A Uniform Framework For Integrating Knowledge In Heterogeneous Knowledge Systems *

Sibel Adalı & Ross Emery
Department of Computer Science & Institute for Advanced Computer Studies
University of Maryland, College Park, Maryland 20742.

Abstract

Integrating knowledge from multiple sources is an important aspect of automated reasoning systems. Wiederhold and his colleagues [5, 6] have proposed the concept of a mediator *– a device that will express how such an integration is to be achieved. In [1, 2, 11], Subrahmanian et al. presented a uniform declarative and operational framework for mediators for amalgamating multiple knowledge bases and data structures (e.g. relational, object-oriented, spatial, and temporal structures) when these knowledge bases (possibly) contain inconsistencies, uncertainties, and non-monotonic modes of negation. In this paper, we specify the programming environment for this framework and show that it can be used to extract and integrate information obtained from different sources of data and resolve conflicts. We will also show that it can be extended easily to integrate new knowledge bases.*

Keywords: *Incomplete Imprecise or Uncertain Information, Heterogeneous Systems and Interoperability*

1 Introduction

Integrating knowledge from multiple sources is an important aspect of automated reasoning systems. Complex reasoning tasks in the real world utilize information from a multiplicity of sources. These sources may represent data and/or knowledge about different aspects of a problem in a number of ways. Wiederhold and his colleagues [5, 6] have proposed the concept of a *mediator* – a device that will express how such an integration is to be achieved. In [1, 2, 11], Subrahmanian et al. presented a uniform declarative and operational framework for mediators, based on *annotated logics*, for amalgamating multiple knowledge bases and data structures (e.g. relational, object-oriented, spatial, and temporal structures) when these knowledge bases (possibly) contain inconsistencies, uncertainties, and non-monotonic modes of negation. The framework capitalizes on existing software products as well as application programs to develop new hybrids that are more powerful than the sum of all these products. This approach allows the programmers to use the best source available for a special task and integrate the results with the rest of the program.

In this paper, we specify the programming environment for this framework which will be referred to as the "Mediatory Programming Environment (`MPE` for short)". We show that `MPE` can be used execute external programs, extract and integrate information obtained from different sources of data and resolve conflicts in a uniform way. We also demonstrate that `MPE` is easily extendable. In other words, when a new knowledge base is added to the existing system, only simple definitions need to be made to enable the interface between this knowledge base and the existing mediatory program. Also the details of the real implementation of `MPE` are hidden from the programmer. This paper is part of a project for integrating multiple knowledge bases, and we have been developing a system called `HERMES` (short for "A Heterogeneous Reasoning and Mediator System") for the last two years. `HERMES` system will not only integrate information, but it will provide automation and help in various aspects of the integration process. The details of the `HERMES` system is described in [12].

A great deal of work has been done in *multidatabase* systems and *interoperable* database systems[7, 13, 10]. However, most of this work combines standard relational databases (no deductive capabilities). Not much has been done on the development of a semantic foundation for such databases. The work of Grant et. al. [7] is an exception: the authors develop a calculus and an algebra for integrating information from multiple databases. This calculus extends the standard relational calculus. Zicari et. al [10] describe how interoperability may be achieved between a rule-based system (deductive DB) and an object-oriented database using special *import/export* primitives. No formal theory is developed in [10]. Perhaps closer to our goal is that of Whang et. al. [13] who argue that Prolog is a suitable framework for schema integration. In fact, the approach of Whang et. al. is in the same spirit as that of metalogic programming discussed earlier. Whang et. al. do not give a formal semantics for multi-databases containing inconsistency and/or uncertainty and/or non-monotonicity and/or temporal information. Reasoning with temporal mismatches has been studied by Jajodia and Wiederhold and their colleagues [6, 5]. This framework is com-

*This work was supported by the Army Research Office under grant number DAAL-03-92-G-0225, and by ARPA/Rome Labs contract F30602-93-C-0241 (ARPA Order Nr. A716).

plementary to ours. Dubois, Lang and Prade [4], also suggest that formulas in knowledge bases can be annotated with, for each source, a lower bound of a degree of certainty associated with that source. The spirit behind their approach is similar to ours and although our framework is more general, their theory is a bit richer in one special case of our framework. Our paper however addresses a different aspect of the heterogeneous knowledge systems – that of integrating multiple knowledge bases containing (possibly) inconsistencies, uncertainty, and possibly even temporal information, and also providing a flexible and extendable programming environment.

