IDEA: Intelligent Data Retrieval in Prolog

Cristina Ruggieri, Mirko Sancassani
DS logics S.r.l.
Viale Silvani, 1 - 40122 Bologna (Italy)
email: cxr@dslogics.it, mirko@dslogics.it

July 5, 1994

Abstract

This paper describes an intelligent data retrieval system, designed as part of a general-purpose decision support system, named IDEA. This system is an attempt at formalizing the interaction between the data logical view and its real physical organization in the databases. The formalization chosen comes from expert-system knowledge representation technology. Its implementation has been realized using an object-oriented library of SICStus Prolog.

1 IDEA: An Overview

IDEA is a decision support system designed for a manager in the need of computing and analyzing statistical data relative to his decision making domain.

Two sets of problems arise in the design of this kind of systems. The first has to do with interfacing input data coming from different databases and with the construction of a set of relevant management indicators. The second set of problems is strictly related to the application domain. It concerns the formalization of methods and rules to support a correct interpretation of the computed indicators and thus the final decision making process.

The current development of IDEA has solved most of the problems in the first set. It has built an intelligent data retrieval system, which is capable of representing and computing knowledge about a set of domain-specific management indicators. The analysis and interpretation of these indicators will be based on expert system technology, and it is currently under design and development.

As a result, this paper will report our experience in the design and implementation of only the first component of the final decision support system. This component, however, already contains a large portion of the "intelligence" needed for the data-analysis component.

Interfacing, merging and elaborating data coming from several and heterogeneous sources represents an everyday problem for people who work in database applications. People who need to produce statistics, for example, must often combine quantities

stored in different systems and located in different sites. This task may become quite difficult since different database management systems (DBMS) have different access procedures and even the same data, when stored in different databases, may have multiple formats and encodings.

In many applications, the final user does not have sufficient computer knowledge, or does not master all the names and organizations of the databases he needs to query. The solution which is usually adopted and which is often responsible for delays and misunderstandings, is to consult database experts.

As an answer to all these problems, IDEA is an attempt at formalizing the interaction between a logical view of the data, which is used in the computation and decision-making process, and its real physical organization in the databases. The formalization chosen comes from knowledge-representation technology imported from expert systems.

More specifically, IDEA provides the application-domain manager with a uniform framework to define, elaborate and report statistical computations based on data with heterogeneous nature and organization. IDEA users can enter data queries, which may involve statistical or mathematical computations, without any knowledge of the databases which will answer the query.

IDEA then acts as an interpreter between the application domain and the databases. It knows both application-domain and technical jargon. It understands what the user wants to compute and it knows how to solve the problem by consulting its knowledge bases.

Special efforts were put in the design of an easy-to-use, highly interactive and robust graphical interface. Multiple solutions to a query, ambiguous or inconsistent query specifications, substitution of unavailable data, are all problems solved asking the user for further information. To make the interaction easier, the system works in a *propositive* manner: the user can always choose among different IDEA proposals.

Finally, IDEA allows the decision making manager to update and extend the existing knowledge bases, through a syntax-directed editor of computable objects. This is motivated by the recognition that the ability to modify the knowledge bases is an essential feature of decision support systems, particularly in the data analysis phase. We have verified, however, that even in the early stage of decision making, when indicators are computed, the choice of which indicators are relevant is a major influencing factor of the final outcome. Such choice must therefore be controllable by the decision-making manager.

2 The System Architecture

The design of IDEA architecture for intelligent data retrieval was strongly influenced by the two major requirements mentioned in the introduction:

- The decision maker, identified as IDEA end user, was to be able to define and use objects in the system by means of high-level descriptions, based solely on a simple, formalized language, strongly related to the application domain.

- The availability of new databases or the substitution of old ones were recognized as frequent events in real applications. These events were therefore to involve only a minimal change in the system. That meant leaving the definitions of logical object representations and of management indicators unchanged.

As a consequence of these requirements, IDEA architecture reflects a complete independence between knowledge about the application domain, that we call *logical* knowledge, and knowledge about available data and DBMS access methods, that we call *physical* knowledge. We can see the system as built of two descriptive layers, a *logical* layer and a *physical* layer.

A computation is an inference process which attempts to translate information of the logical layer into the corresponding information in the physical layer. Upon a user's query, IDEA dynamically finds the set of operations and data in the *physical* layer which best fits, according to a set of rules, the logical description given by the original query. The logical and the physical layers are organized in a set of knowledge bases, each representing a specific system functionality. A similar architectural solution can be found in [6].

Besides these kernel layers, the architecture includes a graphical user interface, a server for interfacing remotely connected databases, and an internal relational database, locally connected to IDEA knowledge bases.

IDEA graphical interface is a set of tools for the interaction with the computational layers. This part of the system is completely general. However, each tool has the capability, consulting the knowledge bases, to become instantiated to a specific domain-based tool, and to communicate using application domain jargon. This is the only IDEA layer directly visible to its end-user.

IDEA Remote Query Processor is a server module which manages the communication between IDEA and remote database systems. It accepts a database query, satisfies it and stores the resulting tuples in a temporary relation in IDEA internal database.

The following sections of this paper will discuss the design and implementation of the logical and physical layers of the architecture. A description of the graphical interface can be found in [8], while issues related to remote database access are described in [5].

3 A specific application: Decision Making in Health Care

Even though IDEA was designed as a general, domain-independent system, throughout the rest of the paper we will refer to one of its current applications: aiding the decision making process in the management of public health care. The purpose is twofold. One the one side, a specific application domain allows us to show how real problems were solved by IDEA. On the other side it helps avoiding an excess of abstraction in describing the system.

Health-care management involves identifying the major sources of health problems

in a given population, and making relevant decisions on how to best use the available human and financial resources to solve these identified problems.

Management indicators are, in this case, epidemiological indicators, such as mortality rates, health-risk indicators, hospital efficiency measurements, cost/effectivness indicators of planned health-care policies, etc.

Data values are retrieved from a set of standard databases, such as the national population and mortality databases, hospitalization reports collected by local health-care districts, pharmacological prescription reports, etc.

4 Implementation Issues

The architecture we designed, lead quite naturally to an object-oriented implementation. Objects allowed us a concise structural representation of IDEA domain knowledge and of the main relationships among epidemiological entities. Furthermore, using an object-oriented language we were able to construct prototypes of given classes of objects, for example an indicator. We then defined new objects in that class by a simple and concise definition-by-specialization technique.

In IDEA, however, object-oriented facilities alone were not adequate to express the functionalities of the "intelligent" parts of the system behaviour. A language without any sort of inference engine is in fact ill-suited to represent domain-dependent inference rules and reasoning strategies.

