HERMES: A Heterogeneous Reasoning and Mediator System *

V.S. Subrahmanian[1]
Sibel Adalı[1] **Anne Brink**[2] **Ross Emery**[1]
James J. Lu[3] **Adil Rajput**[1]
Timothy J. Rogers[1] **Robert Ross**[1] **Charles Ward**[1]

Abstract

Hybrid Knowledge Bases (HKB), due to Nerode and Subrahmanian, is a paradigm for integrating information from diverse sources. It forms a framework within which mediators, proposed by Wiederhold, may be formalized. HERMES is a system based on the HKB theory that has been under development over the past two years. It provides a general, declarative language for defining a mediator, which expresses semantic integration of information from diverse data sources and reasoning systems. The compiler for this language contains a facility that enables the incremental integration of new systems into an existing mediator system. Thus mediators are extensible. This paper describes the design and implementation of HERMES. Techniques for extending an existing mediator are described, illustrating how certain parts can be automated, while others may be interactively presented to a system integrator for his/her input. The system currently runs on Sun Sparc stations (under Unix), as well as on the IBM-PC platform under DOS/Windows 3.1.

*This work was supported by the Army Research Office under Grant Nr. DAAL-03-92-G-0225 and by the Air Force Office of Scientific Research under Grant Nr. F49620-93-1-0065, and by ARPA/Rome Labs contract F30602-93-C-0241 (ARPA Order Nr. A716). Address Correspondence to V.S.Subrahmanian

[1]Department of Computer Science & Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742. {vs,sibel,emery,adil,robross,rogers,ward}@cs.umd.edu

[2]Modeling and Simulation Division, U.S. Army Topographic and Engineering Center, Alexandria, VA 22315–3864. anneb@piglet.tec.army.mil

[3]Department of Computer Science, Bucknell University, Lewisburg, PA 17837. lu@sol.cs.bucknell.edu.
Supported in part by the National Science Foundation under Grant CCR9225037.

1 Introduction

The past few decades have 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. Consequently, the different bodies of data have been organized in ways that best suited the task in question.

In light of the ever increasing volume of data, Wiederhold [24, 25] has proposed the important concept of a *mediator*, which can be regarded as a framework for performing semantic integration over multiple data sources and reasoning systems.

Integrating Heterogeneous Data Sources: Certain obvious advantages exist in being able to *integrate* information across multiple data sources. Clearly, such an integration alleviates the burden of duplicating the data gathering efforts. More importantly however, is that it enables the extraction of information that would otherwise be impossible. As examples, law enforcement agencies such as Interpol would benefit from having the ability to access databases of various national police forces, to assist their efforts in fighting international terrorism, drug trafficking, and other criminal activities. Insurance companies, using data from external sources including other insurance companies and police records, can identify possible fraudulent claims. Medical researchers and epidemiologists, having access to records across geographical and ethnic boundaries, are in a better position to predict the progression of certain diseases. In each case, the information extracted from the diverse sources would not have been possible had the data sources been viewed in isolation.

Integrating Heterogeneous Reasoning Paradigms: In conjunction with the ability to integrate a variety of data sources is the need to integrate diverse forms of reasoning. Access to such reasoning systems provides mediators with sophisticated abilities to extract and produce new information from existing data. As a simple example, consider the problem of *terrain reasoning* [3] used to analyze the terrain of a given region, which can be used to optimize the deployment of certain resources (e.g. defensive weapons systems, first aid centers, etc.) over the region. This region may be part of a "rogue" nation developing nuclear or chemical weapons. It may also be the terrain of a planet such as Mars or Venus. The resources deployed may be defensive weapons or scientific observation equipment. Determining where these resources can be physically situated requires access to multiple terrain databases containing information about soil structure, vegetation maps, elevation maps, and road maps. In addition, the deployment strategy may require reasoning over certain properties that involve *numerical optimization* of an objective function, and the actual movement of the resources from their current locations to their destinations may use another reasoning method that incorporates *route planning*. Hence the availability of two reasoning techniques in this case is critical in the successful extraction of a deployment plan over several data sources. More generally, solutions to complex problems require integrating multiple forms of reasoning that may include logical inference, numerical optimization, planning, pattern recognition, scheduling, and learning.

Our Contributions: The aim of the system described in this paper is to take the first steps towards the development of a principled methodology for integrating multiple data sources and reasoning systems, and to propose a mediator language within which access to the data sources and reasoning systems can be expressed uniformly. There are two important aspects to constructing a mediator: *domain integration* and *semantic integration*. Intuitively, domain integration is the physical linking of the data sources and reasoning systems, while semantic integration is the coherent extraction and combination of the information provided by the data and reasoning sources, serving a given purpose.

1. **Domain Integration** is the task of adding a new source of data or reasoning system to an existing mediated system (or one being developed) so that resources provided by the new system, whether it is new data, or new representations of data, or a corpus of new reasoning algorithms, may be accessed by various mediators.

   Previous approaches to software integration have generally used relatively ad-hoc coding methods. Extending such a system with new software presents formidable challenges for various reasons. The process through which a new domain may be integrated into our mediated system on the other hand, follows a simple sequence of steps. Indeed, certain of these steps can even be automated, and a set of integration toolkits have been designed for further simplifying the integration process.[4] One reason for our ability to integrate new softwares with relative ease is due to the clean separation of the domain integration phase from the semantic integration phase described below.

2. **Semantic Integration** is the process of specifying methods to resolve conflicts, pool information together, and define new, compositional operations based on existing operations in the individual data sources. Numerous forms of *conflicts* can be identified, and methods will be described for resolution in each case. Of course, these methods are only there to assist the author of the mediator. Often there may exist no unique way to resolve conflicts. The system architecture contains a *conflict handling toolkit* to detect certain types of potential conflicts, and to suggest possible resolutions. The integrator is not required to adhere to any existing resolution strategy, and may instead specify his/her own strategies. The system has the capability to then store the new strategy for future use. There is also an *information pooling toolkit* that contains a number of predefined methods for semantically integrating two data sources. The mediator author may select certain of these options when implementing a mediator. Again, a new method may alternatively be specified and the system possesses the option to incorporate the method into its repertoire.

Each of domain integration and semantic integration entails conceptually very different sets of activities. In the former case, the knowledge to perform the required set of activities includes system dependent information, such as details needed to access the pre-existing functions of a specific database/software package. In the latter case, the implementor must have a clear understanding of the application requirements of the resulting mediated system. Examples include law enforcement or terrain reasoning. The primary purpose of the implementor – called the *mediator author* – in this case is to build a system that satisfies the application requirements. Clearly, the mediator author should be able to perform this task from the logical perspectives

[4] One of us (Robert Ross) integrated Paradox into the existing mediated system in under 60 hours using the above methodology.

of the application, without having to concern him/herself with the details of how information is extracted from each of the data and reasoning sources. It is this observation that prompts us, in our system, to clearly delineate the domain integration phase from the semantic integration phase. This is in sharp contrast to existing mediated systems, where there is typically no clear logical separation between the code that performs the mediation, and the code that accesses the data sources required to carry out the mediation. From a software engineering perspective, our approach gains the important advantages of modularity and abstraction, since packages are incorporated into the larger system independently of specific mediation tasks. To the actual mediating code, domains/packages are treated as abstract entities whose internal details are hidden, and mediating code may be written without being concerned about how different domains can be, or have been, integrated.

The language that we use for writing semantic integration code is rule-based, and hence is declarative. For instance, suppose an integration is to be performed between a database $DB1$, stored in Borland's Paradox DBMS, and a database $DB2$, stored in DBase V. The language offers mechanisms to extract the data in $DB1$ and $DB2$ in a systematic way. The data can then be *logically* combined in a semantically meaningful way. Note that the semantics of the combined data may contain conflicts, which can be resolved naturally through our language. In addition, the combined data may be used to define *new* derived operations based on one or more compositions of the operations in the original databases $DB1$ and $DB2$. These new operations can subsequently be used to specify additional new operations. A *query processing engine* for the language is under development.

Our system contains a *yellow pages facility* to assist the mediator author in locating data sources. As more domains are incorporated into our system, this facility will prevent the mediator author from having to manually navigate through the potentially complex maze of data sources. Interestingly, the yellow pages server is not all that different from other domains being integrated. Indeed, they may be regarded as domains as well. We will show how the mediator language may access and query yellow-page servers, and how the databases indicated by these yellow pages servers may then be queried for the desired information.

It is possible to couple *actions* to the results of query processing. Thus, the result of executing a given query, and obtaining an answer $\mathcal{A}$ may lead, based on an analysis of the answer $\mathcal{A}$, to an action. Such an action may for instance be an email message being sent, or an update transaction being executed to a remote database. In our system, it is possible for such actions to be executed automatically after a query has been processed.

The principle result of our work is a piece of software that can currently be seen at the University of Maryland. An initial release is expected by anonymous ftp in the Spring of 1995. The system conforms to the software architecture shown in Figure 1, which provides a pictorial sketch of the ideas discussed in this paper. At present, two versions of the system exist, one on the IBM-PC platform running under DOS/Windows 3.1, and one on the SUN/Unix platform running under XWindows. The PC version currently integrates the following.

- relational data in multiple heterogeneous formats in ASCII text files,
- relational databases developed under Borland's Paradox DBMS,
- relational databases developed under DBase V DBMS,

- spatial data in the form of PR-Quadtrees,
- Raw text data (as in the case of USA Today electronic newswires),
- pictorial data (in the form of .GIF files)

The SUN/UNIX version integrates the following.

- relational data in multiple heterogeneous formats in ASCII text files,
- relational databases developed under INGRES,
- object-oriented databases developed using ObjectStore (in progress)
- a path planning package developed by the US Army for optimal path planning across "free" (i.e. mountainside, countryside, etc.) terrain,
- spatial data in the form of PR-Quadtrees,
- Raw text data (as in the case of USA Today electronic newswires)
- pictorial data (in the form of .GIF files files)

The system, even in its current experimental form, has been applied to the two tasks below. Other applications are being researched and developed.

- *US Army:* The system is being applied by the US Army for developing a terrain reasoning system. This system integrates a *path planning software* developed at the US Army's Topographic and Engineering Center, with various relational and spatial databases describing aspects of the region covered by the terrain maps. Additional details can be found in [3].
- *NIST:* The National Institute of Standards and Technology is developing an application of our system for reasoning in mobile robotics. Here, the integration involves spatial data represented at multiple levels of resolution (occupancy grids to quadtrees) as well as relational data. Details may be found in [7].

2 The HERMES System

The high-level architecture of the the Maryland Heterogeneous Reasoning and Mediator System (HERMES) is first described, after which the key components will be examined in more detail.