2 An Overview of the Hybrid and Amalgamated Knowledge Bases

This paper is a continuation of a series of papers [1, 2, 11] developing the theory and practice of integrating information with the help of a "mediator". These two papers together with a third paper [8] on integrating information from deductive databases together with non-logical databases such as relational databases, auxiliary data structures and numerical constraints, are extensions of the "generalized annotated program" (`GAP`s for short) framework proposed in [9]. The `GAP` framework syntax is an extension of the logic programming. It has been proposed as a framework within which inconsistencies, temporal information and probabilistic logic can be handled in a uniform way.

The "mediator" is a program written in a special language which will be referred to as the "mediatory language" and it operates on information coming from different sources of data. Typically, these sources of data are pre-existing application programs and software packages. The programmer of the mediator who will be referred to as the "mediator developer" considers these applications as a black box. The mediator developer only needs to know what kind of information is stored in the black box at hand, what kind of operations can be performed on the existing information and what the format of the output of these operations is. Then, the mediator developer has to decide how to combine this information in order to create the mediator.
As an example, consider a developer who is writing a program which will help students choose which universities to apply to. She (or he) is aware of an expert system, which questions students about their areas of interest, goals after graduation, their GRE (or SAT) scores and etc, and comes up with a description of the candidate universities. This description contains a list of departments (or special areas in a given department) the student should consider, together with a degree of certainty of each answer. In addition to that, the developer has found a world-wide database containing information about universities such as the location, living costs, information about departments, requirements for application and so on. Let's assume that this database is object oriented. The developer wants to write a program which will ask the same questions as the expert system, and come up with a list of universities that are suitable for the student (together with the reasons why they are suitable). The program will also take into account the deadlines, student's scores and etc. The developer also decides create an option in which a student can give a list of universities that he or she had in mind for application. The developer decides to use a deductive database system to store these options.

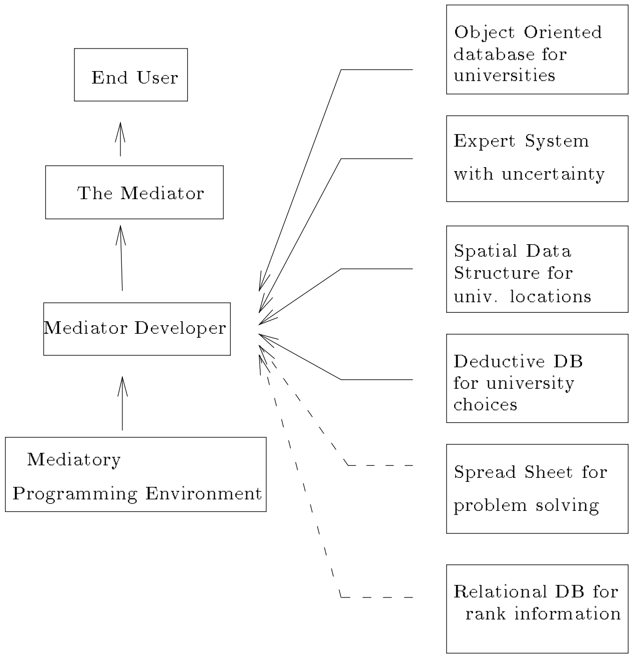

Figure 1: The mediatory programming environment

The "mediatory programming environment" (`MPE` for short) is the "interpreter" that is written to execute programs written in the mediatory language and to interface with external programs. In order for the `MPE` to be practical the following has to be true:

- The underlying language (mediatory language) must be sufficiently flexible to express any kind of information obtained by accessing/querying an external program. It must be able to handle inconsistent and incomplete data, express consensus of data coming from different sources. The amalgamated knowledge base framework developed in [11, 8] integrates information coming from deductive databases, relational databases, different data structures, numerical domains such as the linear programming problems, object oriented databases and etc. All of the properties mentioned above are satisfied in this framework.
- Different program servers may be located at lo-

cations scattered at different sites on a network. The mediator program should function as if all the servers are located where the end user is. In [2] we developed a theory for distributing the local computations of the mediator across a network and studied the properties the distributions should satisfy for the mediator to function properly even if some of the links in the network go down.