In the context of knowledge representation languages, objects are often identified with the notion of *frames*, following [7]. Indeed, frames are the key structuring device in most of the commercially available Knowledge Base Management Systems [3], [4]. Most of these systems integrate the frame paradigm with production rule languages, to form hybrid representation facilities, cabaple of logic inference.

Since in IDEA logic inference plays a central role, we decided that the most direct implementation of IDEA specification and architecture could be obtained by using a logic programming language, like Prolog, augmented with object-oriented facilities necessary to correctly structure IDEA knowledge bases. Indeed Prolog has a well-understood operational and declarative semantics, which makes the semantics of IDEA reasoning rules clear and easy to debug. Furthermore, with Prolog, we were able to use the same language to implement the intelligent parts of the system as well as its standard computational parts.

This last point is what mostly distinguishes a solution based on an object-oriented Prolog, from the one adopted by standard KBM systems, which often need to resort to languages like Lisp of C, to provide adequate computational facilities to their knowledge basis.

IDEA is implemented in SICStus-Objects, an object oriented library of SICStus Prolog [1], in conjunction with the APPEAL programming environment [2], for the user interface modules.

Thanks to the large number of platforms supported by SICStus Prolog (and consequently by APPEAL), IDEA runs on any Unix system connected to an X-window screen. The internal database used in the current implementation is INGRES. How-

ever, we envision no major difficulties in porting our Prolog/SQL interface to another relational database which supports standard SQL.

5 The Knowledge Bases of the Logical Layer

The main purpose of IDEA logical layer is to define a set of high-level logical views of computable objects. We distinguish between two types of objects: *variables*, which are logical descriptions of tuples directly obtainable from one of the databases, and *indicators* which are tuples obtained as operational combinations of one or more variables, eventually produced by different databases.

Both *variables* and *indicators* represent domain-specific knowledge which describes the logical entities of a specific application. Both are qualified by *attributes*, whose values are used to select variable values during a computation. The difference between variables and attributes is only logical: in the databases they may both correspond to fields of a relation or to some aggregation of fields.

The logical layer implements also the *operational* knowledge needed to compute *indicator* values. This operational knowledge is completely independent from the application domain, except for possible constraints that an application may require on some operations.

5.1 Variables

From a technical point of view, *variables* and *indicators* are tuples, whose first element is identified as the *variable* or *indicator* value, and whose other elements represent attribute values. For example, in the epidemiological field, a typical variable is `population`, qualified by the attributes `place`, `time`, `age` and `sex`. A value of this variable represents the number of people of a given `age` and `sex` who live in a given `place` at a given `time`. The SICStus object code which implements the epidemiological variable `population` is the following:

```
population(AttList) :: {
   super(variable) &

   name(population) &

   output_default(hystogram, [time, place, sex, age]) &

   value(Tuple) :-   /* external interface */
      self :: value(Tuple, []) &

   value(Tuple,Inst) :-  /* internal interface */
      super <: value(AttList, Tuple, Inst) &

   value(Op, Tuple, Inst) :-          /* Op is a statistical operator */
      super <: value(AttList, Op, Tuple, Inst) &

/* attribute order name and type (granularity ) */
```



```
   attribute(1, place, city)  &
   attribute(2, age,   year)    &
   attribute(3, sex, sex)   &
   attribute(4, time, year)
}.
```

where `::` is the message sending and `<:` the message delegating operators. As the above definition shows, variables (as well as indicators) are computed by sending a `value` message to the corresponding object in the knowledge base:

```
<variable_name>(AttList) :: value(Answer).
```

`AttList` may contain input values for one or more of the variable (indicator) attributes. No value specified for a given attribute means, by default, that that attribute does not constrain the value. In this case, the variable (indicator) is computed for every known value of that attribute.

The object `population` defines three methods to solve a `value` message. The first is the external interface. It just expands the call with an extra parameter. This parameter `Inst` represents possible instantiations of attribute values, resulting from IDEA's internal calculations. The second aswers internal messages sent to the variable by other knowledge bases. Finally, the third method answers a `value` message from the operator knowledge base. In IDEA computations are generally performed in Prolog. However, there are some cases in which an operator could be directly applied by the database on its raw data, before returning the tuples to Prolog. The operator knowledge base recognizes such cases and uses the third `value` rule to send a message to an appropriate data dictionary element. The complete definition of `value` is inherited by the `variable` object (`super` in the code).

The task of the `variable` object is to translate a `value` query into the corresponding SQL query for an appropriate database. This is done in two steps. The first step translates the query into a complete logical description of the variable, and finds an appropriate database to solve it. If the user has not explicitly selected a database, IDEA can backtrack over all the available databases that are capable of answering the requested query. The second step, performed by the *data dictionary* knowledge base (see section 6), translates the logical query into SQL, according to the characteristics of the database it represents.

5.2 Attributes

The *attribute* knowledge base is used during a computation to obtain information about attribute values and types. Attributes are described as logical objects, independent from their actual codification in the database relations. The translation between attribute logical representations and their corresponding database fields is performed by the data dictionary knowledge base.

Attribute value types are mostly numeric (for example the value 1993 for the type `year` of attribute `time`). Some are enumerated types represented by string constants. Different types for the same attribute may be related to one another by aggregation. For example, the value type of a `place` attribute can be `city`, `province`,

health-district (usl) or region. Clearly province data can be computed by grouping together city data belonging to the same province. In IDEA's epidemiological application an attribute type is known as its *granularity*. The SICStus object code for the place attribute is the following:

```
place::{

name(place) &
super(attributes) &

 % available granularities
granularity([city, usl, province, region]) &

 % granularity compatibility
gr_result(X,X,X)  :- ! &
gr_result(city, usl, usl) &
gr_result(usl, city, usl) &
...

 % available output codifications
output_type(city,      [code, name]) &
output_type(usl,       [number]) &
...

 % used to estimate number of answer tuples
different_values(city, 341) &
different_values(usl, 41) &
...

 % internal database relation storing granularity correspondence
granularity_relation(gplace_cupr) &

ingres_type(city, integer) &
ingres_type(usl, integer) &
...

dynamic codifica/5 &

load_city_cod :-
  % LogicSQL query to INGRES
 user:lsql(select [cplace_c.name:Name,  cplace_c.code:StandardCode]
           from [cplace_c]),
 <:retractall(codify(place, city, name, Name, StandardCode)),
 <:retractall(codify(place, city, code, StandardCode, StandardCode)),
 <:assertz(codify(place, city, name, Name, StandardCode)),
 <:assertz(codify(place, city, code, StandardCode, StandardCode)),
 fail &

load_city_cod.
```



```
 % codify dictionary
codify(place, city, name, 'PIACENZA', 33000) &
codify(place, city, code, 33000, 33000) &
...
codify(place, region, name, 'EMILIA ROMAGNA', 8) &
codify(place, region, code, 8, 8) &

}.
```

In IDEA, the same attribute value can be expressed by or to the user under different codifications. For example `place(city:bologna)` represents the same information of `place(city:37006)`. The attribute knowledge base implements a dictionary to translate attribute values into and from a logical standard codification which is understood by each object in IDEA knowledge bases. Through this standard codification all objects in the system can understand each other and exchange information.