2.1 System Architecture

Figure 1 is the software architecture of the HERMES system. As emphasized in the introduction, a prime motivation behind the design of HERMES is to modularize the activities involved in creating a mediator. In particular, central to HERMES is the logical separation of the construction of the mediation code, and the integration of new domains. These were referred to, respectively, in the previous section, as semantic integration and domain integration. In Figure 1, the domains that are integrated are represented by the box in the lower right-hand

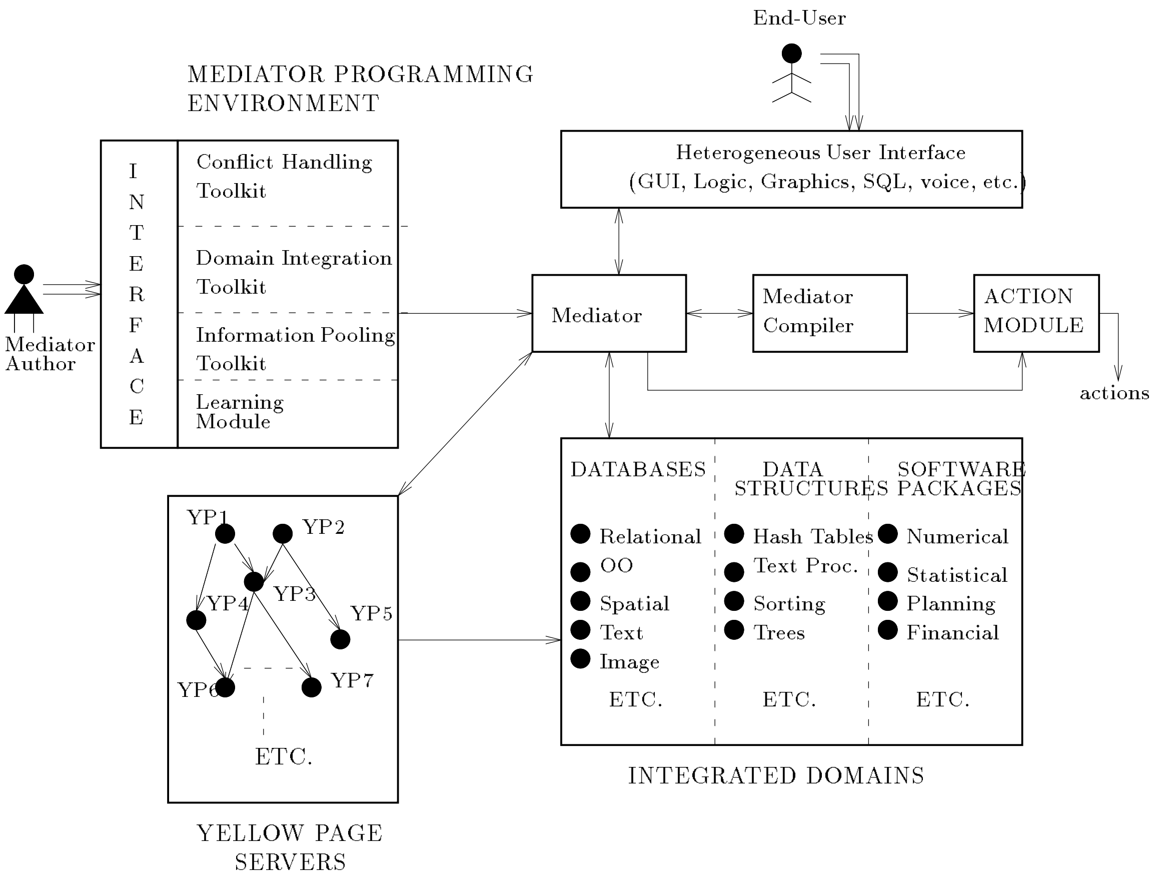

Figure 1: Software Architecture for the HERMES System

corner. Each of the items in the box represents a data source or a reasoning system that mediation code written inside HERMES can access. Each of these software packages constitutes a domain.

As the figure indicates, the system is accessed by two types of individuals: the mediator author, and the end-user. The interaction of the mediator author with HERMES is through the mediator programming environment, while the end-user accesses existing mediators through a variety of user-interfaces. The mediator programming environment offers a set of toolkits to assist the mediator author in both the domain, and the semantic integration phase.

2.1.1 The Mediator Author

Broadly speaking, the mediator author designs and implements the code that integrates data across data sources and reasoning systems. The logical division that we have taken between semantic and domain integration implies that a mediator author may, at one time, perform only domain integration, or only semantic integration, but not necessarily both.

Existing mediating software is typically written in C-like languages. In HERMES, we have chosen a high-level, declarative, logic based language for constructing mediation code. The theory of this language has been investigated in considerable detail in [15, 22, 1]. The syntax of this language will be introduced in Section 2.2.

HERMES provides a set of tools to assist the mediator author in the construction of mediators. Some of the tools are useful for domain integration, while others are used for semantic integration. These tools together with the logical language forms the mediator programming environment depicted in Figure 1. We briefly discuss the ideas behind the tools.

- (Domain Integration) This is the process of adding another data source or reasoning system to HERMES, so that mediation code may access the information/software contained within. A *domain integration toolkit* that partly automates this integration has been designed for this purpose, and will be described later.

  In general, access to a software package involves a few, relatively common steps. These same steps need to be taken whether the software is accessed manually, or automatically. As an example, to manually execute a query in Sybase requires first invoking Sybase from a Unix prompt, followed by retrieving the relevant Sybase databases from within Sybase, and lastly issuing a query using the SQL query language. When the answer is received, there may exist an option for storing the answer for future use. The final operation is to close Sybase. The automation of these steps is non-trivial as it involves communication between multiple processes on a workstation. These processes include: the calling process, namely HERMES; Sybase, the called process; and any subprocesses that Sybase itself may spawn. The situation becomes considerably more complex when these subprocesses in turn spawn other new processes.

- (Conflict Resolution in Semantic Integration) Often, conflicts may exist between information reported from difference data sources. The ability to identify and resolve conflicts in a simple and natural way is an important component of the HERMES mediating language. The idea is based partly on the recent work in annotated logics [9, 8].

  On the other hand, there exists a collection of commonly used conflict resolution strategies, each corresponding to a frequently occuring conflict. These strategies have been gathered in the *conflict resolving toolkit* in HERMES. The mediator author may access the strategies through the toolkit, which can also automatically generate appropriate mediating rules based on strategies selected.

- (Information Pooling in Semantic Integration) As we have discussed in the introduction, there may be multiple ways of coalescing information from multiple data sources to obtain new information that could not be inferred from a single database, by itself. We have developed an *information pooling* toolkit that will present the mediator author with a number of options on how to pool information together. If the mediator author selects one or more of these approaches, then this toolkit will automatically generate mediating rules to be placed in the mediator.

2.1.2 End-Users

The mediator language underlying HERMES is a logical, rule-based language. Consequently, a natural choice for the end-user query language is also a *logical* query language, similar to HERMES'. In addition, a host of other query languages is provided in HERMES that together, make up the *heterogeneous* user interface. The interface supports conjunctive queries whose parts may be expressed in different languages, including the logical mediator language, and an English-like language. A mix of the logic and English language may also be possible. Work is currently under way to augment the interfact with voice, as well as graphical queries. More discussion on the user interface will be given in Section 2.6.

2.2 The Mediator Language

The HERMES mediator language can extract information from different domains – data sources and reasoning systems. An underlying assumption is that each of the domains internally provides a set of operations through which the functionalities of the domain are accessed. Hence, a *domain* $\mathcal{D}$ may be viewed as an abstraction of databases and software packages, and is made up of three components: a set of values, Σ; a set $\mathcal{F}$ of functions on Σ; and a set of relations on the data objects in Σ. The elements of Σ may be thought of as the data objects that are being manipulated by the software in question. For example in a numerical computation software package, Σ, contains the real numbers. In general, values in Σ may be typed, so that Σ is composed of a collection of different universes of values. The functions in $\mathcal{F}$ take objects in Σ as input, and return as output, objects from their range. These functions may be thought of as the pre-defined functions existing in the package/domain $\mathcal{D}$ that we are seeking to integrate into a mediated system. An example is the function to perform numerical integration in the aforementioned software package. Lastly, the relations are functions over Σ whose output value is either true or false.

Given a domain $\mathcal{D}$, a *domain call* is a syntactic expression

$$\texttt{domainname} : \langle\texttt{domainfunction}\rangle(\langle\texttt{arg1}, \ldots, \texttt{argn}\rangle)$$

where $\langle\texttt{domainfunction}\rangle$ is the name of the function in $\mathcal{F}$, and $\langle\texttt{arg1}, \ldots, \texttt{argn}\rangle$ are the arguments that it takes. The informal reading of the call is: in the domain `domainname`, execute the function `domainfunction` defined therein on the arguments $(\texttt{arg1}, \ldots, \texttt{argn})$.

Before considering the finer details of the mediation language, we first present several examples of domains to ground our discussion in more concrete terms.

2.2.1 The Domain of Relational DBMSs

Consider the `PARADOX` database management system. The values in this domain consists of the collection of all tables, as well as the individual values that may be stored in the tables. We denote this domain by $\mathcal{D}_{\texttt{PARADOX}}$. The functions $\mathcal{F}$ over $\mathcal{D}_{\texttt{PARADOX}}$ include the usual database operations `project`, `select` and `join`. Each of these operations take tables as input, and produces new tables as output. Other operations, for example aggregates, typically accept tables and attribute names as input, and produce real values as their output. Below are some specific examples.

The expression

$$\texttt{PARADOX} : \texttt{project}('\texttt{parts}', "\texttt{partid}")$$

invokes the execution of the `project` operation, projecting out the `partid` field of the relation called `parts`. Expressing selects is somewhat more complex. To each boolean condition C, we have a corresponding $\texttt{select}_C$ such that C is imposed on the value of an attribute, specified respectively by the third and the second arguments. Thus, to select all tuples in the `parts` relation in `PARADOX` with a cost of over 50, the appropriate domain call may be expressed:

$$\texttt{PARADOX} : \texttt{select}_{>}('\texttt{parts}', "\texttt{cost}", 50).$$

Operations may be composed. Therefore, getting a list of `partid`s of of parts that cost over 50 may be expressed through the domain call

$$\texttt{PARADOX} : \texttt{project}(\texttt{select}_{>}('\texttt{parts}', "\texttt{cost}", 50), "\texttt{partid}").$$

Joins may be similarly expressed. For instance, the statement

$$\texttt{PARADOX} : \texttt{join}('\texttt{parts1}', '\texttt{parts2}', '\texttt{partname}', '\texttt{partnom}')$$

joins the relations `parts1` and `parts2` on the fields `partname` and `partnom`, respectively. Finally, an example domain call to an aggregate may be

$$\texttt{PARADOX} : \texttt{sum}(\texttt{project}(\texttt{select}_{=}('\texttt{parts}', "\texttt{color}", "\texttt{green}), "\texttt{cost}")),$$

which finds the total cost of all parts that are green.

This syntax has been used uniformly in HERMES to access any relational database management system, including DBASE, INGRES, and PARADOX.

2.2.2 The Domain of Spatial DBMSs

In a spatial database, the set of values Σ consists of the collection of all coordinates and the values that may be contained in an ordinary, say relational database. The usual set of operations include `range`, which can be used to find all points within a specified distance of a given location, and horizontal and vertical slice queries, that can be used to find points in which one of the axis is within some specified distance of a given location. Some example domain calls are as follows.

1. $\texttt{spatial} : \texttt{range}('\texttt{map}', \texttt{given}, \texttt{distance})$

   is a range query on a spatial database called '`map`' where `given` is the given pair of (x, y) coordinates.