- MPE should provide a uniform programming environment to the mediator developer. Integrating a new application program should only change the mediator, not the mediatory programming environment. This ensures that MPE is easily extendable, i.e. as new application programs become available to the mediator developer, new mediators can be built on top of the existing one very easily. In addition to that, the mediator developer should not need to know to specifics of the operation of MPE. This will be accomplished by adding an interface specification facility to MPE which will be referred to as the domain specification interface.

  As an example consider the mediator for University Applications. The developer finds a data structure storing the universities in his database. Then, he wants to add a feature which will allow the user to specify which region of the current map he or she would like to go to school. In [11] we developed a unifying framework within which knowledge from several deductive databases can be integrated and the syntax of the MPE for this framework is easily extendable to any number of deductive databases. However, the theory developed in [8] to integrate information from arbitrary knowledge bases does not specify how extensibility is achieved. In this paper, we develop a new syntax for the general case.

The tasks performed by the mediator developer can be divided into two mutually exclusive subtasks. These subtasks require expertise in different areas and they can be performed by different programmers: (1) "Domain Integrator" is the person who writes up the interface between MPE and a given software. This requires some preliminary knowledge of the underlying programming language and the operating system, (2) "Mediator Programmer" is the person who uses the interface provided by the domain integrator and writes the rules that manages the interoperation between different domains. We will assume that the domain integrator and the mediator programmer is the same person for simplicity.

These ideas are shown in figure 1. As it can be seen from this figure, the end user only uses the user interface developed for the specific mediator which executes queries on the external programs as mentioned before. The mediator is written by the mediator developer using the mediatory programming environment. The knowledge databases that are mentioned in the previous section are shown in solid boxes. The last two knowledge bases shown in dashed boxes will be introduced later in the paper.

2.1 Overview of Mediatory Knowledge Systems

2.1.1 The GAP Framework

In this section, we will give the syntax of the logical predicates in the mediatory language. As mentioned above, this syntax is an extension of the "generalized annotated program" (GAPs for short) framework proposed in [9]. 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 GAP framework operates on any specified set $\mathcal{T}$, of truth values that forms a complete lattice under a given ordering $\preceq$. As an example, if the set of real numbers $\mathbf{R}^+$ is chosen as the truth value lattice, then the $\leq$ ordering is a natural ordering for this lattice. Hence, the atom $A : 0.7$ can be read as the atom A is true with certainty of at least 0.7. Thus, the atom `should_apply(smith,maryland):0.7` can be read as the student named Smith should apply to the University of Maryland with certainty of at least 70%. An annotated clause is a sentence of the form:

$$\mathtt{A} : \mu \leftarrow \mathtt{B_1} : \mu_1 \,\&\, \ldots \,\&\, \mathtt{B_n} : \mu_n.$$

where $\mathtt{A}, \mathtt{B_1}, \ldots, \mathtt{B_n}$ are atoms, and $\mu, \mu_1, \ldots, \mu_n$ are expressions denoting truth values in the given truth value lattice. It was shown in [8, 11] that the mediatory knowledge base framework can be easily extended to any complete truth value lattice.

2.1.2 Integration of Multiple Knowledge Bases

Since the mediatory language is an extension of the GAP framework, it consists of prolog-like clauses with only one atom in the head. The body of a clause contains two parts: (1) the constraint part: this part contains queries to the external programs, (2) the logical part: this part refers to atoms in the mediator. The GAP framework as proposed above is not expressive enough to handle these new types of clauses. Hence, we need to extend the GAP syntax.

Suppose we have a set of knowledge bases that are accessed in the system. These knowledge bases are referred to as constrained domains. The reason for this is that queries to these databases may be viewed as constraints over their respective domains. These constraint domains can be either one of spatial data structures, relational or object-oriented database systems, linear programming domains, numerical domains and etc. As an example, we may be using a spread-sheet for solving linear programming questions, an object-oriented database to access data information and an implementation of R-trees to solve range queries. Similarly, the domains may be more complicated and general purpose application programs.

In addition to this, we assume there is an additional database called the mediator. The mediator is a logic program defined in terms of clauses in the "mediatory language". The clauses in the mediator specify the sub-queries that need to be solved in order to solve a query and where these sub-queries should be directed. Since the mediator is itself is a program, it can thought of as another domain for another mediator. Hence, it is possible to have different hierarchies of mediators.