The values of each attribute type (*granularity*) are stored in the internal database, and inserted in the appropriate attribute knowledge base at IDEA startup time. This solution allows to mantain the Prolog code independent from the codification values which can be easily updated by the system administrator, using standard database tools.

5.3 Indicators

An indicator is the result of an operation which generally combines data originating from different databases. Its knowledge base must specify the variables and the statistical or mathematical formula necessary to compute it. An example of indicator is the `mortality_rate` of a population, computed dividing the `mortality` variable (number of deaths) by the `population` variable. The indicator is qualified by the attributes `place`, `age`, `sex` and `time`, which we have already seen for `population`, with the addition of `cause`, recording the cause of death. Here is the SICStus objects implementation of the epidemiological indicator `mortality_rate`:

```
mortality_rate(Term) ::
      {
       super(indicator) &
       name('Mortality Rate') &

       variables([mortality,population]) &

       output_default(table, [time, place, cause, age, sex]) &
       value(Tuple) :-
         self :: value(Tuple, []) &

       value(Tuple,Inst) :-
         operator :: value(mult(rate(
                                  mortality(Term),
                                  population(Term)
                                  ),
```

```
                                ),
                        1000), Tuple1, Inst),
        Tuple1 =.. [result|List],
        Tuple =.. [mortality_rate|List] &

        attribute(1, place, city)  &
        attribute(2, age,   year)  &
        attribute(3, sex,   sex)   &
        attribute(4, time,  year)  &
        attribute(5, cause, ICD-IX)
}.
```

Each indicator sends the `value` messages it receives to the `operator` object, which translates the message into a computable operation for the operator `mult` and then sends the translated message to this operator.

The *indicator* knowledge base also implements a set of predicates called by the user interface to check the semantic correctness of a query. Some of these predicates are general-purpose checks, others are strictly related to the epidemiological domain. An example of a general-purpose semantic check is the predicate `check/3`, which implements the compatibility of attribute types (*granularity*). In IDEA, this otherwise standard check is quite complex, since even a logically correct attribute granularity, may not be computable from the available databases. For indicators, the semantic check must first determine the logical correctness of the specified granularities, starting from the granularities of the variables computing the indicator. Then it must verify the availability of such granularities in the connected databases.

This search is difficult, since the same variable may be computed from different databases. For example, if an indicator is computed with three variables and each variable by two databases, the algorithm must examine all eight possible combinations of databases, to find the best one for solving the initial query.

Another interesting semantic predicate is `check_tempo`, used to check the compatibility of data with respect to their temporal attributes. In epidemiology in fact, it is possible to combine databases referring to different temporal data. For example the indicator `mortality_rate` in 1993 could be computed using `mortality` data of 1993 and `population` data of a recent year, but non necessarily 1993.

The `check_tempo` algorithm determines, according to a set of epidemiological inference rules, the best available data, from a temporal point of view. It is clearly a predicate strongly related to a specific application domain.

5.4 Operators

To conclude this section, we will briefly discuss the only domain-independent knowledge base of the logical layer : the *operator* knowledge base. This knowledge base provides two important functionalities: the translation of the standard functional syntax, which describes mathematical and statistical calculations, into Prolog predicates and the implementation of operators over constrained values, represented by *variable* and *indicator* tuples.

The result of applying an operator `op`, which takes in input two variables (or two indicators) `var1` and `var2`, is a new variable (indicator) `var3`. Its value is the result of applying `op` on the values of `var1` and `var2`. Its set of attributes is the union of the set of attributes of `var1` and `var2`. For the operation to be meaningful, values of common attributes in `var1` and `var2` must be the same, i.e. `var3` must satisfy the same constraints as `var1` and `var2`.

Following is an example of the implementation of an arithmetical operator, which are the simplest set of IDEA operators:

```
div :: {
        super(op_arith) &

        name(div) &
        value(List, Answer, Inst) :-
             super <: value(List, Answer, Inst) &

        execute_op(Var1, Var2, Result) :-
             (Var2 == 0 -> write(' Attention : Attempt to divide by zero'), nl,
                           Result is 0  /* for simplicity */
                         ;  Result is Var1/Var2 )
    }.
```

In general, operators are computed by sending a `value` message to the generic `operator` object in the knowledge base. This object translates the functional-style syntax of a `value` query into the Prolog representation understood by single operators. Then it delivers the translated `value` query to the appropriate operator object (for example `div`). The algorithm implemented by the general `operator` object, first computes the value of the first operand, passing to this computation all external(input) constraints. It then computes the second operand using any constraint (attribute values) derived from the first computation. Finally it extracts the values from the two resulting tuples, applies the arithmetic operator to these values, and produces a result tuple.

6 The Knowledge Base of the Physical Layer

The main purpose of the IDEA physical layer is to map the logical description of an IDEA variable into the low-level descriptions and procedures, which compute it. Thus, the physical layer provides all the knowledge bases of the logical layer with a uniform method to access data stored in the databases connected to IDEA. This task is performed by exporting to the logical layer simple views of data that can be directly used to compute variables. Each view codifies the knowledge needed to retrieve a variable value: DBMS name, database name, relation and field names, access procedures, knowledge about field codifications.

The components of this layer are all elements of the *Data Dictionary Knowledge Base*. A data dictionary element defines a set of logical views of the fields and relations of a specific database. Each view corresponds to an IDEA variable description and it

looks like a set of Prolog clauses for the predicate `value`. A data dictionary implements each variable view by dynamically mapping logical constraints represented by attribute values into different SQL queries. Furthermore, even though the meaning of the field content of different databases may coincide, their value codifications may still differ substantially. Therefore, a data dictionary knows how attributes are coded in the relations of the database it interfaces, and can translate physical attribute values into or from logical codifications.

We have already described how a variable is computed by sending a `value` message to a data dictionary element. Here we will look closely at the implementation of the objects which answer these messages.