2. $\texttt{spatial} : \texttt{vertslice}('\texttt{map}', \texttt{given}, \texttt{distance})$

   is a vertical slice query on 'map'. If $\texttt{given} = (\texttt{x}, \texttt{y})$ then this call returns the set of all points (x', y') represented in the `'map'` database such that $||\texttt{x} - \texttt{x}'|| \leq \texttt{distance}$.

Note that the above syntax completely abstracts away the details of internal representation in the databases. Though this is a useful and important software engineering technique, we will see later that the activities of domain integration cannot be done in complete independence from these internal details. However, in many cases, the tools that are provided in the domain integration toolkit may make these tasks less painstaking (cf. Section 2.4.1).

2.2.3 A Return to the Mediator Language

We use the domains discussed above to elaborate on the mediator language. The language is rule-based, with Prolog-like syntax. Access to the various domains integrated in HERMES is achieved through a small, but fairly general special set of predicates. These predicates take as input, various domain calls, whose syntax was introduced in the previous subsections.

1. (=) This is the usual binary equality relation that takes two arguments. It succeeds in the case that both of its arguments are identical. Note that either of the arguments may be a domain call, in which case their output will be compared. The = predicate may take complex types (e.g. records, arrays, array of records, etc.) as input.

2. (`in`) This is a binary predicate symbol that takes two arguments as input. The second argument is a domain call, whose output is assumed to be a set of values. The first argument may be either a variable, or is a constant. In the former case, the relation always succeeds, with the side-effect that the variable is substituted by one of the values from the domain call. In the latter case, the relation `in` holds just in the case that the constant represented by the first argument is one of the elements in the set returned by the domain call.

   In the Paradox example, the query

   $$\texttt{in(P, PARADOX : project('parts', color))} \,\&\, = (P.color, \text{"}green''))$$

   succeeds if the `parts` database contains at least one object whose color is green.

3. (`is`) This is a binary predicate symbol that takes two arguments, both sets. The relation `is` succeeds if the first argument is either equal to, or can be instantiated to something that is equal to the second argument.

   For instance, the atom

   $$\texttt{is(\{"green"\}, PARADOX : project('parts', color))}$$

   succeeds just in case the `parts` database contains only green objects.

The formal syntax of a mediator has now been explained. We refer to each relation constructed using one of =, `in`, or `is` as a *domain call atom*. An *annotation* is a pair $[\mu, \tau]$ where μ is an expression representing a value between 0 and 1 inclusive, and τ is an expression representing a set of non-negative real numbers. Given an atom A, the expression $A : [\mu, \tau]$ is called an *annotated atom*, where A is the *atomic part*. A *mediatory clause* (or *mediatory rule*) is a statement of the form

$$A_0 : [\mu_0, \tau_0] \quad \leftarrow \quad A_1 \,\&\, \ldots \,\&\, A_n.$$

where each A_i, for $i = 1, ..., n$, is either an annotated atom, or is a domain call atom. The first annotated atom $A_0 : [\mu_0, \tau_0]$ is called the head of the rule, while the conjunction to the right of the symbol $\leftarrow$ is called the body.

For those readers familiar with annotated logics, the annotations extend ordinary Prolog clauses with the reasoning capability over uncertainty and time. Informally, to assert an annotated atom $A : [\mu, \tau]$ is to say intuitively that the relation A is true with certainty at least μ at all time points in τ. Thus, the annotated atom $\mathtt{at}(\text{"}\mathtt{john}\text{"}, \text{"}\mathtt{office}\text{"}) : [\mathtt{0.9}, [\mathtt{0800}, \mathtt{1700}]]$ says that between the time points 0800-1700 hours, there is a certainty of over 90% that John is in his office. The reading of the mediatory clause above is then: Suppose for each annotated atom $A_i : [\mu_i, \tau_i]$ in the body, A_i is true with certainty at least μ_i at all time points in τ_i. Suppose in addition that each domain call atom holds in the body. Then conclude that A_0 is true with certainty at least μ_0 at all time points in τ_0. The formal semantics of these annotations has been studied in [9, 15].

We remark that the language for expressing uncertainty in annotations contains as constants, the real numbers between 0 and 1. The language also contains variables and functions interpreted over the interval $[0, 1]$. Likewise, the second component of each annotation is a term in a first order language for expressing temporal information. The constants in the language are *sets* of real numbers between 0 and 1. The intersection of the temporal language, the uncertainty language, and the logical language need not be empty. This way, a variable V belonging to all three languages may be used to integrate temporal information with uncertainty information, logical information, and information gleaned from different databases.

Example 1 Suppose we are given data stored in a Paradox database called $DB1$, a DBASE database called $DB2$ and a spatial database called $DB3$. Suppose $DB1$ is a relation containing fields "qty" and "name", $DB2$ is a relation containing fields "name" and "location", and $DB3$ is a spatial data structure who nodes have a field "location", in which two subfields, "x" and "y", are specified. Using the following rule, the query $query1(Supplier, Part, Quantity, Factory)$ retrieves any $Supplier$ that lies within 50 units of distance from a given $Factory$, and the $Supplier$ has enough of the component $Part$ to satisfy $Factory$'s request for a given $Quantity$ of the component.

$$
\begin{array}{l}
query1(Supplier, Part, Quantity, Factory) : [1, \mathbf{R}] \leftarrow \\
\quad in(Supplier, \mathtt{PARADOX} : \mathtt{project}(\mathtt{select}_{>}('db1', "qty", Quantity), "name")) \,\& \\
\quad in(Loc1, \mathtt{DBASE} : \mathtt{project}(\mathtt{select}_{=}('db2', "name", Supplier), "location)) \,\& \\
\quad in(Loc2, \mathtt{DBASE} : \mathtt{project}(\mathtt{select}_{=}('db2', "name", Factory), "location)) \,\& \\
\quad in(Loc1, \mathtt{SPATIAL} : \mathtt{RANGE}('db3', Loc2.x, Loc2.y, 50)).
\end{array}
$$

According to the rule, once the $Part$ and $Quantity$ being desired by a particular $Factory$ are specified, any supplier is either satisfactory or unsatisfactory. However, in some cases, it may be desired to evaluate the "goodness" of how well a supplier matches the needs of the Factory based on the distance of the supplier from the $Factory$, and the available extra stock that the supplier has. The following modified rule accomplishes this through the annotation variables.

Example 2 Let $query2$ be defined similarly to $query1$, augmented with a goodness of fit between a supplier and the factory he supplies by evaluating the quantity of overstock the supplier has, and the distance from the factory of the supplier. The evaluation is performed by the annotation function `EVAL`.

$$
\begin{array}{l}
query2(Supplier, Part, Quantity, Factory) : [\mathtt{EVAL}(Dist, Over), \mathbf{R}] \leftarrow \\
\quad in(Supplier, \mathtt{PARADOX} : \mathtt{project}(\mathtt{select}_{>}('db1', "qty", Quantity), ["name", "qty"])) \,\& \\
\quad in(Loc1, \mathtt{DBASE} : \mathtt{project}(\mathtt{select}_{=}('db2', "name", Supplier), "location)) \,\& \\
\quad in(Loc2, \mathtt{DBASE} : \mathtt{project}(\mathtt{select}_{=}('db2', "name", Factory), "location)) \,\&
\end{array}
$$

$$in(Dist, \texttt{SPATIAL} : \texttt{DIST}('db3', Loc2.x, Loc2.y, Loc1.x, Loc2.x))\ \&$$
$$= (Over, SQ.qty - Quantity).$$

Here, $Dist$ and $Over$ are both annotation variables, as well as variables in the logical language. The complex annotation term $\texttt{EVAL}(Dist, Over)$ appearing in the annotation of the head of the clause is assume to return a value between 0 and 1.

We present a few more examples of domains that are currently integrated in HERMES.

2.2.4 The Domain of Text Databases

Text database systems can be used to index large amounts of text data. A text database may be regarded as a domain whose set of values consists of characters and words.

A simple example of a rules involving text database is as follows.

$$news(Word, Article) : [1, \mathbf{R}] \leftarrow in(P, \texttt{TEXTDB} : headline('usatoday.idx', Word))\ \&$$
$$= (P.filename, Article).$$

This rule defines a predicate that accesses a text database called `textdb` through the *headline* function, which has been implemented in `textdb`. It takes two arguments: '*usatoday.idx*' is the index file used to index a body of text data (actually, in our implementation, this indexes a body of data from on-line versions of the USA Today newspaper), and $Word$ is the keyword on which to search. A query to this rule through which a user may find, for example, the *name of the spouse of the person whose taxes are reported in USA today* is:

$$\leftarrow news(\text{"}taxes\text{"}, Article)\ \&\ news(Person, Article)\ \&$$
$$in(Spouse, \texttt{PARADOX} : \texttt{project}(\texttt{select}_{=}('spouse', Spouse1, Person), Spouse).$$

This query assumes that the information on spouses is kept in a relational Paradox table called 'spouse'.

2.2.5 The Domain of Pictorial Databases

Pictorial databases are repositories of images. Suppose a module existed for querying these databases to determine features present in a picture. A general architecture for such queries and a formal theoretical framework for it has been studied by Marcus and Subrahmanian in [16]. For instance, consider the predicate $p(Person, Rank, Picture)$ defined below which succeeds just in case $Picture$ is one of George Bush, with the spouse of a person whose tax dealings have been reported in the newspaper. This may be expressed as follows:

$$p(Person, Rank, Picture) : [1, \mathbf{R}] \leftarrow$$
$$in(\text{"}GeorgeBush\text{"}, \texttt{PICTUREDB} : feature(File))\ \&$$
$$in(OtherPerson, \texttt{PICTUREDB} : feature(File))\ \&$$
$$in(Spouse, \texttt{PARADOX} : \texttt{project}(\texttt{select}_{=}('spouse', Spouse1, OtherPerson), Spouse))\ \&$$
$$news(Spouse, Article) : [1, \mathbf{R}]\ \&$$
$$news(\text{"}taxes\text{"}, Article).$$

Note that the mediator author need not be concerned with how the pre-defined function called *feature* is implemented within the pictorial database – a variety of implementation possibilities exist including by an image processing program, or a face recognition program, or it may have been created by annotating the pictorial data by a human. The above example has been implemented in HERMES using the last approach. In addition, we have incorporated a face recognition system developed at the Vision Lab of the University of Maryland (we are currently testing how well this face recognition algorithm works).

2.3 The Mediator Compiler

In order for HERMES to execute queries expressed in the mediator language described in Section 2.2, several auxiliary facilities are necessary. The first is a *catalog* used to maintain information, including argument types, about the predefined functions in the domains that have been integrated. The second is a *standard format* used by the mediator compiler to convert data returned by domain functions into "internal" formats.

2.3.1 The Domain Catalog

The *catalog* contains information about the functions that make up the interface to a domain. This information is used to facilitate compilation, and to give indications on how output from domain function calls should be interpreted. As an example, suppose a mediator accesses the `SPATIAL` database package. The catalog for the `SPATIAL` database package is shown in Figure 2. The catalog specifies information concerning 10 functions that collectively make up the interface of the `SPATIAL` database to the external world. The first line in each function entry defines the name of the function. The second line enumerates the arguments and their associated types. The final line indicates the output type.