The basic problem that needs to be solved is how to communicate with different constraint domains. Since a query to a domain may return an answer in any format, an interchange format needs to established between different domains and `MPE` to ask queries and access the answers uniformly. In this paper, we will extend the syntax of the "mediatory language" to solve this question. In the following section we will give an example of how this is accomplished:

2.1.3 A Complete Mediatory Clause

In this section we will give an example of a mediatory clause that interoperates between two deductive databases and an object oriented database. The reader must keep in mind that these databases are in fact existing programs, so the format for the output and input is known in advance. This example is intended as an illustration of the usage of the "mediatory language". The formal definition of the syntax will be given later in detail.

```
should_apply(Student,University):1 ←
      in(true,expert:query(DB1,
         suitable_dept(Student,Dept):0.7)) &
      in(true,deductive:query(DB2,
         prefers(Student,University))) &
      in(UnivObj,oodb:equal(university_db,
         university.college.dept.name,
         Dept)) &
      =(UnivObj.university.name, University).
```

This clause shows interoperation between three databases. The expert system which is a `GAP`, the deductive database and the object oriented database. It can be read as: "The mediator will conclude that student **A** should apply to the university **U**, if the student should consider applying for the department **Dept** according to the expert system `DB1` with at least 70% certainty and **U** is in the preferred list of universities of **A** according to `DB2` and the object oriented database states that **U** offers courses in department **Dept**."

The domains are given aliases in the mediator, such as `expert,oodb,deductive`. Then domain specific information is defined for each alias separately. Hence, if `university_db` is stored in Postgres, then `oodb` is an interface to the Postgres system. When the mediator is processing the above clause, it will use the relevant domain specifications as written by the domain integrator.

Special Predicates. There are two special predicates used in the program, namely `in()` and `=()`. These predicates are executed differently in the logic than the other predicates. They are used to execute external programs and manipulate the output. To see what they mean, let us examine the above clause:

- `in(UnivObj,oodb:equal(university_db, university.college.dept.name,Dept))` will execute the selection operation "select * from `university_db` where `university.college.dept.name` = `Dept`" on domain `oodb`. (assuming that `Dept` is non-ground when this clause is being executed) The set of all answers which in this case is a set of tuples from the database `university_db`, will be stored in a temporary storage and one of these answers will be chosen and it will be unified with the variable `UnivObj`.
- `=(UnivObj.university.name, University)` will access the answer `UnivObj` which may be a complex object and unify the value of the attribute `university.name` of this object with the variable `University`. Alternatively, if `University` is specified in the processing of this clause, it will succeed/fail according to the satisfaction of the equality condition.

Execution of the Above Clause. Assume we ask the query ← `should_apply(smith,U):1` to the mediator which can be interpreted as follows: find a university `U` such that student `smith` should apply to `U` according to the mediator. Then, the following operations take place in the mediator:

- Access the expert system and run the query `suitable_dept` for `Student=smith`. This process may involve invoking an interactive process. Then, find a department `Dept` that satisfies the query ← `suitable_dept(smith,Dept):0.7`. (Note that there may be more than one department that satisfies this query. If the chosen department `Dept` does not satisfy the remaining part of the query, then the program will backtrack and find the next department that satisfies this query.)
- Access the database of preferred universities, and find a university `U_x` such that `prefers(smith,U_x)` is true in this database.
- Access the database `university_db` which is object oriented and make a call to the query formed by selection on the department name attribute. This will produce a list of universities, pick a university from this list and check if its name is `U_x` from this list.
  - If this is true, then output `U=U_x` as an answer to this query,
  - if this is not true then backtrack, and try the next option. This option may be a new university name from the list produced by the SQL query, or a new department name from the expert system.

3 Querying a Constraint Domain

The process of querying a constraint domain involves calling a program with a specific input, storing the output of the program in a temporary structure and interpreting the output. Then, only the following need to be known in advance to operate on a constraint domain:

1. Names of the data files that are accessed,
2. Operations that are going to be executed on this domain,
3. Format of the input for every operation, and
4. Format of the output, and the storage facility of the output (file, external data structures).

When a new software is being integrated into the mediator, the mediator developer (domain integrator) is faced with the following questions: (1) What are the operations that can be performed on this new software and which of these operations are going to be useful for the problem at hand? (2) How to perform these operations from another program, i.e. how to send and receive commands between this new software and an external program? (3) What is the format of the output, is it a formatted output or an object that can be accessed and copied into another system? Having solved these questions, the developer will write the necessary functions to execute these operations, obtain the output and store it in a format accessible by the mediator. We assume that a software package has either a programming language interface (like C), or it can be executed by a shell script. In the latter, the output may need to be processed to differentiate between different answers. These functions then will be linked to the domain interface module of the mediator.