The following example shows a schema of the `value` predicate implementation in the data dictionary element called `regional_population`. The example shows how the data dictionary knowledge base implements its main tasks.

```
regional_population::{

value(population, var(Var),
                  time(GraT:Time),
                  place(GraP:Place),
                  age(GraA:Age),
                  sex(GraS:Sex)) :-
  <:check_gran(population, place(GraP:Place)), ...
  <:solve_code(sex, GraS, Sex, CSex), ...
  ::var_or_atomic(population, sex, CSex), ...
  ::relation(1, time(Time), Rel), ...
  ::field(1, CSex, MaleOrFemale),
  /* LogicSQL query  */
  user:lsql(
          select  [sum(Rel.MaleOrFemale):Var,
                   place_gra.GraP:Place,
                   age_gra.GraA:Age]
          from    [Rel, place_gra, age_gra]
          where   [Rel.age = age_gra.year,
                   Rel.cod_com = place_gra.city]
          groupby [age_gra.GraA,
                   place_gra.GraP],
          SQLQuery,
          VarList),
  ::execute_query(SQLQuery, VarList),
  <:solve_code(sex, GraS, Sex, CSex) &
...
}
```

A `value` message sent to a data dictionary must always specify a variable name, while attribute values can be left unspecified. When an attribute value is present, it specifies conditions that must be satisfied by the returned result. The variable value is returned into an initially unbound Prolog variable.

For example, the computation of the epidemiological variable `population` for all ages and sexes in the year 1993 and in the towns whose national codes are 37006 and 37007, can be activated by sending to the data dictionary `regional_population` the following message:

```
value(population,var(X),time(year:1993),place(city:[37006,37007]),age(year:A),sex(sex:S)).
```

This call uses Prolog backtracking to return all the population tuples satisfying the requested constraints. For example:

```
value(population,var(8),time(year:1993),place(city:[37006,37007]),age(year:0),sex(sex:m)).
value(population,var(9),time(year:1993),place(city:[37006,37007]),age(year:0),sex(sex:f)).
...
```

Thanks to the data dictionary knowledge base, all IDEA logical components can interact with variable values as if they were a set of Prolog facts.

The data dictionary takes care of all the physical details about database, relation and field names, about operations on fields, about access procedures to local or remote databases, about codifications to and from standard attribute values.

The heart of a data dictionary implementation is a LogicSQL query. LogicSQL is a logical language, part of the APPEAL programming environment [5]. It allows to interface SQL queries in Prolog. A LogicSQL clause has the same semantics of the corresponding SQL clause, with the addition of logical variables. These variables can be used in input or output, depending on their being ground or not at execution time.

The LogicSQL query, specified in the body of this `value` clause acts as a configurable input mask which gathers, at run-time, all possible variables' constraints and produces the appropriate SQL query. This query, in turn, will be sent to the appropriate database, according to the access methods inherited by the object.

An interesting aspect of a data dictionary object shown by the example is that some relation and field names in the LogicSQL query are expressed as Prolog variables. These variables will be instantiated at query execution-time by calling the Prolog predicates: `relation/3` and `field/3`.

This solution was adopted to solve the problem of databases, whose organization does not fit the data logical view in the application. This is true when a variable attribute does not have any direct counterpart in the database.

For example, in the database `regional_population`, which is used to compute the population variable, data for different years is stored in different relations. In other words, the attribute `time` does not correspond to a specific database field.

The `regional_population` data dictionary must then contain a mapping between the logical `time` values and the corresponding database relation names. These facts are:

```
relation(1, time(year:1991), relpop1991) &
relation(1, time(year:1992), relpop1992) &
relation(1, time(year:1993), relpop1993) &
```

The computation mechanism works accordingly to the LogicSQL data retrieval model: one tuple at a time is retrieved using Prolog backtracking. In our example, when the `time` attribute is left unbound, `regional_population` first computes all the tuples derived from the first relation `relpop1991`. Then it backtracks over the `relation` predicate, finds the second `relation` fact and builds a new SQL query containing the next available year and relation. This process goes on until no more `relation` facts are found.

When there is a mismatch between logical attributes and their representation in the database, a further problem arises in the computation of statistical operators, like aggregation. This is the case of the sex attribute in our example. Clearly these computations could not be translated into a single SQL query. The solution was to exchange data with the logical layer. More specifically, we added a rule to the data dictionary, which handles these cases by delegating them to an appropriate element of the operator knowledge base. In these cases, statistical computations over attributes are thus solved at the logical level.

Finally, the data dictionary solves the problems related to attribute granularity translations. Logical attributes often specify granularities which are not directly found in the database relations. A solution would be to let the database return the data with its stored granularity and then perform the necessary translations (mainly aggregations) at the logical level, using the operator knowledge base.

However, for efficiency reasons, IDEA implements this translation in the physical layer, using SQL. More specifically we stored a set of internal database relations, which define the correspondence between granularity values (`place_gra` and `age_gra` in the example). The computation of a variable then joins the relation containing the variable value with relations storing granularity mappings. The output result (the projected field in the SQL query) for an attribute with "translated" granularity is the field of the mapping table corresponding to the logical granularity.

7 Conclusions and Future Work

As pointed out in the introduction, intelligent data retrieval and indicator computations, are only the first steps of a management decision support system. We think that our implementation, based on knowledge representation, inference rules and logic programming, will make the data analysis and interpretation phase of the system easier to conceive, to design and to implement.

Another important aspect we have addressed in IDEA is the possibility of describing management indicators indipendently of the physical data used to compute them. However, this independence between *logical* and *physical* descriptions is not complete in the current implementation. Domain experts can completely define only indicators. Variables still require a knowledge base designer who can program their representations in the data dictionary.

It is in our future plans to attempt to overcome this problem by augmenting the power of the inference process that translates logical knowledge into its physical counterpart. The idea is to remove all variable references from the data dictionary, leaving

only attributes as communication vehicles between logical and physical representations.

While general performances are quite satisfactory, we have not yet done a complete study of IDEA's inefficiencies. This will certainly be one of the first goals, before proceeding to the second phase of the project.

8 Acknowledments

We thank the Epidemiological Observatory of Emilia Romagna, especially Barbara Curcio Rubertini, Eleonora Verdini, Lino Falasca and Massimo Negri who sponsored this project and gave us many interesting ideas about how a decision-support system in health care should work. This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant n. 92.01662.69 and by the Health Ministry of Emilia Romagna, Dr. Barbolini.

References

[1] M. Carlsson and J. Widen. Sicstus Prolog user's manual. SICS research report R88007C, Swedish Institute of Computer Science, PO Box 1263, S-164 28 Kista, Sweden, July 1990.

[2] DSlogics. Appeal 2.1 user's manual. CNR Technical Report of Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo R-04-092, DS logics S.r.l., Viale Silvani 1, 40123 Bologna ITALY, 1993.

[3] InferenceCorporation. Art: Reference manual, 2 vols. Technical report, Inference Corporation, Los Angeles, California, 1987.

[4] IntelliCorp. KEE software development system user's manual. Technical report, IntelliCorp, 1985.