For example, the entry

```
function range "Range"
input    quadtree:file x:integer y:integer radius:integer
output   x1: integer  y1:integer
```

specifies a function called `range`, whose input consists of three arguments: a `file` and two integers, and whose output is a *set* of pairs of integers. For technical reasons, boolean functions such as `less` will either return a point (`x1`, `y1`) to indicate true, or it returns a null represented by the distinguished pair $(*, *)$.

2.3.2 The Standard Data Format

At run-time, when a domain function `f` is called, the solution returned will adhere to the output format associated with the function `f` in the domain catalog. However, in order for the mediator to access the data uniformly across domains, some simple and consistent internal data format needs to be maintained. This data format should be sufficiently general to represent a variety of external data formats – a non-trivial task. We do not purport to provide an all-encompassing solution. However, the approach described below appears to work well in most cases.

A special data structure is associated with each domain, function pair `d`, `f`. This data structure, denoted $\mathtt{d_f}$, is a record type whose field names and types are identical to those in the

```
domain quadtree "SPATIAL"
    /* Variable Data */

function all "All Points"
input    quadtree:file

function range "Range"
input    quadtree:file x:integer y:integer radius:integer
output   x1: integer  y1:integer

function vertical "Vertical Slice"
input    quadtree:file left:integer right:integer
output   x1: integer  y1:integer

function horizontal "Horizontal Slice"
input    quadtree:file lower:integer upper:integer
output   x1: integer  y1:integer

function point_data "Point Data"
input    quadtree:file x:integer y:integer
output   value: void

function less "Less Than"
input    quadtree:file field:name value:void
output   x1:integer y1:integer

function less_equal "Less Than or Equal"
input    quadtree:file field:name value:void
output   x1:integer y1:integer

function equal "Equal"
input    quadtree:file field:name value:void
output   value1: boolean

function not_equal "Not Equal"
input    quadtree:file field:name value:void
output   x1:integer y1:integer

function great_equal "Greater Than or Equal"
input    quadtree:file field:name value:void
output   x1:integer y1:integer

function great "Greater Than"
input    quadtree:file field:name value:void
output   x1:integer y1:integer
```

Figure 2: A Catalog for SPATIAL

output field of `f`'s output format described in the catalog. Returning to the `SPATIAL` domain and the function `RANGE` defined therein.

```
function range "Range"
input    quadtree:file x:integer y:integer radius:integer
output   x1: integer  y1:integer
```

This internal data structure $\mathtt{d_f}$ corresponding to the domain-function pair (`spatial`, `range`) is:

```
type spatial:range = record of x1 : integer; y1 : integer end
```

Had one of the field types been a complex type, then the corresponding complex type would have to be included in the internal data structure. For instance, if the output of `range` is of type `point` where `point` is a user-defined record having two fields x and y, each of type integer, then the data structure defined in the internal format would appear as

```
type spatial:range = record of
     p : point
     end;
```

These *declarations*, one for each function in the domain being integrated, will be placed in a file called the *output-format* file, which can be automatically generated by examining the domain catalogs. The compiler will subsequently generate variable declarations to correspond to each of the types defined above.

2.3.3 Internal Facts

The mediator compiler keeps track of a number of internal facts whose values serve different purposes. For example, there is an internal clock used to keep track of the current time (year, month, day, hour, sec, millisec). The value of the clock will maintain temporal data on the creation of various data sources. When a new data source is integrated into a mediated system, the mediator compiler stores the time at which this is done. This information will facilitate conflict resolution strategies, described later in Section 2.4.2. The time is then stored through a special predicate called `data`. For instance, the fact

$$\mathtt{data(dbase,'part.2', \langle Schema\rangle, \langle Time\rangle) : 0.8}$$

declares `part.2` to be a Paradox relation having the specified schema, and was created at time *Time*. The number 0.8 is a *criticality value* that specifies that this data is believed to be correct with 80% certainty.

2.4 The Mediator Programming Environment (MPE)

This section discusses the three toolkits that can be used to assist the mediator author in constructing a mediator. The first of these is used in the domain integration phase, while the latter two are for use in the actual writing of mediating rules.

2.4.1 The Domain Integration Toolkit

Domain integration enables a mediator to execute the functions defined in the domain being integrated, and to convert the output of those functions into a form intelligible to the mediator compiler.

The various steps involved in executing a function call of the form $\mathtt{d} : \mathtt{f}(\mathtt{a}_1, \ldots, \mathtt{a}_n)$ are described below.

1. (Domain Opening) When executing the above call, the software package named `d` must first be opened. If for example this software package is `PARADOX`, then it may be opened through a C programming interface. In case no C programming interface is available, then a shell script or a batch file may be used. In general, the C programming interface is preferable due to its speed. A shell script on the other hand, involves less coding and may suffice if the domain in question is seldom used. The domain integration toolkit makes available both options to the mediator author. When the mediator author interacts with the toolkit, directions will be given on where the code, whether it accesses the C programming interface or the shell script, should be placed.

2. (Domain Execution) The next step involves developing facilities to handle output from run-time calls to different domain function. Specifically, this involves creating the data structures in the output-format file referred to above. Recall that a call of the form $\mathtt{d} : \mathtt{f}(\mathtt{a}_1, \ldots, \mathtt{a}_n)$ returns a *set* of objects from the data source. The type of this set can be determined at compile-time by examining the output-format file of the function `f`. There are two ways in which this set of objects may be stored and used by the mediator. The current implementation simply places them in a flat ASCII file. The output format of the function `f` declared in the domain catalog for the domain `d` is used to *parse* this flat file. Thus, for instance, if the output of a function such as `range` in the spatial domain is a set of points, then the output file may be a flat ASCII file containing:

   31 5 27 11 19 26

   Assuming that each `point` is a pair representing an x and an y coordinate, this file can be parsed to obtain 3 points – $(31, 5)$, $(27, 11)$ and $(19, 26)$. It is the responsibility of the person performing the domain integration to implement the necessary parse routines, and he/she will be appropriately instructed by the domain integration toolkit.

   An alternative to the file approach is currently under development that would maintain the set of solutions in a dynamic data structure in main memory. For example, we may use a simple linked list, and to use insertion and deletion algorithms to manage the list. Each node in the linked list contains a record whose form corresponds to the output format specification for $\mathtt{d} : \mathtt{f}$ (as specified in the output format file described in Section 2.3.2), and a pointer to the next element in the linked list. The insertion algorithms themselves would share a common skeleton that is independent of what `d` and what `f` are. However, the node structure would differ because the first field of a node would be a pointer to a structure of type $\mathtt{d} : \mathtt{f}$. Writing the insertion routines requires some work on the part of the programmer. However, it should only involve minor modifications to a skeleton code (e.g. replacing calls to $\mathtt{d}_0 : \mathtt{f}_0$, say, by calls to $\mathtt{d} : \mathtt{f}$).

The main advantage of the file approach is that the results of a function call can easily be stored on disk. Thus, computations to domains can be cached (either in core, or on disk). In the case when large amounts of data are returned by domain functions, or that the computation of the domain functions is time-intensive, this would lead to a tremendous saving by avoiding recomputation. On the other hand, storing results to files itself requires considerable overhead.

3. (Domain Completion) The third step involves closing the software package once the computation of the function calls is complete. This requires writing code for detecting when an operation has completed its execution, and can be done only on a case-by-case basis. The mediator author is responsible for writing this code.

The mediator programming environment (MPE) features a graphical interface to assist in the following five step systematic integration of domains.

1. First the name of the domain being integrated is requested by the MPE in order to create a catalog file for this domain.
2. It then asks for a list of functions in this domain that will be accessed by the mediator.
3. For each function in the above list, the mediator author specifies the input and output format. This specification is written into the catalog file for the domain as described in the domain execution phase.
4. The mediator author then implements the code necessary for performing the domain opening, execution, and completion phases.
5. For each body of data being accessed within a domain the mediator author is queried on whether the data contains time-stamp information, and to implement, if necessary, the code to convert the time-stamp in the domain to a time stamp consistent with the standard clock maintained by HERMES. For each body of data, the toolkit automatically generates a `data()` declaration as described earlier. Note this declaration also contains a time-stamp obtained from HERMES.

2.4.2 The Conflict Resolution (CR) Toolkit

The purpose of the conflict resolution toolkit is to assist the mediator author in dealing with semantic conflicts that might arise during semantic integration. A menu of commonly used strategies is available to the mediator author. For each predicate defined in the mediator, the mediator author has the option of choosing to apply one of these conflict resolution strategies. Each strategy selected will cause the CR Toolkit to generate a set rules that becomes a part of the mediator. On the other hand, the mediator author may define a new strategy by writing a set of rules. The learning module (cf. Section 2.4.4) will then attempt to determine whether the rules embodies a strategy that should be developed further for future use. Below, we specify some existing conflict resolution strategies currently implemented in the toolkit, and describe how these facilities are incorporated into the mediator compiler.

We assume a background set of integrity constraints that expresses invariant semantical relationships between different data sources. The syntactic forms of these constraints are not important for our discussion.

Latest Data Preference Strategy: Very simply, this strategy says that preference is given to temporally more recent data. To accommodate this, the mediator accesses a special domain implemented in HERMES called `HTIME` (for HERMES time). `HTIME` contains a domain operation called `timefind` that takes a relation name, say r, and a tuple t as input. It first checks if the relation r has associated time-stamp information. Recall that during domain integration, this information may either be explicitly included by the domain integrator, or implicitly generated by the mediator compiler and stored in the `data()` declaration. If a time-stamp exists, a temporal conversion function supplied by the domain integrator is invoked with r as input. The value returned is the time, normalized to a global clock maintained internally by HERMES. If relation r has no associated time-stamp information, then `timefind` examines the appropriate `data` declarations in the mediator to determine the time at which relation r was introduced into the mediator system. This time is used as the time at which the data was created. Using the results returned by `timefind`, the most recent data is selected whenever conflicting data arises.

Predicate Preference Strategy: Many databases may define concepts that are essentially the same. Consider a `PARADOX` relation called `part1` and a `DBASE` relation called `part2`. Suppose `part1` has a field `color`, and `part2` has a field `col`. Though `part1.color` and `part2.col` are syntactically distinct, they in fact refer to the same semantic concept, viz. color. The CR-Toolkit allows a mediator author to express that in the Paradox domain, if $\texttt{PARADOX} : \texttt{part1.name} = \texttt{DBASE} : \texttt{part2.name}$, then `part1` and `part2` are the same, and hence their colors should be identical, i.e. $\texttt{PARADOX} : \texttt{part1.color} = \texttt{DBASE} : \texttt{part2.col}$. The system, upon recognizing this integrity constraint, will generate all possible violations. For each violation, it asks the mediator author to indicate a preference between the two relations. Hence the mediator author may say, for instance, that the `PARADOX` database's information is to be preferred with respect to color, but that the `DBASE`'s value is to be preferred with respect to price. Once the indications are made, rules are generated by the CR-toolkit to reflect these preferences.