3.1 The Implementation of the Special Predicates

Once the mediator developer resolves the above problems and integrates the new constraint domain into the mediator by specifying the input and output characteristics of the necessary functions, she is ready to write programs in the mediatory language that interoperate between this domain and the other available domains. As discussed earlier, this is achieved by means of some predicates provided by the `MPE`. The meaning of these are given in detail below:

- `in(Obj, Domain : Function(File, ⟨Args⟩))`: This predicate sends a command to execute `Function` defined on the given `Domain` for the file (or program) given by `File` using the `Args` which may be ground or non-ground terms. Every function is assumed to return a set of answers for different instances `Arg'` of its arguments. These sets are stored in queues which are updated by the process initiated from this call every time a new answer (or a set of answers) is found. Then, the mediator chooses one answer (possibly the first) from a relevant queue and copies this answer to `Object`. Hence, `Object` is complex object whose structure is specified by `Function`. This predicate succeeds as many times as there are different answers to `Domain : Function(File, ⟨Args⟩)`. Each time it backtracks, it will fetch another answer and process the rest of the program using the value of this answer.
- `=(Object.Attr`$_{i_0}$`.Attr`$_{i_1}$`....Attr`$_{i_k}$`, Value)`: This is an identity function, it succeeds when the first and second arguments are unified to the same object (if `Value` is a variable, it is instantiated to the given attribute). Both `Object.Attr`$_{i_0}$`.Attr`$_{i_1}$`....Attr`$_{i_k}$ and `Value` may be complex objects, hence the values of all their arguments must also be equal. An attribute of `Object` may take one of several values, then the predicate will succeed once for each value. This will be discussed in the next example. Similar predicates are implemented for conditions $\leq, <, >, \geq, \neq$. As opposed to the `=()` predicate, these are only practical when one of the arguments is instantiated. Otherwise, there are infinitely many answers satisfying each condition and it may not be possible to enumerate them all for complex objects.

Example 3.1 (Handling Object Hierarchies) Let's see how these are used with an example. Recall the object oriented database which stores the information about universities. Assume the student is interested in the linguistics courses offered in a university. Linguistics courses may be offered in linguistics departments, computer science departments. We want to find all the departments in the University of Maryland that offer courses in linguistics. Hence, we make a selection on the university name using the `in()` predicate. This will produce an output which is an object that contains the list of colleges for a given university, list of departments for every college and a list of courses for every department. We then find the correct departments. The complete query is given below:

```
← in(Univ, oodb:equal(university_db,
           university.name,``UMD'')) &
  =(Univ.college.dept,Dept) &
  =(Dept.course.name,``Linguistics'') &
  =(Dept.name, Department).
```

Assume both linguistics and computer science departments offer courses in linguistics. Assume `Univ` contains only the computer, physical and mathematical sciences (CMPS), and arts and humanities (ART) colleges. The following are the departments in these colleges:

College	Departments
Computer, Physical and Mathematical Sciences	Computer Science Mathematics Physics
Arts and Humanities	Art Linguistics

Assume that the above query is asked to the mediator. We are interested in the value of the variable `Department`. Then, assuming that we iterate until we exhaust all the possibilities, the execution of this query will be as follows:

- `Univ` is selected using the `in()` predicate. Suppose there is only one such university.

- Univ.college points to the first college, i.e. CMPS.
  - Univ.college.dept points to Computer Science, all the courses offered in this department are fetched one by one by the =() predi-
    cate and checked whether ''Linguistics'' is among them. Since ''Linguistics'' is among the courses offered, the last predicate is reached, Department is set to ''Computer Science'' and the query succeeds.
  - Suppose we continue processing. Then we backtrack, and try to re-satisfy the last two predicates. Since Dept.name has a single value, =() succeeds only once, hence we backtrack to the second predicate. Now, Univ.college.dept points to the Mathematics department. All the courses offered are scanned, since ''Linguistics'' is not one of them the third predicate fails for all and the execution backtracks to the second predicate.
  - Univ.college.dept points to the Physics department, the third predicate fails for all courses as in the Mathematics department. Since, there are no more departments in the college of CMPS, the second predicate is reevaluated, this time the next college is chosen.
- University.college points to the Arts and Humanities college and the ''Linguistics'' department is found as another answer.