[5] Ugo Manfredi and Mirko Sancassani. LogicSQL : Augmenting SQL with logic. Cnr technical report of progetto finalizzato sistemi informatici e calcolo parallelo, DS logics S.r.l., Viale Silvani 1, 40123 Bologna, ITALY, July 1993.

[6] B. McClay. A query server for diverse sources of data and knowledge. In *Proceedings of the first International Conference on the Practical Application of Prolog*, 1992.

[7] M. Minsky. A framework for representing knowledge. In P.H. Winston, editor, *The Psychology of Computer Vision*. McGraw Hill, 1975.

[8] Mirko Sancassani, Nenna Dore, and Ugo Manfredi. The Idea User Interface: the Power of Logic Programming in GUI Implementations. In *Proceedings of the second International Conference on the Practical Application of Prolog*, April 1994.

Optimal Management of a Large Computer Network with CHIP*

Massimo Fabris - Adriano Tirabosco - Carlo Chiopris

ICON s.r.l.

Lungadige Rubele, 34 - 37121 Verona - ITALY

Tel. +39 (0) 45 8004767 Fax +39 (0)45 8004770

e-mail: ITA0282@applelink.apple.com

Abstract.

We describe the analysis and the formulation in CHIP of a problem concerning the optimal management of a large inter-banking computer network. Banks use this network to send one another messages and applicative data.

During the analysis, many different policies of management of the computers have been defined and many strategies to obtain local and global optimal management of resources have been proposed. Each strategy can be represented as a constrained search problem. We present one of these problems in detail: the traffic produced by m software processes of the computer of a bank must be partitioned among n computers of the network. The optimising task is to minimise the unbalance of employment of the computers.

A prototype of the application has been developed in CHIP, an extension of Prolog combining the declarative aspects of logic programming with the efficiency of constraint solving techniques. The language resulted to be flexible and to permit rapid software development.

* This work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR.

1 Introduction

In this paper, we describe the analysis and the implementation in CHIP of an application supporting the management of a large inter-banking computer network. Banks use this network to send one another messages and applicative data. The application splits the traffic produced by one bank among the computers of the network. The optimising task is to minimise the unbalance of employment and the memory consumption in the computers.

The problem of the optimal location of resources of the network is gaining more and more in evidence to the management of the network. Due to the relevance of the data being transferred, the correct working of the network is fundamental. Since the number of banks using the network and the number of services they require is rapidly increasing, the management must locate resources without loosing sight of the future. On the other hand, the underutilization of computers is an economic damage to avoid. These contrasting requirements originate a set of constrained search problems. During the analysis, we defined many different policies of management of the computers and many strategies to obtain local and global optimal management of resources. Furthermore, the network is an entity which continuously changes structures and needs. The flexibility of the language is fundamental to develop an application in this context, but even more important is the rapid software development and maintenance. Another requirement on these optimisation applications is an acceptable efficiency to let them be used also as simulation tools. To meet all these needs, we chose to make use of the language CHIP v4.1 distributed by Cosytec.

CHIP is an extension of Prolog combining the declarative aspects of logic programming with the efficiency of constraint solving techniques. It has been designed to tackle real world constrained search problems. It reduces significantly the development time of these applications while preserving most of the efficiency of procedural languages.

The paper is structured as follows: section 2 briefly describes the language; section 3 introduces the characteristics of the network and the problem the application aims to solve;

section 4 explains the formalization in CHIP of the problem, section 5 reports the obtained computational results; section 6 contains some conclusive remarks.

2 The CLP Language CHIP

In spite of its elegance, readability and clean semantics, logic programming always suffered of two main problems: the limitations imposed by the uninterpreted terms of the Herbrand universe, and by its simple but uniform computation rule. The first problem was faced by the CLP scheme [12], an extension of the LP scheme where unification in the Herbrand universe has been replaced by the concept of constraint-solvability in some specific domain(s) of application. A partial solution to the second problem was indicated by Gallaire [9] and by Van Hentenryck [17], who advocated the use in logic programming of constraint manipulation and propagation techniques developed in the late 70s and early 80s by the artificial intelligence community [15, 14]. These two fields of research originated many new logic languages. One of the most interesting is CHIP [6, 17]. CHIP is a powerful language combining the declarative aspects of logic programming with the efficiency of constraint solving techniques. It extends Prolog by introducing three new computation domains in addition to Herbrand universe: finite domains for natural numbers, boolean terms and linear rational terms. For each of them CHIP uses specialised constraint solving techniques: local consistency techniques for finite domains, boolean unification for boolean terms and symbolic simplex-like algorithm for rational terms. In particular, the constraints over finite domain variables can be arithmetic ($>, \leq, =, \neq$), symbolic (*element, alldifferent, atleast, atmost* and many other), or user defined. CHIP has been designed to tackle real world constrained search problems. Such problems appear in areas as diverse as operation research, biology, finance, and hardware/software design. CHIP has been successfully applied to many industrial problems in the areas of planning, scheduling and circuit design [1, 2, 3, 4, 5, 16].

Logic programming is an appropriate language for stating combinatorial search problem: its relational form makes it easy to state constraints while its (don't know) non determinism removes the need for programming a search procedure. Unfortunately,

traditional logic programming languages can also be very inefficient largely because of a passive use of constraints as tests instead of an active use in pruning the search space. As a consequence, traditional logic programming languages lead to a *generate and test* approach with its well known performance problems. On the contrary, CHIP's philosophy might be stated as *constrain and generate* . To give an intuition of the difference between the two approaches, consider the following simple CHIP program, which is closely related to our application :

```
find_solution([A,B,C,D]):-
     [A,B,C,D] :: [0,1],
     6 #>= 3*A+5*B,          %   #>= and #= are operators on
     7 #>= 8*C+2*D,          %   linear terms on domain variables
     A+C #= 1,               %   corresponding to ≥ and =
     B+D #= 1,
     labelling([A, B, C, D]).
```

The goal `:-find_solution([A,B,C,D]).` is solved by the system through the following steps:

1) `[A,B,C,D]::[0,1]` declares that the variables A, B, C, D can be assigned to 0 or 1;

2) `6 #>= 3*A+5*B` is declared. The system tries to remove inconsistent values. No values are removed.

3) `7 #>= 8*C+2*D` is declared. The system removes the inconsistent value 1 from the domain of C and hence assign C to 0.

4) `A+0 #= 1` is declared. The system removes the inconsistent value 0 to the domain A and A is assigned to 1. Since A is a variable of constraint 2), this constraint is reconsidered (it is waked) and the inconsistent value 1 to B is removed and hence B is assigned to 0.

5) `0+D #= 1` is declared. The system removes the inconsistent value 1 from the domain D and hence D is assigned to 1.