Object Preference Strategy: The idea here is similar to predicate preference except that preference is indicated over individual objects. Hence it relies on the assumption that certain databases have more reliable knowledge of the attributes of certain objects than other databases. For example, a mediator author may note that what is called “dog” in one database is identical to what is called “chien” in another. Furthermore, a preference may be expressed that values of all attributes associated with “dog” in a database override values stored for the same attributes in other databases. The CR-Toolkit allows the mediator author to interactively express such preferences, and automatically generates the appropriate rules that will be placed in the mediator.

Value-Based Preference Strategy: Value-based conflict resolution may be appropriate for certain business applications. In this strategy, the user can specify that when two numeric fields disagree, either the higher or the lower value ought to be picked. Hence for instance, the integrity constraint

$$\texttt{PARADOX}:\texttt{part1.name} = \texttt{DBASE}:\texttt{part2.name} \Rightarrow \texttt{PARADOX}:\texttt{part1.price} = \texttt{DBASE}:\texttt{part2.cost}.$$

says (assuming identical currencies) that two parts should have the same price in both `PARADOX` and `DBASE` databases. However, should this constraint fail, the mediator-author who is writing the mediator on behalf of the company that produces these products may choose to quote the higher price. This is an example of a *value-based* preference strategy. The CR-Toolkit allows the user to interactively express such preferences. Again, appropriate rules will be generated by the toolkit and placed in the mediator.

Criticality Number Strategy: Recall that when creating the mediator, the mediator author includes various declarations such as the `data` declarations described earlier. Similarly, we have a special declaration called a *criticality* declaration of the form

$$\texttt{criticality(filename)} : \texttt{Crit},$$

where `Crit` is a real number. The higher it is, the more "reliable" is the data in that domain. The Criticality Number Strategy allows the user to use the rather coarse indication that all data in a domain of a specified criticality c_1 is "more reliable" than all data in a domain of criticality $c_2 < c_1$. This broad, sweeping strategy may be used as the default in cases where, for example, the mediator author does not truly believe that conflicts will arise.

2.4.3 The Information Pooling (IP) Toolkit

Information pooling is a complicated matter. It is further complicated when the sources from which the information is pooled differ vastly from one another, both in structure, and in content. To facilitate this process, the IP-Toolkit consists of a number of subtoolkits that allow integration both within a given domain, as well as integration across different but similar domains, as well as integration across multiple and dissimilar domains. For example, suppose integration is to be performed over eight databases (call them $DB1, \ldots, DB8$). Suppose moreover that $DB1$ and $DB2$ are `PARADOX` databases, $DB3$ and $DB4$ are `DBASE` databases, $DB5$ and $DB6$ are text databases, and $DB7$ and $DB8$ are pictorial databases. Figure 2 shows how the IP Toolkit in HERMES would accomplish the desired integration. As can be seen from Figure 2, the IP-toolkit contains many "sub-toolkits" that need to be independently developed. Each of these subtoolkits allows the generation of some rules (but not all) within the mediator. Using the subtoolkits leads to a stepwise procedure. For instance, in Figure 2, the mediator author could define certain rules in the mediator by using the `PARADOX` IP Toolkit; similarly, he may define certain rules in the mediator by using the `DBASE` IP toolkit; he may then define new mediator rules based on these intermediate definitions, perhaps in conjunction with the relational IP toolkit.

Intra-Domain Sub-Toolkits: Often, there may be relatively efficient ways of integrating information *within* a given domain. For example, if a number of different databases are all developed using `PARADOX`, then as they are all structurally similar, it would be easier perform integration over these databases first whenever possible, before doing integration with data sources implemented in other database management systems.

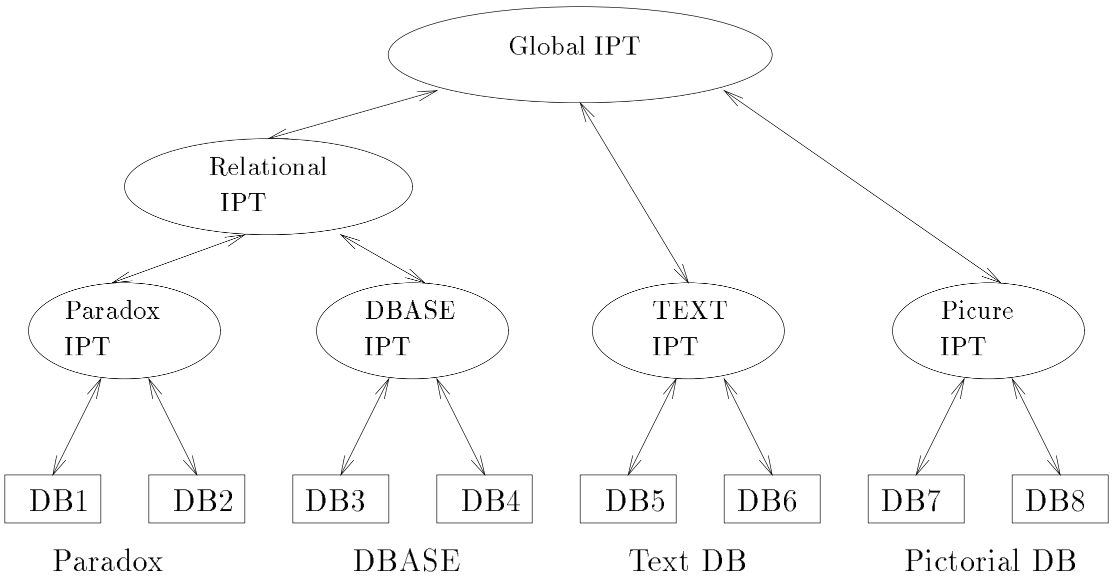

Figure 3: An Example of the Use of the IP-Toolkit

As an example, consider the integration of the two `PARADOX` databases $DB1$ and $DB2$ in Figure 2. These databases may have relations called `emp_comp1` and `emp_comp2`, respectively, showing the employee records of two companies. It may turn out that a particular employee obtains money from both; in such cases, the mediator author may wish to write a rule of the form:

$$
\begin{aligned}
&salary(X,S) : [1, \mathbf{R}] \leftarrow \\
&\qquad in(Sal1, \mathtt{PARADOX} : \mathtt{project}(\mathtt{select}_{=}('db1.emp\_comp1', "name", X)"salary")) \,\& \\
&\qquad in(Sal2, \mathtt{PARADOX} : \mathtt{project}(\mathtt{select}_{=}('db2.emp\_comp2', "nom", X)"sal")) \,\& \\
&\qquad Sal = Sal1 + Sal2.
\end{aligned}
$$

Note that the `name` (resp. `salary`) fields of the `emp_comp1` relation in $DB1$ are identical to the `nom` (resp. `sal`) fields in $DB2$.

In this example, the `PARADOX` Intra-Domain Sub-Toolkit would present, for each database in the `PARADOX` domain, relation-attribute pairs of the form $R.A$ and ask the mediator author for assistance in grouping together semantically "equivalent" $R.A$ pairs. In the preceding example, this involves grouping together the pairs $(db1.emp\_comp1).salary$ and $(db2.emp\_comp2).sal$ in one group, and $(db1.emp\_comp1).name$ and $(db1.emp\_comp1).nom$ in another.

Within any given domain, the mediator author must specify the *structures* of interest with respect to grouping. Once this specification is written, the *mediator author* would need to implement code that accomplishes this grouping. Note, however, that this code needs to be implemented only the first time a new domain is integrated – for example, if we merely wish to add a new `PARADOX` database, say $DB9$, then there is no need to re-implement this grouping code.

Sub-Toolkits for Integrating "Similar" Domains: Formally, `DBASE` and `PARADOX` are two different domains; however, these two domains share several characteristics. For example, as they are both relational database management systems, hence the operations of `select`, `project` and `join` are common to both of them. So are many other aggregate operations including `SUM`, `MIN`, `MAX`, and `COUNT`.

Therefore, it may be advantageous to create a sub-toolkit that accesses the predicates that have been defined in the mediator using the `PARADOX` and `DBASE` IP subtoolkits, and then using them to define other parts of the mediator. This is the purpose of the sub-toolkits for integrating similar domains – the mediator author, though, is left to decide which domains are similar.

Just as in the case of the Intra-domain subtoolkits, the subtoolkits for integrating similar domains are used to group structures together. For example, in the case of say `PARADOX` and `DBASE`, this involves grouping together pairs $(R1.A1)$ from the `PARADOX` domain with pairs $(R2.A2)$ from the `DBASE` domain. Thus, for instance, the mediator author may specify that $(db2.emp\_comp2).sal$ in the `PARADOX` domain is the same as $(db3.employee).income$ in the `DBASE` domain.

Sub-Toolkits for Integrating Diverse Domains: At this point in our system, this integration must be done by the mediator author. The GUI for the MPE merely serves as a tool for expressing the mediation rules.

2.4.4 Learning in HERMES

Though small, an important aspect of the HERMES mediator programming environment is a facility for learning. When a mediator author develops a mediator, strategies other than the ones available in the CR-Toolkit may be necessary. Thus the mediator author needs to specify such a strategy through a set of mediator rules. This set, appropriately generalized, may be useful across a sufficiently large number of mediation tasks, that storing it as a new strategy in the CR-Toolkit for later use is worthwhile.

The decision on whether the new strategy will be useful in the future lies with the mediator author, while the appropriate generalization to the set of rules may often be generated automatically. In certain cases the mediator author may need to manually adjust the strategy thus generated.

A similar learning component may be developed for the Information Pooling Toolkit.

2.5 Yellow Page Servers

A basic problem that arises when creating a mediator is that often, one may not know which data source contains the information desired. Moreover, even if the data source is known, the mediator author may not know how it can be accessed.

We envisage using a *hierarchy* of yellow pages servers to assist the mediator author in locating a data source. They will contain information for each domain that is currently accessible in HERMES, including information useful for accessing the domain. Yellow-pages servers can themselves be regarded as *domains*, just like all other domains. It is defined as part of HERMES, and hence the mediator author need not perform domain integration to access it. However, additions may be necessary to create new entries into the yellow pages as new domains are integrated.

Within our yellow-pages domain, we have exactly one function called `topic` that a mediator may invoke. This function takes as input, a string representing the name of a topic that the user is interested in. It returns as output, a *set* of structures. Each structure contains the name of the domain in which information about the named topic resides, and any associated information stored in the yellow pages server about this domain. As mentioned above, the information can be used by the mediator to access the domain. Hence, in particular it must contain the names of functions that can be used within this domain to access that data, and the list of arguments (and associated types) that the above function takes.

Consider for example the databases shown in Figure 3. Assuming that we are interested in the topic `money`, then the following predicate accomplishes this task:

$$\begin{aligned}\mathtt{find(Topic,D,F,A):1} \quad \leftarrow \quad & \mathtt{in(X,yp:topic(Topic))}\ \& \\ & = (\mathtt{X.dom},\mathtt{D})\ \& \\ & = (\mathtt{X.func},\mathtt{F})\ \&\ = (\mathtt{X.args},\mathtt{A}).\end{aligned}$$

The call `yp:topic(Topic)` calls the yellow pages domain with the given `Topic`. The returned set of values are structures of the form discussed. Hence the query $\leftarrow$ $\mathtt{find(``money'',D,F,A):1}$ accomplishes the desired task. The predicate `find` may now be used to define other, more complex predicates that require locating a resource before using it. The precise data structures and algorithms for implementing the yellow pages domain are currently under development.