4 Example Domains

4.1 Relational Database Domain

Now, we will show how certain and most popular domains can be expressed in our framework and how the basic operations can be expressed. We provide the basic operations for the relational database domain first. The main functions that can be provided in the mediatory language are selection, projection and of course join operations. For the selection function, the mediator developer has two options. He can either provide the selections on given attributes using the comparisons less than, less than or equal to, greater than, equal, not equal and etc. Alternatively, he can simply write shell scripts which execute complex SQL statements. Below, we give examples of both of these options.

$\mathtt{in(Obj, relation : select\_cond(File, Attr, Value))}$
$\mathtt{in(Obj, relation : project(File, \{Attr_1, ..., Attr_n\}))}$
$\mathtt{in(Obj, relation : join(File_1, Attr_1, File_2, Attr_2))}$
$\mathtt{in(Obj, relation : sql\_stmt(File, Args))}$

The reader must keep in mind that relation is not a static name given to all the relational domains. Since all such domains will have different domain specifications, they will be given different names by the mediator developer and these names will be substituted for relation whenever these domains are used. In our implementation, if the two domains use the same database system, the they may share the same domain name, but the files accessed are going to be different.

Example 4.1 Recall the university application guide program that was presented in the first section of the paper. In addition to the domains which were depicted in figure 1, the mediator developer wants to integrate a relational database containing rank information about top 50 universities for different departments. The mediator developer wants to find out if a given university is in the top 25 for the given department. Assume that every department is stored in a different table, these tables are declared in the mediator using the predicate relation_file. The mediatory clause that can be written to find this answer is as follows:

```
in_top_25(Univ,Dept):1 ←
        relation_file(rank_db,Dept,DRank):1 &
        in(U,relation:equal(DRank,
           university_name,Univ)) &
        ≤ (U.rank, 25).
```

A query of the form ← in_top_25(Univ,Dept):1 (find if the department Dept of the university Univ is in top 25 according to the mediator), is executed as follows: the relation_file predicate finds the correct relation corresponding to the department Dept. Then, the tuple corresponding to the university Univ is selected from this relation with the in predicate. If this university is not in top 50, then this clause will fail and the whole query will fail. Otherwise, the tuple corresponding to Univ will be unified with U. Next, the $\leq$ () predicate will check if the rank attribute is less than 25, than the query will succeed, otherwise it will fail.

4.2 Spatial Domain

There are many special data structures for storing spatial information. These data structures are generally designed for a specific purpose and they provide very efficient algorithms for storing, accessing and querying information for this purpose. R-trees, quadtrees, Z-ordering are among the many examples of such data structures. In this section, we will present the necessary functions for accessing 2-dimensional PR-quadtrees. Some of the useful operations defined for quadtrees storing 2d points are horizontal,vertical which find the points that lie in a horizontal or vertical slice, box finds all the points within a given box (given the coordinates of two corners), and range which finds all points within a circle given the center and the radius. Below we give the functions for these operations, similar functions may be defined for different operations.

```
in(Obj,quadtree:horizontal(File,Lower,Upper))
in(Obj,quadtree:vertical(File,Left,Right))
in(Obj,quadtree:box(File,X1,Y1,X2,Y2))
in(Obj,quadtree:range(File,X,Y,Radius))
```

Example 4.2 Recall the university applications program. The mediator developer intends to use a spatial data structure storing the locations of the universities. The data structure is a PR-quadtree and it stores the longitude and the latitude of the university locations. Assume, she wants to find out all the universities in the north-east states of the US. Assume that the quadtree is stored in the file locations. Then, the following clause in the mediator can be used:

```
north_east(Univ):1 ←
        in(Temp,quadtree:box(locations,
               85°,70°,37°,50°)) &
        =(Temp.university, Univ).
```

4.3 Linear Programming Domain

In this section, we will give an example of a numerical domain. Implementations of linear programming can be found in many commercial products. The problems that will be solved for the mediator can be written as functions together with the domain specifications. When these functions are called in the mediator, the variables in the problem are set to the specified values. The mediator then executes the corresponding problem using the available software product. Results of the problems contain the values of variables as well as the maximum or minimum values of objective functions.

```
in(Solution,tableau:Function(File,Args))
=(Solution.Variable,Value))
=(Solution.Maximum,Value))
=(Solution.Minimum,Value))
```