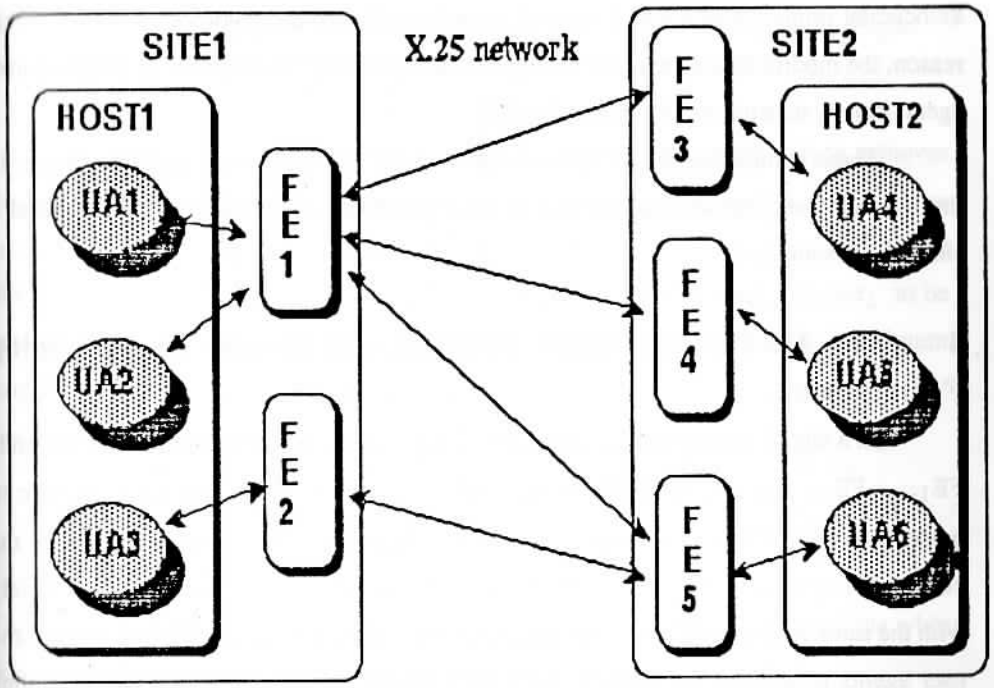
6) `labelling([1,0,0,1])` is executed. `labelling` is a built-in or user defined predicate that searches for all possible combinations of residual values in the domains of the variables which form a solution. During this search, a constraint is waked every time the domain of one of its variables is modified. In this particular case the only solution has been already found and hence no back-tracking steps are taken.

Notice that writing an active Prolog program or an procedural program comparable to this simple CHIP program involves the explicit coding of all tests. The user is free from programming the most tedious and time requiring parts of the code.

3 A Large Inter-Banking Computer Network and its Management

The application concerns the optimal management of a large inter-banking computer network. The network architecture is defined according to the ISO-OSI (International Standard Organisation - Open System Interconnection) standards [10, 11]. Banks use this network to send one another messages and applicative data. Every computer of the network, called front-end (FE), is located on a bank, called site, and it is connected to a computer of the bank (usually a mainframe), called host. The user agent (UA) is an applicative software located on the host, and it manages the transfer of data from and to the bank through the network: when a bank sends a message to another bank (called remote), it entrusts a user agent located on the host. This user agent relies on the services of a front-end located on the site. This front-end, following the instructions of the user agent, sends the message to the remote bank passing from the remote front-end to the remote user agent located on the remote host. Messages are measured in protocol data units (PDA) or bytes.

In the following, if UA relies on the services of FE, we say that UA is *associated* to FE. Every user agent is associated to a unique front-end, while a front-end can be associated to more than one user agent.



This set of associations must satisfy the constraint that the number of packet data units sent and received by a front-end must not be greater than a fixed maximum (per hour or per day). The number of packet data units sent and received by a front-end is the sum of the PDAs sent and received by the user agents associated to that front-end.

Any set of associations satisfying these constraint is not definitive. The most frequent reasons to change it are

a) the decreasing/increasing of traffic managed by some user agents, or the creation of new user agents;

b) the adoption of a different policy of management of the FEs. As examples of policies: downloading a particular set of FEs which supports critical applications; concentrating loads outside some hours to permit hardware and software maintenance.

All these goals can be achieved changing the association of some user agents. In the following we will say that (the association of) UA is *moved* from FE_1 to FE_2 if the association UA-FE_1 is replaced by the association UA-FE_2. However, changing an

association implies a substantial work of reconfiguration of the involved objects. For this reason, the modifications are made locally and incrementally. The number of involved user agents strictly depends on the obtained result.

These contrasting requirements originate a set of constrained search problems. In the following section we describe one of these problems through the formalization of a simplified example.

4 The AU-FE Association Problem. An Integer Programming Formulation.

On a site of the network, n user agents UA_1, ..., UA_n are associated to m front-ends FE_1, ..., FE_m. Due to a variation of the traffic managed by each user agent, we are in presence of an unbalance of employment of the front-ends. This is in contrast with the current policy of management which requires that they should be charged approximately with the same load of traffic. As a consequence, the management is thinking to move some user agents to achieve the balance. Since they cannot change all the associations, they impose the constraint the `MaxMoves` is the maximum number of user agents they will move.

We formulate the problem as an integer programming one. Let UA_{ij} be a boolean variable whose value is 1 if UA_j is associated to FE_i, and it is 0 otherwise. The set of variables UA_{ij} can be partitioned in two subsets **Old** and **New**. **Old** is the set of n variable UA_{ij} such that UA_j is actually associated to FE_i. This association is preserved iff $UA_{ij}=1$. **New** is the set of (m-1)*n variables UA_{ij} such that UA_j is not actually associated FE_i. Then UA_j is moved and associated to FE_i iff $UA_{ij}=1$. The constraint on the maximal number of moves `maxMoves` can be stated as

$$\sum_{UAij \in New} UA_{ij} \leq MaxMoves.$$

Since every user agent must be associated to a unique front-end, for every j we state the additional constraint

$$\sum_i UA_{ij} = 1.$$

If Trf_j is the quantity of traffic managed by UA_j, the total traffic assigned to FE_i is

$$\sum_j UA_{ij} * Trf_j.$$

Assume now that $AverageLoad_i$ is the load on front-end FE_i which achieves the balance of employment on the machines. Any additional traffic will be considered as an overload. It is important to measure the overload because this is the quantity we would like to minimise. $OverLoad_i$ is the minimal quantity to add to $AverageLoad_i$ to be greater or equal than the traffic assigned to FE_i. This traffic cannot exceed a maximum load $MaxLoad_i$. This can be expressed by the constraints:

$$AverageLoad_i + OverLoad_i \geq \sum_j UA_{ij} * Trf_j.$$

$$0 \leq AverageLoad_i + OverLoad_i \leq MaxLoad_i$$

The second constraint can be omitted declaring $OverLoad_i$ as a domain variable whose minimum is 0 and whose maximum is $MaxLoad_i - AverageLoad_i$.