2.6 Heterogeneous User Interface for End-Users

The user interface in our system architecture consists of a framework wherein end-users may pose a conjunctive query $\mathtt{A_1}\ \& \ldots \&\ \mathtt{A_n}$ using a variety of input forms. For example, consider a simple query where a user has a picture of someone displayed on the screen, and the user wishes to ask the query: "Find the name of the father of this person." The query may be performed in two parts. The first is *pictorially specified* by drawing a box around the person's face displayed on the screen. The second is to *textually* input the query "Find me the father of the person whose face I just enclosed in a box." This can be written logically as:

$$\begin{aligned}\leftarrow\ & \mathtt{in(X,facedb:matchface('box','mugshot\_lib'))}\ \& \\ & \mathtt{in(Y,paradox:project(select_{=}('parent',"person",X),father))}.\end{aligned}$$

The above assumes the existence of a face database that contains the pre-defined function `matchface` which takes an input face, and determines the name of the person pictured by comparing it against a mugshot library. Similarly, we are assuming that the `parent` relation is stored in the `PARADOX` DBMS, and that the relation has the scheme (`Person`, `Father`, `Mother`).

In the above query, the `facedb:matchface('box','mugshot_lib')` part is being specified using graphical input, while the $\mathtt{paradox:project(select_{=}('parent',"person",X),father)}$ part is specified using textual input.

In general, the purpose of the heterogeneous user interface is to support such conjunctive queries where parts of the the query can be expressed using different forms of input. In the long run, we expect HERMES to support a wide variety forms of inputs (e.g. voice, SQL, logic, point and click menus, graphical inputs, etc.), but for now, the system supports logic queries, point and click pulldown menu-based queries, and limited graphical inputs.

3 Applications

In this section, we will present two case studies that have/are being developed as applications on top of HERMES.

3.1 Case Study: Integrating Terrain Maps, Relational Databases, and Planning

In this section, we will describe an application of our work to intelligent terrain reasoning that involves integrating terrain map data, relational data, and planning packages (developed at the US Army Corps of Engineers). The purpose of such an integrated system is many-fold. It can be used as a basis for vehicular navigation in disaster relief situations (e.g. floods, earthquakes, volcanic disasters, etc.), as well as in military mission planning applications. In these applications, a user, who may either be a human or may be an autonomous vehicle, may be interested in posing queries of the following types:

- **(Unknown Destination)** Given a location, find a place that has an airfield as well as certain types of ammunition. Presumably these resources are needed in order for the autonomous/manned vehicle to satisfy its mission.
- **(Route Properties)** Furthermore, no point in the route may be less than 4 miles from an enemy outpost. In this example, in addition to the fact that the destination is unknown, we have the fact that the query asks not only for a route to this unknown destination point, but it asks for a *route that satisfies certain desiderata*, i.e. which satisfies certain conditions that require accessing external databases (e.g. to figure out where enemy outposts lie).

A route planner (which we will call `RP`) has been implemented at the US Army Topographic and Engineering Center. Given two points, this route planner will find an optimal, least-cost path between these two points (if one exists). Thus, for instance, the query

$$\mathtt{rp} : \mathtt{route}((35, 70), (200, 98))$$

returns the set of least-cost paths from the origin point, $(35, 70)$ to the destination point $(200, 98)$ that are found by the Army's route planner.

We illustrate how this example may be solved within the HERMES framework, using `RP` as a domain. For this, let us suppose that we have a relational (`PARADOX`) database containing a relation called `facilities` having the schema (Name,Facility). Thus, this relation may contain a tuple of the form $(awasa, airport)$ denoting that the place, Awasa, has an airport. Other tuples in the relation `facilities` may be similarly interpreted. Suppose there is another (`DBASE`) database containing a relation called `supplies` having the schema (Place,Item) – an example tuple in this relation is $(awasa, gas)$ specifying that gas is available at Awasa[5].

[5] In practice, these relations will contain much more detail, but we keep them simple here in order to facilitate presentation.

Using HERMES, we have integrated the following *four domains* – RP, DBASE, and PARADOX – along with a spatial database that identifies points (xy-coordinates with place names). We have written a mediator that may be used to solve this example using the rule given below.

Query 1: Let rte1 be a ternary predicate such that $\texttt{rte1}(O, D, R)$ is satisfied iff R is a route from the origin to an unspecified destination such that the destination has an airfield as well as certain types of ammunition. For this, we may define the following clause in the mediator:

$$
\begin{aligned}
&\texttt{rte1(O, D, R)} \leftarrow \\
&\qquad \texttt{in(P1, paradox : select}_=\texttt{(facilities, facility, ``airfield''))} \,\&\\
&\qquad \texttt{in(P2, dbase : select}_=\texttt{(supplies, item, ``ammunition''))} \,\&\\
&\qquad \texttt{= (P1.place, P2.place)} \,\&\\
&\qquad \texttt{in(D, spatial : findpt(P1.place))} \,\&\\
&\qquad \texttt{in(R, rp : route(O, D))}.
\end{aligned}
$$

To obtained the result from a given location ℓ, we can pose the query

$$\leftarrow \texttt{rte1}(\ell, \texttt{D}, \texttt{R}).$$

This is then processed as follows: PARADOX is invoked which SELECTs all tuples from the facilities relation that have the an airfield facility. P1 is then instantiated to one of the selected tuples. Next DBASE is then queried to SELECT all tuples from the supplies relation that have the item field set to ammunition. P2 is instantiated to one such tuple. A check is made to see that P1 and P2 have the same place field. In other words, this ensures that *single* place is found with both ammunition and an airfield? If not, the HERMES inference engine looks for other possible instantiations of P1 and P2 that satisfy these constraints. Finally, the xy-location of the place P1.place is computed using the spatial domain, instantiating D, and RP is called to find a route from the origin ℓ to D.

Solving the second query is somewhat more complicated because of the additional requirement that no enemy outpost lies within 4 miles of any point on the route. Let us suppose that the spatial domain contained in our solution to Query 1 above has a special function called range that takes a point P and a radius Rad as inputs and returns all enemy outposts within Rad units of P.

Query 2: We define a new predicate called rte2 as follows.

$$
\begin{aligned}
&\texttt{rte2(O, D, R)} \leftarrow \texttt{rte1(O, D, R)} \,\&\, \texttt{good(R)}.\\
&\texttt{good(nil)} \leftarrow\\
&\texttt{good(cons(H, T))} \leftarrow \texttt{goodpoint(H)} \,\&\, \texttt{good(T)}.\\
&\texttt{goodpoint(H)} \leftarrow \texttt{is(\{\}, spatial : range(H, 4))}.
\end{aligned}
$$

Note the use of the special HERMES predicate is described earlier.

3.2 Case Study: Face Recognition for Law Enforcement Applications

We have also been developing a mediator in HERMES for law enforcement applications. The general problem is best illustrated with an example. Consider the problem of identifying all people P who have been recorded, by surveillance cameras, as having met with an individual

X (for instance, X may be a Mafia chief like Don Corleone), who live within a hundred mile radius of Washington DC, and who work for a suspected front company "ABC Corp." Solving this problem may require access to a wide variety of data sources and furthermore, require recourse to diverse *reasoning paradigms* as well. For example, it may be necessary to

- access a background **face database** containing pictures (e.g. passport pictures) of individuals. In this face database, the identity of the photographed individuals is known.
- access a database containing **surveillance photographs**, that might have been obtained by surveillance cameras.
- access **face-extraction algorithms** to extract the "prominent" faces from the images generated by the surveillance camera.
- access methods of **matching** faces extracted from the surveillance data by the face-extraction algorithm, so as to be able to figure out who appears in which images.
- access a **relational database** (`PARADOX`) of phones and addresses.
- access a **spatial database** in order to determine whether a given address lies within 100 miles of Washington DC.
- access a **second relational database** (DBASE) about the employees of ABC Corp.

We define the following three mediating rules.

```
seenwith(X, Y) ←
        in(P1, facextract : segmentface('surveillancedata')) &
        in(P2, facextract : segmentface('surveillancedata')) &
        = (P1.origin, P2.origin) &
        P1 ≠ P2 &
        in(P3, facedb : findface(X)) &
        in(true, facextract : matchface(P1, P3))&
        in(Y, facedb : findname(P3)).
swlndc(X, Y) ←
seenwith(X, Y) &
        in(A, paradox : select_eq('phonebook', "name", X)) &
        in(Pt1, spatialdb : locateaddress(A.streetnum, A.streetname, A.cityname)&
        A.statename, A.zipcode)) &
        in(true, spatialdb : range('dcareamap', Pt1.X, Pt2.X, 100)).
suspect(X, Y) ← swlndc(X, Y) &
        in(Tuple, dbase : select_eq('empl_abc', "name", Y)).
```

The `seenwith` predicate access a domain called `faceextract` that we assume is a pattern recognition package, and that it uses the function `segmentface` to locate the faces in a set of photographs. The output of the function contain mugshots that can be stored. The extraction procedure returns a list of pairs of the form ($<\mathtt{resultfile}, \mathtt{origin}>$) specifying which image in the surveillance data, a given face was extracted from (the origin) and where the mugshot is now stored. The `faceextract` domain also contains a function called `matchface` that takes a face (such as those extracted by the `faceextract` domain) and checks if this face is identical to another face in the mugshot library. Likewise, the `seenwith` predicate access a domain called `facedb` containing a function called `findface` which determines, given a person's name, whether his or her face is in a mugshot library. The `facedb` domain also contains a function called `findname` which, given a mugshot in the mugshot library, returns the name of the person involved.

Given that a person `Y` has been `seenwith X`, `swlndc` (for "seen with and lives near DC"), accesses a relational database to find the address of `Y`, and then accesses a spatial data management system to determine what $(\mathtt{x}, \mathtt{y})$ coordinates, on a map of the DC area, this address corresponds to (using a function called `locateaddress`). It then determines, using a function called `range`, whether this address lies within the specified distance from DC.

Finally, a person `Y` is a suspect just in case `swlndc("Don Corleone", Y)` is true and if he is an employee of "ABC Corp." For this, a DBASE relation called `empl_abc` is accessed. The above three clauses express the mediator for this example in its entirety (though the individual domains are not completely shown).

3.3 System Usage

When the user brings up HERMES under Unix, the first screen shown in Fig 4 appears. The top window in this screen contains a list of available mediators. The user may select a mediator by clicking on it with the mouse button. To select the terrain mediator, the user clicks on `terrain`. The second screen in Figure 4 shows the screen that results – the top screen contains various predicates defined within the terrain mediator. The buttons in the command bar at the top are suitably modified as well. The bottom window brings up a description of the mediator. The user may scroll through either the top window or the bottom window by using the scroll bar on the left. If the user wishes to find help information on a particular predicate, then s/he may click on that predicate with the mouse button. The first screen in Figure 5 shows the result of clicking on the `routeToFacility` predicate. At this stage, the second window from the bottom shows the usage of the predicate `routeToFacility` together with a help string explaining the utility of the predicate. The figure shows the screen after the user has keyed in a specific query

```
routeToFacility(100, 100, 50, F, RC, RL, R, S) : 1.
```

The new screen that pops up with the answer is shown in the second screen of Figure 5, The second window from the bottom contains the desired solution. Also observe that the top window now contains the statement

```
Files : 2.scr
```

indicating that the file `2.scr` is available for viewing. This is a script file. If the user wishes to view this file, then he may click on it – doing so generates a pictorial version of the actual path generated (see Figure 6 where the red path denotes the desired route to the facility in question. In general, the files shown in response to a user query may themselves be heterogeneous files such as .GIF files of .SCR files, plain text files, or hypermedia documents. The display of these files requires the use of the action module to "bring up" the files in the desired format.