Example 4.3 Now, assume the mediator developer decides to use a linear programming software in the mediator. He wants to create an option in the mediator where the student can specify the amount of money she wants to spend on the applications. In addition to this, the mediator calculates the number of requirements the student satisfies for a given university. Obviously, the more requirements of a university the student satisfies, the better are her chances of getting accepted. Hence, given the choice of three (or ten) universities, the mediator will find the best set of universities the student may apply to within the given budget. The maximization problem that can be written for this problem is as follows:

function application(Fee,Req,Budget)

$$\textbf{maximize} \;\; \mathtt{Req}[1]*\mathtt{x}_1+\mathtt{Req}[2]*\mathtt{x}_2+\mathtt{Req}[3]*\mathtt{x}_3$$
$$\textbf{subject to} \;\; \mathtt{Fee}[1]*\mathtt{x}_1+\mathtt{Fee}[2]*\mathtt{x}_2+\mathtt{Fee}[3]*\mathtt{x}_3 \le \mathtt{Budget}$$
$$\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5 \in \{0,1\}$$

The array Req stores the number of requirements satisfied, Fee stores the application fee and Budget is the student's budget. The solution to this problem will be the total number of requirements satisfied. When one of the $\mathtt{x}_i$ is set to 1, this means the corresponding university is selected. An example of the way this problem can be used in the mediator is given below:

```
within_budget(Budget,Fees,Reqs,U1,U2,U3) : 1 ←
    in(S,tableau:application
            (lp,Fees,Reqs,Budget)) &
    = (S.x1,U1) & = (S.x2,U2) & = (S.x3,U3).
```

4.4 More Complex Domains - Application Dependent Choices

Besides these more obvious choices, domains can be any kind of complex application programs. As an example, we may be using Xarchie to locate files related to a given subject, or the catalogs of the Library of Congress to search for a book. In this case, the input and output protocols have to be sorted out. Assuming that this problem is solved, the mediator will most probably not have access to the internal objects. In such a case, the output will be obtained from the user interface of these applications. For text output, the domain integrator will have to write a program which will analyze the output to extract important pieces of information and store them in a data structure. In the case of the non-text output such as graphical interfaces, if the windows contain only text data, a program to recognize the text must be called. Hence, tradeoffs between the cost of these operations (like the integration effort, running time) and the value of the information obtained becomes more significant for complex domains.

In addition to the domains mentioned above, it is possible to have application dependent domains. For example, the application fees stored in the database may refer to different currency units, a program that converts these values to US Dollars (or to the currency the student prefers) must also be executed. Of course this program should find and use an up-to-date currency exchange rate table every time it is executed. Similarly, if the mediator has access to temperature information which is stored in Celsius and/or Fahrenheit measures, then these have to be converted to some appropriate medium.

5 Implementation

We currently have a working implementation of the mediatory programming environment that runs on Sun Sparc stations (under Unix) as well as on the IBM-PC platform under DOS/Windows 3.1. The following domains are integrated on both platforms: relational data in multiple heterogeneous formats in ASCII text files, spatial data in the form of PR-quadtrees, linear programming problems solved by an in house problem solver, text data indexed by a B-tree storing the list of keywords for each document (inversion of terms), and pictorial data. In addition to these a path planning package developed by the US Army for optimal path planning across "free" (i.e. mountainside, countryside, etc.) terrain, Ingres DBMS (partially) are integrated on the Sun platform, and Borland's Paradox DBMS is fully integrated on the PC platform. We have focused our attention on integrating more non-traditional knowledge bases to our system together with commercial systems.

MPE is implemented in C (currently about 12,000 lines of code). Most of the mediatory language has been implemented, including general aggregate data structures, in() and =() (as well as other comparison) predicates. The standard data structures include records, arrays, sets and arbitrarily nested combinations of these structures. We will also provide built-in functions, such as append() for arrays and union() and intersect() for sets, for constructing aggregate data structures. These very general data structures facilitate the integration of custom software as well as standard packages. Alternatively, domain functions can pass arbitrary files (text or binary) to other functions in the same or compatible domains. Also, when a file name is returned as part of an answer to the user, an action module asks whether the user wishes to perform an action based on the type of the file. Text and image files may be viewed; sound clips may be listened to; shell scripts may be run. A uniform graphical interface currently under development on both platforms helps the user ask queries and process answers.