The total overload on the site is defined as the sum of the overloads of the front-ends. This is the cost function.

$$TotOverLoad = \sum_i OverLoad_i$$

4.1 Exploiting CHIP Flexibility

The main drawback of this formulation is the number of variables it needs. In the most complex case we deal with 46 UA located on 10 FE, hence the boolean matrix has 460 elements. Using the flexibility of the language, we have tried to define an efficient labelling procedure to explore the boolean matrix. The boolean matrix has m rows (as many as the FE_j) and n columns (one for each UA_i). The m elements of column j are the m variables UA_{*j} relatives to UA_j while it is not true that the n elements of row i are the n variables UA_{i*} relatives to FE_i. This is due to the fact that in every column, the first variable is in the set **Old**, while all the others are in **New**. In this way, the actual association of UA_j is preserved iff the first variable of the columns is assigned to 1. The

columns of the matrix have been ordered according to the decreasing order on the traffic Trf_j managed by the UA_j, because of the better computational time results.

The predicate `search_matrix(MaxMoves,Matrix)` moves at most `MaxMoves` user agents in every possible way.

```
search_matrix(MaxMoves, Matrix):-
    select_columns(MaxMoves, Matrix, ReducedMatrix),
    search_reduced_matrix(ReducedMatrix).
```

The predicate `select_columns(MaxMoves, Matrix, ReducedMatrix)` selects in every way at most `MaxMoves` columns corresponding to the UAs to be moved. All the others associations are preserved simply assigning to 1 the first element of the column:

```
select_columns(0, Columns,[]):-
    !, dont_move_au(Columns).
select_columns(_, [],[]).
select_columns(N,[[1|_]|Columms], SelectedColumns):-
    select_columns(N, Columns, SelectedColumns).
select_columns(N,[[0|H]|Colums], [H|SelectedColumns]):-
    N1 is N-1,
    select_columns(N1, Columns, SelectedColumns).

dont_move_ua([]).
dont_move_ua([[1|_]|Columns]):-
    dont_move_ua(Columns).
```

Then search_reduced_matrix moves in every way the selected UAs:

```
search_reduced_matrix([]).
search_reduced_matrix([[Elem|RestOfColumn]|Columns]):-
    move_ua(Elem, RestOfColumn),
```



```
    search_reduced_matrix(Columns).
```

The element of the column to be labelled is popped from the column for detecting failure as soon as possible during unification. In fact, many variables are assigned through constrain propagation.

```
move_ua(1, _).
move_ua(0, [Next|RestOfColumn]):-
    move_ua(Next, RestOfColumn).
```

The labelling procedure searches for a feasible solution and estimates its cost through the call to the predicate `indomain/2` which assignees to `Cost` its minimal value. This value is the cost of the feasible solution.

```
labelling(Matrix, Cost):-
    search_matrix(Matrix),
    indomain(Cost, min).
```

To search for the optimal solution, we call the procedure:

```
min_max(labelling(Matrix, Cost), Cost).
```

The predicate `min_max` is a higher order predicate that performs a branch and bound over the search space to find the minimal cost among all feasible solutions.

Before illustrating the computational results, let us make some consideration on the use of constraints. While `labelling` performs the search for a new feasible solution, the domain of the `Cost` variable is restricted by the consistency techniques, and its minimal value increases. As soon as this minimal value exceeds the best value computed so far (the bound), a failure occurs and a new branch is explored. Moreover, every column is searched top-down until a variable is assigned to 1. The constraint $\sum_j UA_{ij}=1$ makes sure that only one element in the column is assigned to 1. Notice that

the `labelling` procedure implies that this constraint will be satisfied. However, because of this constraint, the system immediately detects failure when all the elements of a column (still to be searched) are assigned to 0.

To measure the effective pruning of this constraint, we considered a site with 23 user agents and we counted how many times the procedure `search_reduced_matrix/1` is called. If k is the maximal number of UA to be moved, in a full enumeration procedure this number is $\sum_{r=1}^{k} \binom{23}{r}$ We compare two formulations: C1 including all constraints of section 4, and C2 obtained by C1 omitting the $\sum_j UA_{ij}=1$ constraints.

MaxMoves	C1	C2	enumeration
2	27	85	276
3	51	683	2407
4	100	2002	10902
5	180	3496	44551
6	134	9769	145498
7	155	27217	390665
8	290	67086	880969

Unfortunately, the C1 formulation suffers of an useless reconsideration of constraints. The key point is that we are interested to detect failure: a complete column assigned to 0. This clearly is not the case if we assign a 1 in a column. The problem with the C1 formulation is that every time a variable is assigned to 1, all the other elements of the columns are assigned to 0 and this wakes the traffic constraints. Unfortunately this waking is useless since it does not change the minimum value of the $AverageLoad_i$ variables in the traffic constraints.

This problem is avoided with a little trick: in the C3 formulation, the constraint $\sum_j UA_{ij}=1$ is replaced by $\sum_j UA_{ij} \geq 1$. This constraint detects failure when

all variables are assigned to 0 but when one of them is assigned to 1 the other variables are not assigned to 0 and hence traffic constraints are not waked.

5 Computational Results

We compared the computational times obtained with CHIPv4.0 with the ones obtained with CPLEX, a tool for solving linear optimisation problems. CPLEX run on a Sun Spark 10/41 workstation, while CHIPv4.0 run on a HP 9000/715. CPLEX(1) indicates the results obtained with the default setting of parameters, while CPLEX(2) indicates the best results obtained setting the optional parameters (In particular, the best results where obtained basing the branch procedure on pseudo reduced costs).

In the following we report the computational times in seconds. Values reported inside brackets indicate the time spent before running out of the settled system limits, without finding the optimal solution.

If the site has n user agents and m front-ends the number of possible ways to move at most k user agents is $\sum_{i=1}^{k} \binom{n}{i} * (m-1)^i$. For n=23, m=7 and k=8 this number is approximately $2.5 * 10^{12}$.

