:

"There exists a substitute answer pair$(\mathtt{Ans}\vartheta, \mathtt{Xcall}\vartheta)$ for `Xcall` according to some invariant in $IP(M_I, \Sigma_p)$."

Theorem 3.4 (Soundness and Completeness of QPIP) Suppose M is a mediator and M_I is a set of invariants all of which are well-formed. Then, Q_0 has a QPIAC-refutation iff Q_0 has a QPIP-refutation. □

4 Conclusions

The HERMES ("HEterogeneous Reasoning and MEdiator System") system provides a uniform way of expressing how information from existing programs can be semantically integrated to answer complicated questions that none of the component programs can on its own. Since uniform statistics/estimates of how long a query will take in a given program may not be readily available, choosing an appropriate execution plan for a mediator poses an important problem. In this paper, we have shown that query processors of the HERMES system can take advantage of caching techniques as well as the semantic information about each program to intelligently plan the execution of queries. Moreover, we have examined different methods with varying "intelligence" and compared the trade-offs of these methods.

References

[1] S. Adalı and R. Emery. (1994) *A Uniform Framework For Integrating Knowledge In Heterogeneous Knowledge Systems*, To appear in Proc. of the Eleventh International Conference on Data Engineering. (http://www.cs.umd.edu/projects/hermes/publications/postscripts/cons.ps)

[2] S. Adalı and V.S. Subrahmanian. (1994) Amalgamating Knowledge Bases, II: Distributed Mediators, accepted for publication in: Intl. Journal of Intelligent Cooperative Information Systems. (ftp://ftp.cs.umd.edu/pub/papers/TRs/3124.ps.Z)

[3] S. Adalı and V.S. Subrahmanian. (1994) Amalgamating Knowledge Bases, III: Algorithms, data structures and query processing. Technical Report CS-TR-3124, Computer Science Department, University of Maryland, Aug 1993. Submitted for publication. (http://www.cs.umd.edu/projects/hermes/ publications/postscripts/akbiii.ps)

[4] J. Benton and V.S. Subrahmanian. (1993) *Using Hybrid Knowledge Bases for Missile Siting Problems*, Proc. 1994 Conf. on Artificial Intelligence Applications, IEEE Computer Society.

[5] J. Horst, E. Kent, H. Rifky and V.S. Subrahmanian. (1994) *Hybrid Knowledge Bases for Real-Time Robotic Reasoning*, Proc. IVth Intl. Workshop on Pattern Recognition in Practice (eds. E. Gelsema and L. Kanal), N. Holland/Elsevier.

[6] M. Kifer and V.S. Subrahmanian. (1989) *Theory of Generalized Annotated Logic Programming and its Applications*, Journal of Logic Programming, 12, 4, pps 335–368, 1992. Preliminary version in: Proc. 1989 North American Conf. on Logic Programming, MIT Press. (http://www.cs.umd.edu/projects/hermes/ publications/postscripts/galp.ps)

[7] J. Lu, N. Murray and E. Rosenthal. (1993) *Signed Formulas and Annotated Logics*, draft manuscript. Preliminary version in: Proceedings of the International Symposium on Multiple-Valued Logic, IEEE Computer Society Press, 1993, 48-53.

[8] J. Lu, A. Nerode and V.S. Subrahmanian. (1993) *Hybrid Knowledge Bases*, Accepted for publication in: IEEE Trans. on Knowledge and Data Engineering. (http://www.cs.umd.edu/projects/hermes/ publications/postscripts/hkb.ps)

[9] V.S. Subrahmanian. (1994) *Hybrid Knowledge Bases for Integrating Symbolic, Numeric and Image Data*, Proc. 1994 SPIE Conf. on Applied Imagery and Pattern Recognition.

[10] V.S. Subrahmanian, S. Adalı, A. Brink, R. Emery, J. Lu, A. Rajput, T.J. Rogers, R. Ross. (1994) HERMES: A Heterogeneous Reasoning and Mediator System, submitted for publication. (http://www.cs.umd.edu/projects/hermes/ overview/paper)

[11] G. Wiederhold. (1992) *Mediators in the Architecture of Future Information Systems*, IEEE Computer, March 1992, pps 38–49.

[12] G. Wiederhold. (1993) *Intelligent Integration of Information*, Proc. 1993 ACM SIGMOD Conf. on Management of Data, pps 434–437.