So far, the domain functions integrated into our system write their output into a file which in turn is read by the mediator. To add a new domain to the mediator, the domain integrator needs to update the domain catalog (which is a text file), add two lines to a module that calls an interface module based on the domain name, and provide an interface module for the domain. A template (i.e., text file to be completed) is provided for a generic interface module. In addition, templates are provided for classes of domains, such as relational databases or quadtrees, with predefined function names and parameter types for more efficient and consistent integration of a new domain within a class. The integrator has the flexibility to modify a template as desired. The integrator uses a few C functions provided by the mediator software to extract a domain function term's components at run time. These C functions return standard C values (integers, strings, etc.) while keeping the internal representation of the domain function term hidden. If necessary, the integrator also converts the output format of the domain function to the mediator's standard data format. Overall, a uniform logical framework is provided down to the level of the calling of a domain function; at that point it is up to the integrator to write the necessary block of C code. In the future, we will provide a graphical toolkit to further ease the integration of a new domain.

6 Conclusions

In this paper, we developed a programming environment for mediators which is very powerful and easy to use. We showed how this environment can be used to interoperate between a relational database, an object oriented database, a spatial data structure, different deductive databases including an expert system, a numerical domain such as a linear program solver and more complex domains. We demonstrated how the execution of an external program can be abstracted to few simple predicates in the logic of mediators. We showed that very small effort is needed to facilitate the integration of a new knowledge base into the system. Next steps in our implementation is the development of intelligent caching schemes [3] for efficient query processing, general purpose toolkits both for the domain integrator and the mediator programmer [12], and the integration of temporal information, different domains.

References

[1] S. Adalı and V.S. Subrahmanian. *Amalgamating knowledge bases, II: Algorithms, data structures and query processing.* Technical Report CS-TR-3124, Computer Science Department, University of Maryland, Aug 1993. Submitted for journal publication.

[2] S. Adalı and V.S. Subrahmanian. *Amalgamating knowledge bases, II: Distributed mediators.* To appear in International Journal of Intelligent and Cooperative Information Systems(IJICIS), December 1994.

[3] S. Adalı and V.S. Subrahmanian. *Intelligent Caching in Mediatory Knowledge Bases.* Under preparation.

[4] D. Dubois, J. Lang and H. Prade. (1992) *Dealing with multi-source information in possibilistic logic.* Proc. 10th European Conf. on Artificial Intelligence. Wiley, 1992.

[5] G. Wiederhold, S. Jajodia, and W. Litwin. (1991) *Dealing with granularity of time in temporal databases.* Proc. 3rd Nordic Conf. on Advanced Information Systems Engineering, Lecture Notes in Computer Science, Vol. 498, pages 124–140. Springer-Verlag, 1991.

[6] G Wiederhold, S. Jajodia, and W. Litwin. (1993) *Integrating temporal data in a heterogeneous environment.* Temporal Databases, Jan 1993.

[7] J. Grant, W. Litwin, N. Roussopoulos and T. Sellis. *An algebra and calculus for relational multidatabase systems.* Proc. First International Workshop on Interoperability in Multidatabase Systems, pages 118–124. IEEE Computer Society Press, 1991.

[8] J. Lu, A. Nerode and V.S. Subrahmanian. *Hybrid knowledge bases.* To appear in IEEE Trans. on Knowledge and Data Engineering.

[9] M. Kifer and V.S. Subrahmanian. *Theory of generalized annotated logic programming and its applications.* Journal of Logic Programming, 4(12):335–368, 1992.

[10] R. Zicari S. Ceri and L. Tanca. *Interoperability between a rule-based database language and an object-oriented language.* Proc. First International Workshop on Interoperability in Multidatabase Systems, pages 125–135. IEEE Computer Society Press, 1991.

[11] V.S. Subrahmanian. *Amalgamating knowledge bases.* (1994) ACM Trans. on Database Systems, 19, 2, pps 291–331.

[12] V.S. Subrahmanian S. Adalı A. Brink R. Emery A. Rajput R. Ross T. Rogers and C. Ward. *HERMES: A Heterogeneous Reasoning and Mediator System.* Under preparation.

[13] W.K. Whang S. B. Navathe and S. Chakravarthy. *Logic-based approach for realizing a federated information system.* Proc. First International Workshop on Interoperability in Multidatabase Systems, pages 92–100. IEEE Computer Society Press, 1991.