MaxMoves	CHIP	CPLEX(1)	CPLEX(2)
2	6	0.39	0.61
3	12	1.02	1.11
4	21	2.52	12.61
5	29	10.71	4.04
6	28	17.28	9.80
7	41	41.63	26.87
8	82	452	201
9	70	(4331)	(4000)

First site: 22 UA associated to 7 FE

MaxMoves	CHIP	CPLEX(1)	CPLEX(2)
2	1.6	0.19	0.54
3	2.6	1.45	3.69
4	7	12	18
5	10	59	110
6	3	45	144
7	2.4	4	127
8	3	0.24	99

Second site: 21 UA associated to 5 FE

MaxMoves	CHIP	CPLEX(1)	CPLEX(2)
2	39	34	17.96
3	152	367	176
4	950	(1800)	(1445)
5	3092	(4200)	(4016)

Third site: 46 UA associated to 10 FE

The comparison shows that the problem solved with CHIP provides better average results on a wider set of inputs. We do not pretend this comparison to be exhaustive. We did not test an optional feature of the package that allows the user to choose the order in which variables are branched. On the other hand we did not test a completely different formulation with CHIP that make use of symbolic constraints (like element and cumulative) and it permits to use n domains variables whose domain is 1..m instead of the n*m boolean matrix leading to a dramatic decrease of the search space. We think that these preliminary results are encouraging and show how CHIP can be used to model and solve also linear problems, whenever deep user knowledge can be included.

6 Conclusions

We described the analysis and formulation in CHIP of a problem of optimal management of a large computer network. The application is at a prototype level but it has already confirmed that the language is flexible, efficient and it reduces significantly the development times. The network is an entity which continuously changes structures and needs. In this context, the flexibility of the language is fundamental, but even more important is the rapid software development and maintenance. From the analysis point of view, the main advantages of the CHIP system are:

1) it permits the rapid formulation and testing of the kernel of the algorithms. Usually more than one formulation is admissible. It also permits rapid formulation and testing of heuristics inside the search procedures. This features tight the cycle analysis - prototyping of the application.

2) it is suitable for the realisation of experimental prototypes to cope with boundary problems.

Acknowledgements. We would like to thank Riccardo Casiraghi and Arturo Bragaja for their contribution in analysing the problem, and Annalisa Bossi for her comments and helpful suggestions about the presentation of the paper.

References

[1] Boizumault P., Delon Y., Peridy L., *Solving a Real Life Exams Problem Using CHIP*, Proceedings of the International Logic Programming Symposium, pag. 661, Vancouver, October 1993.

[2] Bisdorf R., Gabriel A., *Industrial Linear Optimisation Problems solved by Constraint Logic Programming*, First International Conference on the Pratical Application of Prolog, London, April 1992.

[3] Dincbas M., Simonis H. and Van Hentenryck P., *Solving the Car Sequencing Problem in Constraint Logic Programming*. In European Conference on Artificial Intelligence (ECAI -88).

[4] Dincbas M., Simonis H. and Van Hentenryck P., *Solving a Cutting Stock Problem in Constraint Logic Programming*. In Fifth International Conference on Logic Programming, Seattle, WA, August 1988.

[5] Dincbas M., Simonis H. and Van Hentenryck P., *Solving Large Combinatorial Problems in Logic Programming*, Journal of Logic Programming, 8 (1-2) : 74-94, 1990.

[6] Dincbas M., Van Hentenryck P., Simonis H., Aggoun G., Graf T., and Berthier F., *The Constraint Logic Programming Language CHIP*. Proceedings of the International Conference on Fifth Generation Computer Systems (FGSC-88), Tokyo, Japan, November 1988.

[7] Fruhwirth T., Herold A., Kuchenoff V., Le Provost T., Lim P., Monfroy E. and Wallace M., *Constraint Logic Programming - An Informal Introduction*. ECRC Report, 2-2-95.

[8] File' G., Nardiello G., Tirabosco A., An Operational Semantics for CHIP, Proc. of Eight Conf. on Logic Programming - GULP'93, Gizzeria Lido, Italy, June (1993).

[9] H. Gallaire. *Logic Programming: further developments*. In IEEE Symposium on Logic Programming, pages 88-99. IEEE, Boston, July 1985.

[10] ISO OSI Application Layer Structure. ISO/DIS 9545

[11] ISO OSI SMI - Part 4 : Guidelines for the Definition of Managed Objects. ISO/DP 10165-4

[12] Jaffar J. and Lassez J.L., *Constraint Logic Programming*. In Proceeding of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany, pages 111-119. ACM, January 1987.

[13] Lassez C., McAloon K. and Yap R., *Constraint Logic Programming and OptionsTrading*, IEEE Expert, Special Issue on Financial Software, (3):42-50, August 1987.

[14] Mackworth A. K., *Constraint Satisfaction*, In Encyclopedia of Artificial Intelligence, 1986.

[15] Montanari U., *Network of Constraints: Fundamental Properties and Applications to Picture Processing*. Information Science, 7(2):95-132, 1974.

[16] Simonis H., *Test Generation Using the Constraint Logiic Programming CHIP*. In Proceedings of the 6th International Conference on Logic Programming, Lisbon, Portugal, June 1989.

[17] Van Hentenryck P. , *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA,1989.

A Debugging Model for Lazy Functional Logic Languages *

Puri Arenas-Sánchez, Ana Gil-Luezas

Universidad Complutense de Madrid, Departamento de Informática y Automática

Facultad de C.C. Matemáticas, Av. Complutense s/n, 28040 Madrid, Spain

email: {puri,anagil}@dia.ucm.es Fax: +34 1 3944607 Tel: +34 1 3944420

Abstract

In this paper we present a debugging model for a lazy functional logic programming language whose semantics is based on lazy narrowing under a particular strategy [1]: *Demand Driven Strategy (dds)*. The sophisticated operational semantics under *dds* -in computing head normal forms for total functions applications- makes necessary to develop a good debugging model which reflects the computational behaviour of programs, showing the different lazy unifications for demanded positions and the set of possible rules to apply depending on former lazy unifications. We present a debugging model based on Byrd's box model but incorporating new kind of boxes corresponding to the different steps followed by *dds* in computing goals. Moreover, the presented debugging model is also appropriate for other existing strategies for lazy narrowing, such as the *naïve strategy*.

1 Introduction

The amalgamation of lazy functions and higher order programming in a logic programming language raises special problems which have been analyzed in many recent proposals [10],[22],[9]. The use of lazy conditional term rewriting systems has been the base for the design of several functional logic languages such as K-LEAF [3],[10] and BABEL [22]. A common difficulty within all implementations of lazy narrowing consist of finding good control regimes which avoid repeated evaluations of arguments and *minimize* the risk of non termination. Currently, techniques of translation into Prolog are being used [24],[5],[1, 2],[17],[19] in order to help to solve the mentioned problems. However lazy strategies make quite hard to foresee the sequence of evaluations, since the evaluation of an expression is delayed until it is needed, depending also on the control of searching solutions. The sophisticated operational

*This research has been partially supported by the Spanish National Project TIC92-0793-C02-01 "PDR" and the Esprit BRA Working Group Nr. 6028 "CCL".

[1] We denote by *dds* to the lazy strategy along with the *demand driven regime control* presented in [19].