3.4 Related Work

Following Wiederhold's pioneering paper on mediators [24, 25], there have been several efforts to develop general-purpose methods of integrating multiple databases. In contrast to approaches that specifically deal with integrating multiple databases, we emphasize that our system and framework are applicable to the integration, not only of multiple databases, but also to the integration of multiple software packages (e.g. spreadsheets), and also multiple reasoning paradigms (e.g. path planning, face recognition, terrain reasoning, etc.).

Multidatabases and federated databases form a well-known family of database integration paradigms. In such frameworks, an attempt is made to create a "global" integrated scheme from a number of local schema – then the global schema is queried. Most of the works in this arena are used to integrate multiple *relational* databases (and in some cases, object oriented databases), identifying schema mismatches, and eliminating them as part of the query processing procedures. Some elegant pieces of work on multidatabases are those due to Litwin et. al. [13, 14, 6]. In addition, Krishnamurthy, Litwin and Kent [11] define criteria that a language that a language that integrates multiple databases must satisfy. They show how to handle schematic discrepancies. Intuitively, two databases may contain the semantic information, but this information may be represented in different ways (e.g. in one database, all the information may be contained in one relation, while in the other, the same information may be spread over two relations). They define a query language that takes such schematic mismatches into account when handling queries. In a similar vein, Spaccapietra and Parent [21] structural conflicts that arise when multiple databases contain similar, but structurally different data. They show how data across multiple databases can be "linked" or "tied together" based on common attribute values (even though the attribute names may be different across different databases). Similarly, Bright et. al. [5] and Sciore et. al. [18] develop an array of methods to automatically resolve (using different techniques) semantic conflicts in multidatabases. Their work can be used to contribute to the conflict resolution toolkit that forms part of the HERMES architecture.

Generally speaking, the HERMES system applies to a wide diversity of domains with very different aspects to them such as the path planning domain, the face recognition domain, etc., that are not handled by multidatabases. As such, the types of software packages and the kinds of reasoning performed in systems like HERMES are broader than those in multidatabases. Other recent efforts that are proceeding synchronously with ours, are the SIMS/LOOM effort at ISI [2] and efforts to develop a language called KQML [12]. SIMS is a system that uses an object-oriented language called LOOM to integrate information from multiple databases. Objects in the world are classified into classes and a set of relations that link objects and classes are also stored. Various types of inheritance properties can be expressed in LOOM. Both HERMES and SIMS can benefit from one another in a variety of ways – for instance, HERMES' generalized domain integration method is applicable to the SIMS model as well. As mentioned, in our Heterogeneous User Interface, the LOOM query language could form an important component therein, allowing users who prefer LOOM's LISP-flavored syntax to express their queries in it. KQML, in contrast, provides a method for communication requests to be made between a mediator and various software packages. Various KQML primitives like **ask-if**, **ask-all**, etc. may be used to manipulate belief-states that an agent may have (for example, this agent may a communications module that controls a communication channel).

HERMES could benefit from KQML by using KQML agents in order to effect communications across the network. In contrast, KQML could benefit from HERMES by using HERMES' query language to identify and extract relevant information from different data sources, pool them together after identifying/resolving conflicts. We see rich potential interactions with both SIMS and KQML/LOOM.

4 Conclusions

Today's world contains a vast quantity of electronic data contained in databases and data structures located across the world. Often, an individual or organization has needs that involve accessing existing, legacy databases, that were created by a different organization for (possibly) an entirely different purpose. In other applications, the very *type* of data being stored (e.g. pictures, audio, etc.) may necessitate storage techniques that are unique to that form of data. When performing complex reasoning tasks (such as the terrain reasoning example, and the law enforcement example), we may need to access multiple representations of data. More importantly, complex reasoning tasks require that we not only access multiple types of data, but that we access implementations of multiple reasoning paradigms (e.g. face recognition, pattern recognition, etc.).

In this paper, we have used the notion of a "mediator," due to Wiederhold [24, 25] as the starting point for our work on integrating multiple databases and multiple software packages. Based on this idea, we have proposed a logic-based language for expressing these mediators in. The language is simple and easy to use and provides a uniform way of passing different types of data structures "back and forth" between the mediator and individual software packages. This language for expressing mediators in represents the backbone of our HERMES ("HEterogeneous Reasoning and MEdiator System") system. We have described the overall systems architecture of HERMES, and also described, in further detail, the different components of the system. Mediators may now be developed within the HERMES framework – HERMES represents not only a language for expressing mediators in, bu a complete environment for developing integrated mediated systems in. At the current point, our HERMES system inter-operates between a number of different software packages developed commercially (e.g. INGRES, PARADOX, DBASE, ObjectStore in progress), as well as two implementations of reasoning paradigms developed by third parties (face recognition, path planning), as well as some locally developed software packages (e.g. spatial databases, text databases, multimedia databases). HERMES has various B-tree based indexing schemes that allow it to efficiently access over 10 Gigabytes of data on auxiliary storage.

The great challenge for scaling up systems like HERMES is to move from the Gigabyte domain to the Terabyte domain. We are currently investigating techniques for the efficient implementation of a distributed version of HERMES. This would recognize the practical reality that usually, the databases and/or software packages that we may be inter-operating between, are located across multiple nodes on the network. In addition, by utilizing distributed processors, it is hoped that the Terabyte challenge can be addressed. However, at this stage, it is too early to pass judgement on distributed scalability of HERMES.

References

[1] S. Adalı, V.S. Subrahmanian, Amalgamating Knowledge Bases, III: Distributed Mediators, *International Journal of Intelligent Cooperative Information Systems*, 1994.

[2] Y. Arens, C.Y. Chee, C.-N. Hsu and C. Knoblock. (1993) *Retrieving and Integrating Data From Multiple Information Sources*, International Journal of Intelligent Cooperative Information Systems, 2, 2, pps 127–158.

[3] J. Benton, V.S. Subrahmanian, Using Hybrid Knowledge Bases for Missile Siting Problems, *Proceedings of the Conference on Artificial Intelligence Applications*, IEEE Computer Society, 1994.

[4] Y. Breitbart, H. Garcia-Molina, A. Silberschatz, Overview of Multidatabase Transaction Management, *Journal of Very Large Databases*, 2, pp. 181–239, 1992.

[5] M.W. Bright, A.R. Hurson, S. Pakzad, Automated Resolution of Semantic Heterogeneity in Multidatabases, *ACM Transactions on Database Systems*, 19, 2, pp. 212–253, 1994.

[6] J. Grant, W. Litwin, N. Roussopoulos, T. Sellis, An Algebra and Calculus for Relational Multidatabase Systems, *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, IEEE Computer Society Press, pp. 118 – 124, 1991.

[7] J. Horst, E. Kent, H. Rifky, V.S. Subrahmanian, Hybrid Knowledge Bases for Real-Time Robotic Reasoning, *Proceedings of the International Workshop on Pattern Recognition in Practice*, E. Gelsema, L. Kanal eds., North Holland/Elsevier, 1994.

[8] Kifer, M., Lozinskii, E., "A Logic for Reasoning with Inconsistency", *Journal of Automated Reasoning*, 9, pp. 179 – 215, 1992.

[9] M. Kifer, V. S. Subrahmanian, Theory of Generalized Annotated Logic Programming and its Applications, *Journal of Logic Programming*, 12, 4, pp. 335–368, 1992.

[10] W. Kim, J. Seo, Classifying Schematic and Data Heterogeneity in Multidatabase Systems, *IEEE Computer*, December 1991.

[11] R. Krishnamurthy, W. Litwin, W. Kent, Language Features for Interoperability of Databases with Schematic Discrepancies, *Proceedings of ACM Conference on Management of Data*, 1991.

[12] Y. Labrou and T. Finin. (1994) *A Semantics Approach for KQML – A General Purpose Communications Language for Software Agents*, Proc,. 1994 Intl. Conf. on Information and Knowledge Management, pps 447–455.

[13] W. Litwin, A. Abdellatif, Multidatabase Interoperability, *IEEE Computer*, pp. 10 – 18, December 1986.

[14] W. Litwin, A. Abdellatif, A. Zeroual, B. Nicolas, Ph. Vigier, MSQL: A Multidatabase Language, *Information Sciences*, 49, 1989.

[15] J. Lu, A. Nerode, V.S. Subrahmanian, Hybrid Knowledge Bases, *IEEE Transactions on Knowledge and Data Engineering*, 1994. To appear. Available as technical report CS-TR-3037, University of Maryland.

[16] S. Marcus, V.S. Subrahmanian, *Multimedia Database Systems*, 1994. Submitted for publication.

[17] W. Niblack, et. al., The QBIC Project: Querying Images by Content Using Color, Texture and Shape, IBM Research Report, Feb. 1993.

[18] E. Sciore, M. Siegel, A. Rosenthal Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems, *ACM Transactions on Database Systems*, 19, 2, pp. 254–290, 1994.

[19] H. Samet *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1989.

[20] A. Sheth, J. Larson, Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases, *ACM Computing Surveys*, 22, 3, pp. 183–236, 1990.

[21] S. Spaccapietra, C. Parent, View Integration: A Step Forward in Solving Structural Conflicts, *IEEE Transactions on Knowledge and Data Engineering*, 6, 2, pp. 258–274, 1994.

[22] V.S. Subrahmanian, Amalgamating Knowledge Bases, *ACM Transactions on Database Systems*, 19, 2, pp. 291–331, 1994.

[23] W.K. Whang, S. B. Navathe, S. Chakravarthy, Logic-Based Approach for Realizing a Federated Information System, *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, IEEE Computer Society Press, pp. 92–100, 1991.

[24] G. Wiederhold, Mediators in the Architecture of Future Information Systems, *IEEE Computer*, pp. 38–49, March 1992.

[25] G. Wiederhold, Intelligent Integration of Information, *Proceedings of the ACM Conference on Management of Data*, pp. 434–437, 1993.

[26] G Wiederhold, S. Jajodia, W. Litwin, Integrating Temporal Data in a Heterogeneous Environment, *Temporal Databases*, Benjamin Cummings, January 1993.

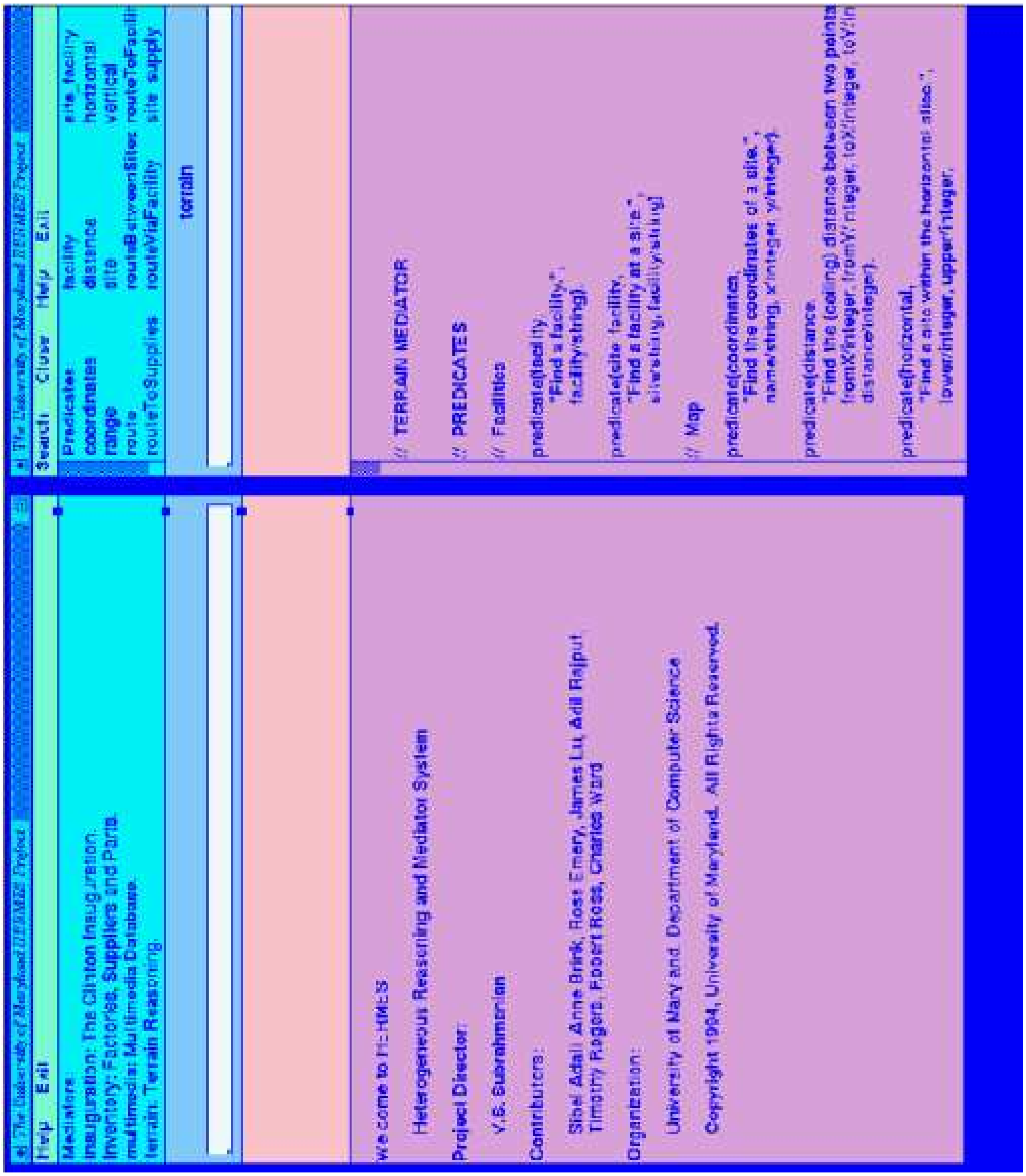

Figure 4: HERMES: Mediator Selection and Predicate Listing Screens

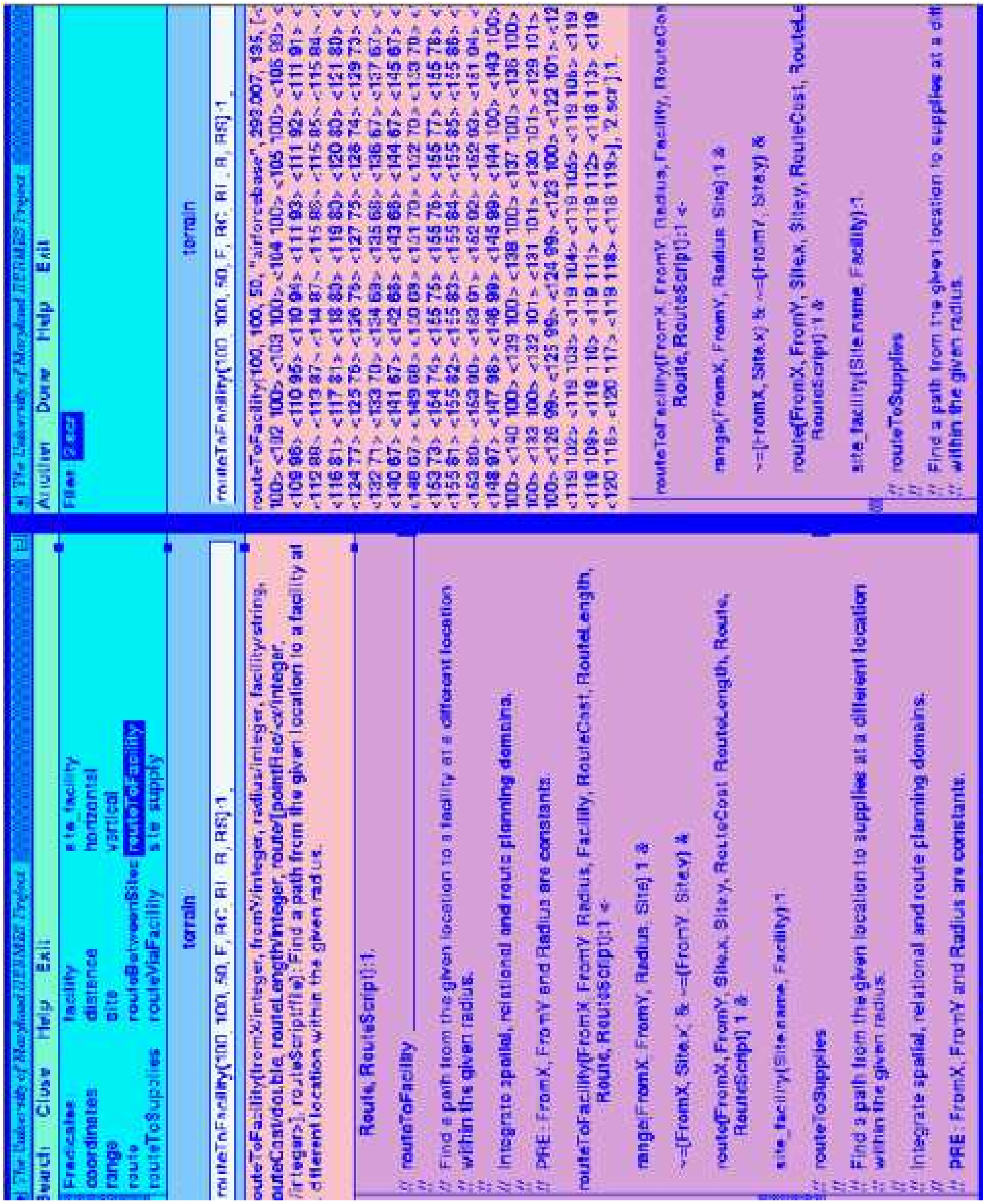

Figure 5: HERMES: Query Articulation Screen and Screen after Query Execution

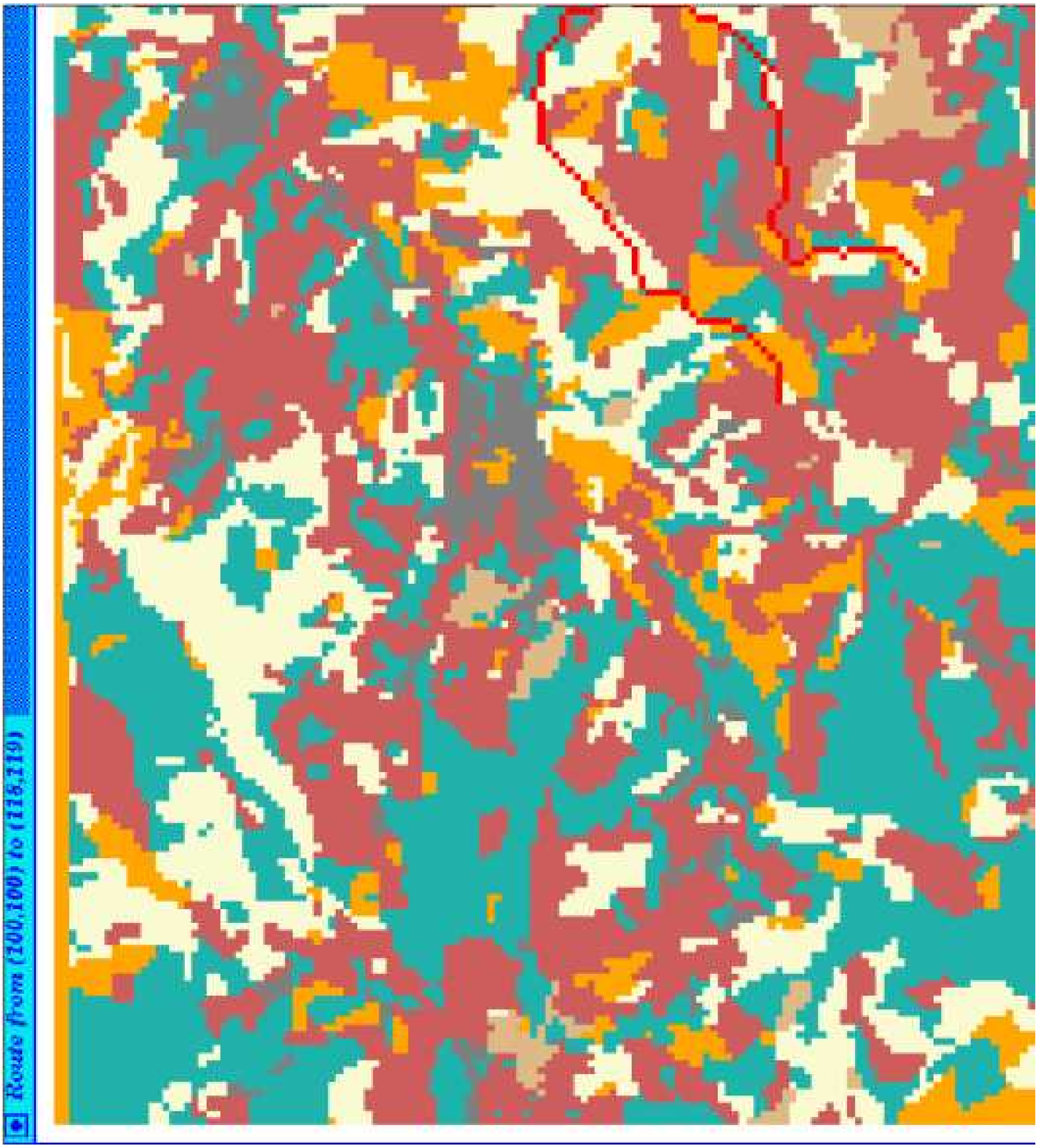

Figure 6: HERMES: Screen after Action Module's Intervention